

Minimum Spanning Trees

Jared Saia

University of New Mexico

Today's Outline

- Minimum Spanning Trees
- Safe Edge Theorem
- Kruskal's and Prim's algorithms
- Graph Representation

Graph Definition

- A graph is a pair of sets (V, E) .
- We call V the vertices of the graph
- E is a set of vertex pairs which we call the edges of the graph.
- In an *undirected* graph, the edges are unordered pairs of vertices and in a *directed* graph, the edges are ordered pairs.
- We assume that there is never an edge from a vertex to itself (no self-loops) and that there is at most one edge from any vertex to any other (no multi-edges)
- $n = |V|$ is the number of vertices in the graph and $m = |E|$ is the number of edges

Graph Defns

For undirected graphs:

- A graph $G' = (V', E')$ is a *subgraph* of $G = (V, E)$ if $V' \subseteq V$ and $E' \subseteq E$ is a set of edges over the nodes in V'
- If (u, v) is an edge in a graph, then u is a *neighbor* of v
- For a vertex v , the *degree* of v , $\deg(v)$, is equal to the number of neighbors of v
- A *walk* is a sequence $v_0, \dots, v_k$ with an edge between each consecutive pair of vertices.
- A *path* is a walk, where the vertices visited are all distinct.
- A graph is *connected* if there is a path from any vertex to any other vertex
- *Connected components* are maximal connected subgraphs: two vertices are in the same component exactly when a path joins them.

Graph Defns

For undirected graphs:

- A *cycle* is a walk visiting at least 3 unique vertices that starts and ends at the same vertex, and where all vertices except the last visited are unique.
- A graph is *acyclic* if no subgraph is a cycle. Acyclic graphs are also called *forests*
- A *tree* is a connected acyclic graph. It's also a connected component of a forest.
- A *spanning tree* of a graph G is a subgraph that is a tree and also contains every vertex of G . A graph can only have a spanning tree if it's connected
- A *spanning forest* of G is a collection of spanning trees, one for each connected component of G

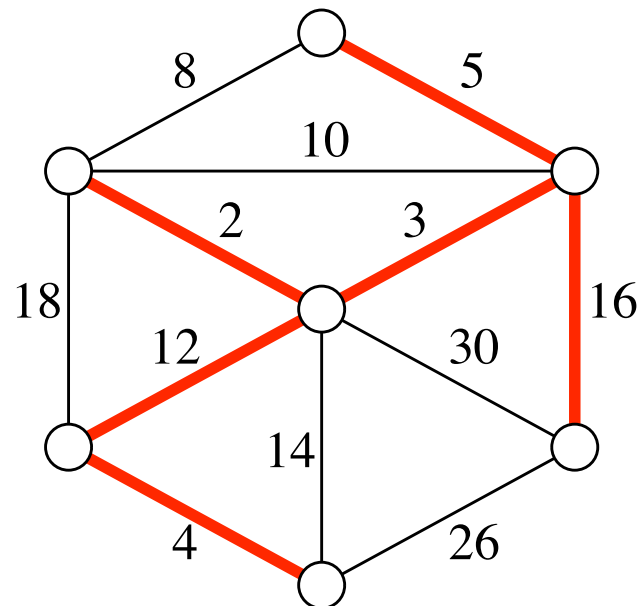
Minimum Spanning Tree Problem

- We are given a connected, undirected *weighted* graph.
- That is a graph $G = (V, E)$ together with a function $w: E \rightarrow \mathbb{R}$ assigning a *weight* $w(e)$ to each edge e .
- Our task is to find a *minimum spanning tree* of G , i.e., a spanning tree T minimizing

$$w(T) = \sum_{e \in T} w(e)$$

- Ties are allowed; an MST need not be unique.

Example



A weighted graph and its minimum spanning tree

Applications

- **Network design:** connect all sites at minimum total edge cost.
- **Clustering:** remove the $k - 1$ largest MST edges to obtain k clusters.
- **Metric TSP:** use an MST to obtain a tour costing at most twice the optimum, when distances satisfy the triangle inequality.

