Algorithms: Lecture 11

Chalmers University of Technology

Today's Topics

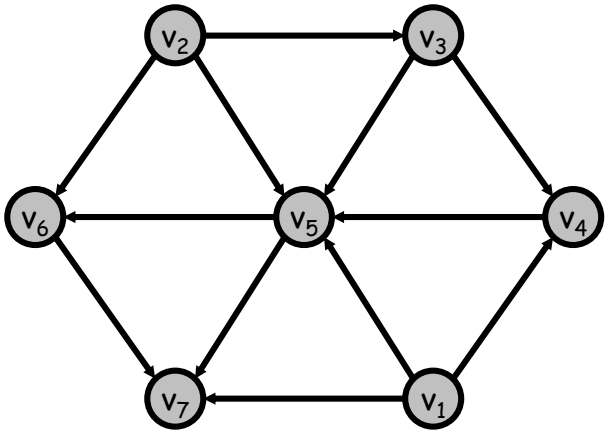
Directed Acyclic Graphs and Topological Ordering

Minimum Spanning Trees

- Prim's Algorithm
- Krusal's Algorithm

Demos and parts of some slides are adapted from https://courses.cs.washington.edu/courses/cse421/09wi/
http://www.vumultan.com/Books/CS702%20Handouts%20Version%201.pdf

DAGs and Topological Ordering



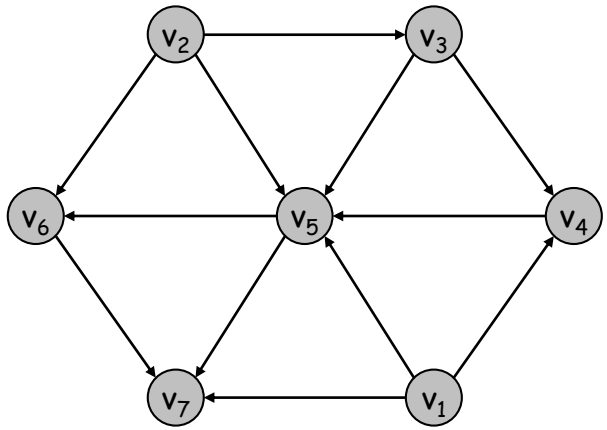
Directed Acyclic Graphs (DAG)

Def. A Directed Cycle in a directed graph is a cycle that can be traversed respecting the orientation of the edges: $v_1, v_2, v_3, \ldots, v_n, v_1$, where every (v_i, v_{i+1}) and (v_n, v_1) is a directed edge.

Def. A DAG is a directed graph that contains no directed cycles.

Given: a directed graph $G = (V, E)$.
Goal: Find a directed cycle in G, or report that G is a DAG.

Motivation: Precedence constraints: edge (v_i, v_j) means v_i must precede v_j.
Task v_i must be done before v_j
tasks can be calculations in a program or logic circuit, jobs in a project, steps in a manufacturing process, etc.
A directed cycle would indicate error in the design of such models



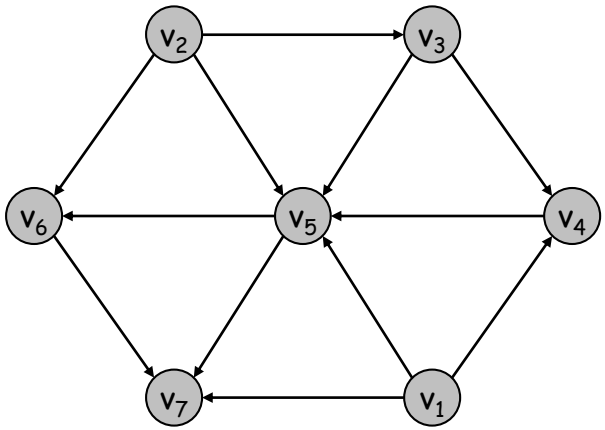
a DAG

Topological Order

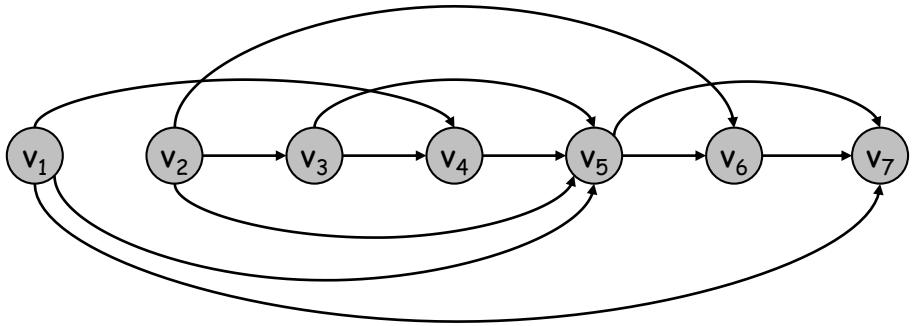
Def. A topological order of a directed graph $G = (V, E)$ is an ordering of its nodes as $v_1, v_2, \ldots, v_n$ so that all directed edges go to the right, that is for every edge (v_i, v_j) we have $i < j$.

Given: a directed graph $G = (V,E)$.
Goal: Construct a topological order of G, or report that G does not admit a topological order.

Motivation: Nodes are the tasks with pairwise dependency constraints.
Existence of a topological order for G guarantees it is a DAG



a DAG