Generic MST Algorithm

Generic-MST(G, w)

$A \leftarrow \emptyset$

while $|A| < n - 1$ **do**

 Choose a safe edge $e \in E \setminus A$

$A \leftarrow A \cup \{e\}$

return A

Invariant: A is contained in some MST.

Safe edges - Definition

- Let A be any set of edges in G that is a subset of some MST of G
- An edge $e \in E \setminus A$ is *safe* for A if $A \cup \{e\}$ is also contained in some MST.

Other Definitions

- A *cut* $(S, V - S)$ of a graph $G = (V, E)$ is a partition of V
- An edge (u, v) *crosses* the cut $(S, V - S)$ if one of its endpoints is in S and the other is in $V - S$
- A cut *respects* a set of edges A if no edge in A crosses the cut.
- An edge is a *light edge* crossing a cut if its weight is the minimum of any edge crossing the cut

Safe Edge Theorem

Theorem 1 *Let A be a set of edges included in some minimum spanning tree. Then an edge e is safe for A if e is a light edge crossing some cut that respects A .*

Proof

Let T be an MST containing A . If $e \in T$, we are done. Otherwise, adding e to T creates a cycle containing another edge e' crossing the cut.

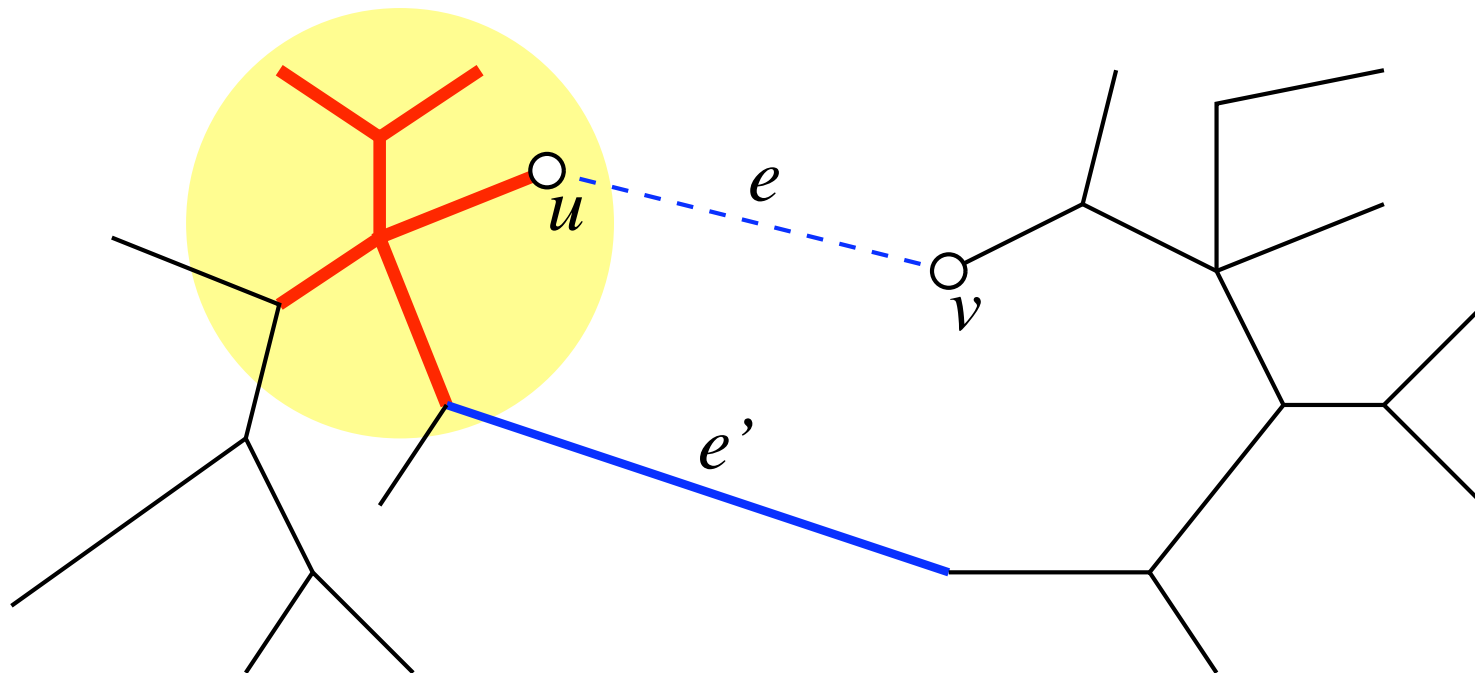
Since e is light, $w(e) \leq w(e')$. Exchanging the edges gives a spanning tree

$$T' = (T \setminus \{e'\}) \cup \{e\} \quad \text{with} \quad w(T') \leq w(T),$$

so T' is also an MST.

The cut respects A , so $e' \notin A$. Thus $A \cup \{e\} \subseteq T'$, proving that e is safe.

Example



The exchange preserves the red edges A : remove e' , then add e .

Corollary

Corollary 2 *Suppose A is contained in some MST, and C is a component of (V, A) . A minimum-weight edge leaving C is safe for A .*

Proof: The cut $(V_C, V \setminus V_C)$ respects A . A minimum-weight edge leaving C is light for this cut.

Two MST algorithms

- There are two major MST algorithms, Kruskal's and Prim's
- In Kruskal's algorithm, the set A is a forest. The safe edge added to A is always a least-weighted edge in the graph that connects two distinct components
- In Prim's algorithm, the set A forms a single tree. The safe edge added to A is always a least-weighted edge connecting the tree to a vertex not in the tree

Kruskal's Algorithm

- Q: In Kruskal's algorithm, how do we determine whether or not an edge connects two distinct connected components?
- A: We need some way to keep track of the sets of vertices that are in each connected component and a way to take the union of these sets when adding a new edge to A merges two connected components
- What we need is the data structure for maintaining disjoint sets (Union-Find) from the amortized-analysis lecture

Kruskal's Algorithm

Kruskal(G, w)

$A \leftarrow \emptyset$

for each $v \in V$ **do**

 MakeSet(v)

Sort E by nondecreasing weight

for each $(u, v) \in E$ in this order **do**

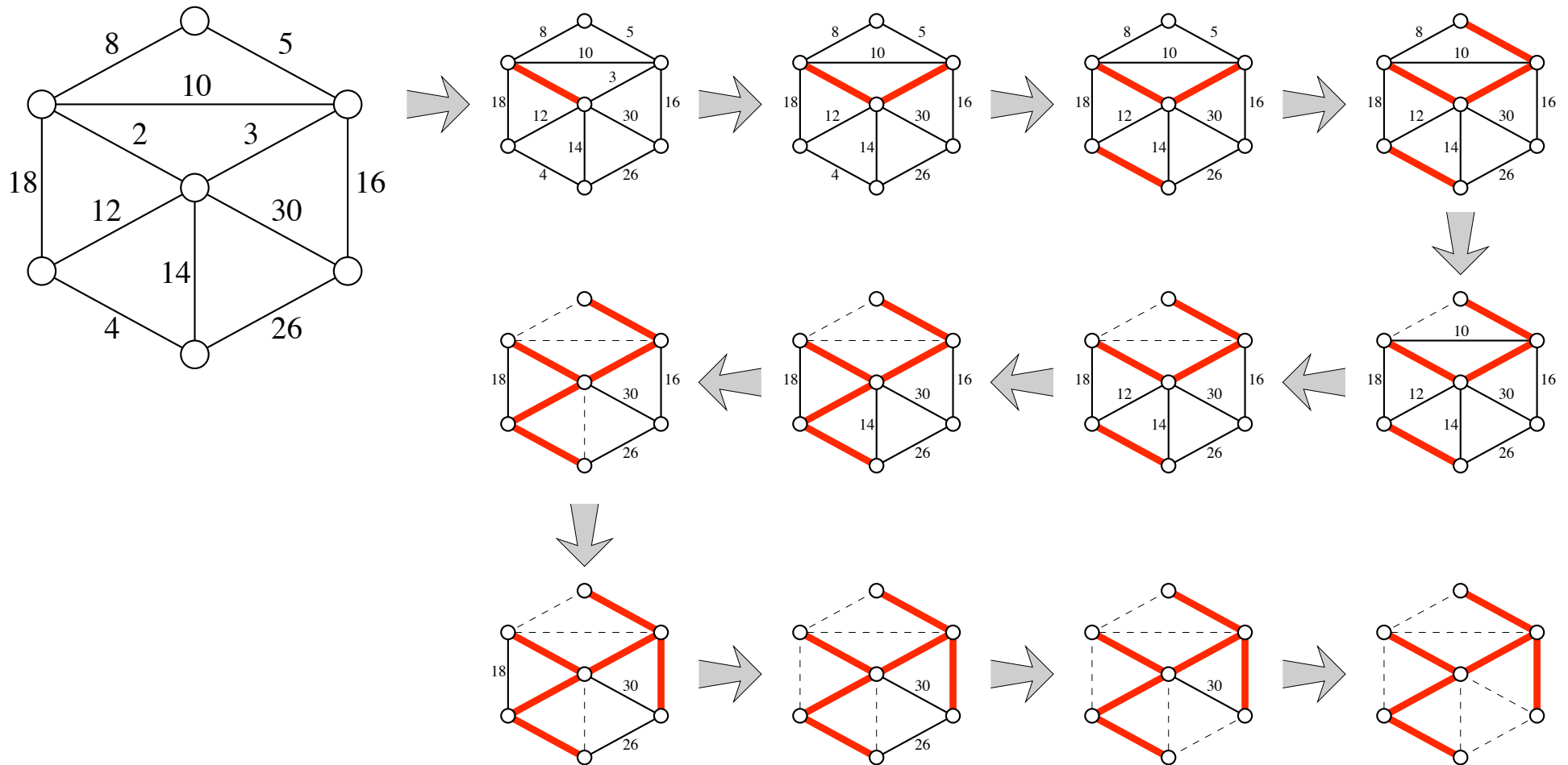
if FindSet(u) $\neq$ FindSet(v) **then**

$A \leftarrow A \cup \{(u, v)\}$

 Union(u, v)

return A

Kruskal's Algorithm



Thick edges are in A ; dashed edges are rejected.

Correctness?

- Correctness of Kruskal's algorithm follows immediately from the corollary
- Each time we add the lightest weight edge that connects two connected components, hence this edge must be safe for A
- At termination, A is connected: otherwise an edge between two components would have been added. Thus A is an MST.

Kruskal's Runtime

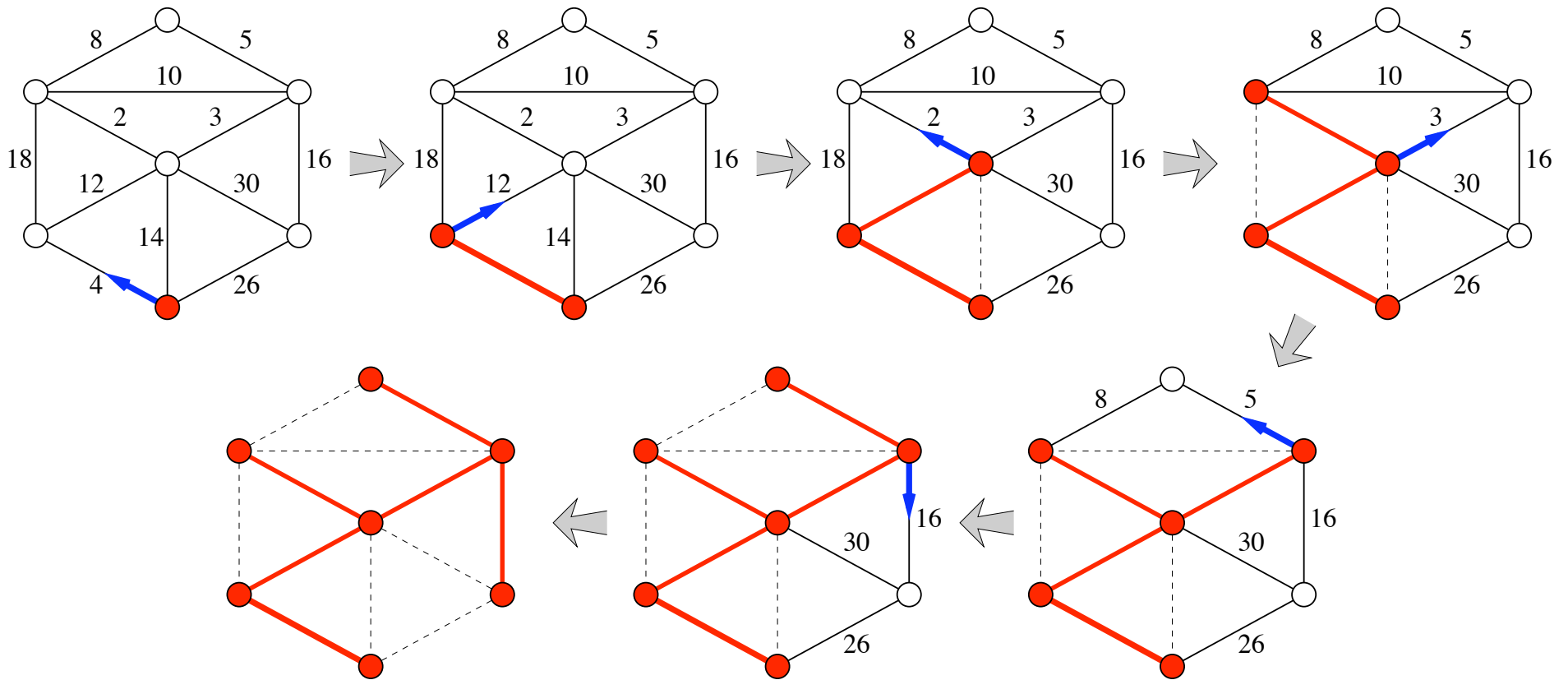
- Sort the edges in $O(m \log m)$ time.
- Use union by rank and path compression. As proved earlier, the amortized cost per operation is $O(\log^* n)$.
- The n MakeSet calls and $O(m)$ FindSet/Union calls take $O((n + m) \log^* n)$ total time.
- For a connected simple graph with $n \geq 2$, $n - 1 \leq m < n^2$. Sorting dominates, giving

$$\boxed{O(m \log n)}.$$

Prim's Algorithm

- In Prim's algorithm, the set A maintained by the algorithm forms a single tree.
- The tree starts from an arbitrary root vertex and grows until it spans all the vertices in V
- At each step, a light edge is added to the tree A which connects A to an isolated vertex of $G_A = (V, A)$
- By our Corollary, this rule adds only safe edges to A , so when the algorithm terminates, it will return a MST

Prim's Algorithm



Start at the bottom vertex; thick edges are in A ; arrows mark the next edge.

Prim with an Edge Heap

- Let S be the vertices already in the tree. Start with $S = \{s\}$ and put the edges incident to s in a min-heap.
- Extract a minimum-weight edge. Discard it if both endpoints are in S .
- Otherwise, add the edge and its new endpoint v to the tree. Insert edges from v to vertices outside S .
- A binary edge heap uses $O(m)$ space and $O(m \log n)$ time.

Prim with a Vertex Heap

- Keep one heap entry for each vertex outside the tree.
- For such a vertex v , $\text{key}[v]$ is the minimum edge weight from the tree to v , or ∞ if no such edge exists.
- $\text{parent}[v]$ records the tree endpoint of that edge. Adding a vertex may decrease its neighbors' keys.
- The heap uses $O(n)$ space. With a binary heap, the time is still $O(m \log n)$.

Prim: Initialization

Choose a start vertex s . The next slide continues this algorithm.

Prim(G, w, s)

$A \leftarrow \emptyset$

for each $v \in V$ **do**

$\text{key}[v] \leftarrow \infty$

$\text{parent}[v] \leftarrow \text{None}$

$\text{key}[s] \leftarrow 0$

$Q \leftarrow$ min-heap of all vertices, ordered by key

$\text{DecreaseKey}(Q, v, x)$ sets $\text{key}[v] \leftarrow x$ and updates the heap.

Prim: Main Loop

```
while  $Q \neq \emptyset$  do  
     $v \leftarrow \text{ExtractMin}(Q)$   
    if  $\text{parent}[v] \neq \text{None}$  then  
         $A \leftarrow A \cup \{(\text{parent}[v], v)\}$   
    for each  $(u, w(v, u)) \in \text{Adj}[v]$  do  
        if  $u \in Q$  and  $w(v, u) < \text{key}[u]$  then  
             $\text{parent}[u] \leftarrow v$   
             $\text{DecreaseKey}(Q, u, w(v, u))$   
return  $A$ 
```

Use weighted adjacency lists. An array stores each vertex's heap index (or None if removed), allowing $O(1)$ membership checks and entry lookup for DecreaseKey.

Prim's Runtime

- Adjacency lists let us scan each undirected edge twice: $O(n + m)$ total scanning time.
- There are n heap entries, n extractions, and at most m key decreases.
- **Binary heap:** each heap operation takes $O(\log n)$, giving $O(m \log n)$ total time.
- **Optional improvement:** a Fibonacci heap gives amortized $O(1)$ insertion and key decrease, and $O(\log n)$ extraction: $O(m + n \log n)$ total time.
- For $m = \Theta(n^2)$, these bounds are $O(n^2 \log n)$ and $O(n^2)$, respectively.

Note

- Prim's runtime uses adjacency lists to find the neighbors of a vertex efficiently.
- Let's compare adjacency lists with another common graph representation: an adjacency matrix.

Graph Representation

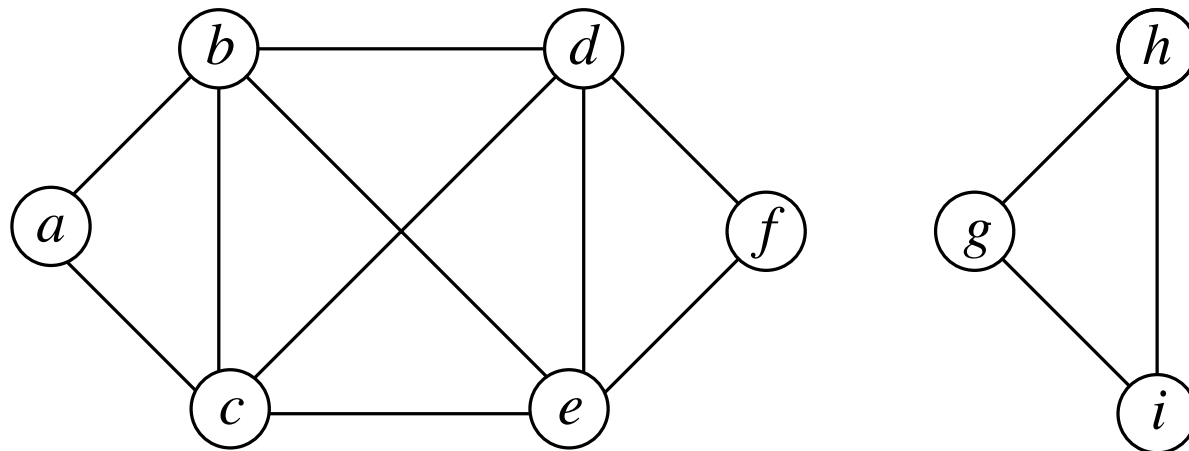
There are two common data structures used to represent graphs:

- Adjacency Matrices
- Adjacency Lists

Adjacency Matrix

- An unweighted adjacency matrix M is $n \times n$. Its entry $M[i, j]$ is 1 if $(i, j) \in E$, and 0 otherwise.
- For undirected graphs, M is *symmetric*: $M[i, j] = M[j, i]$. With no self-loops, $M[i, i] = 0$.
- For a **weighted** graph, store $w(i, j)$ for an edge and ∞ when there is no edge.
- Zero is a valid edge weight, so it cannot also mean “no edge.”

Example Graph



Example Representations

Adjacency matrix

	<i>a</i>	<i>b</i>	<i>c</i>	<i>d</i>	<i>e</i>	<i>f</i>	<i>g</i>	<i>h</i>	<i>i</i>
<i>a</i>	0	1	1	0	0	0	0	0	0
<i>b</i>	1	0	1	1	1	0	0	0	0
<i>c</i>	1	1	0	1	1	0	0	0	0
<i>d</i>	0	1	1	0	1	1	0	0	0
<i>e</i>	0	1	1	1	0	1	0	0	0
<i>f</i>	0	0	0	1	1	0	0	0	0
<i>g</i>	0	0	0	0	0	0	0	1	1
<i>h</i>	0	0	0	0	0	0	1	0	1
<i>i</i>	0	0	0	0	0	0	1	1	0

Neighbor lists

<i>a</i>	<i>b, c</i>
<i>b</i>	<i>a, c, d, e</i>
<i>c</i>	<i>a, b, d, e</i>
<i>d</i>	<i>b, c, e, f</i>
<i>e</i>	<i>b, c, d, f</i>
<i>f</i>	<i>d, e</i>
<i>g</i>	<i>h, i</i>
<i>h</i>	<i>g, i</i>
<i>i</i>	<i>g, h</i>

Both representations describe the graph on the preceding slide.

Adjacency Matrix

- Given an adjacency matrix, we can decide in $\Theta(1)$ time whether two vertices are connected by an edge.
- We can also list all the neighbors of a vertex in $\Theta(n)$ time by scanning the row corresponding to that vertex
- This is optimal in the worst case, however if a vertex has few neighbors, we still need to examine every entry in the row to find them all
- Also, adjacency matrices require $\Theta(n^2)$ space, regardless of how many edges the graph has, so it is only space efficient for very *dense* graphs

Adjacency Lists

- For *sparse* graphs — graphs with relatively few edges — we're better off with adjacency lists
- An adjacency-list representation stores one neighbor list per vertex. Each list can be a linked list or an array.
- For a weighted graph, $\text{Adj}[v]$ stores pairs $(u, w(v, u))$: a neighbor and the edge weight.
- An undirected edge appears in both endpoints' lists, so the lists contain $2m$ entries in total.

Adjacency Lists

- The total space required for an adjacency list is $O(n + m)$
- Listing all the neighbors of a node v takes $O(1 + \deg(v))$ time
- We can determine if (u, v) is an edge in $O(1 + \deg(u))$ time by scanning the neighbor list of u
- Note that we can speed things up by storing the neighbors of a node not in lists but rather in hash tables
- Then we can determine if an edge is in the graph in expected $O(1)$ time and still list all the neighbors of a node v in $O(1 + \deg(v))$ time

Kruskal and Prim: Summary

	Kruskal	Prim
Builds	A forest	One growing tree
Next edge	Lightest joining two components	Lightest leaving the tree
Data structure	Sorted edges; Union-Find	Vertex min-heap
Time	$O(m \log n)$	$O(m \log n)$ with a binary heap

Both preserve the invariant: A is contained in some MST. Ties may be broken arbitrarily.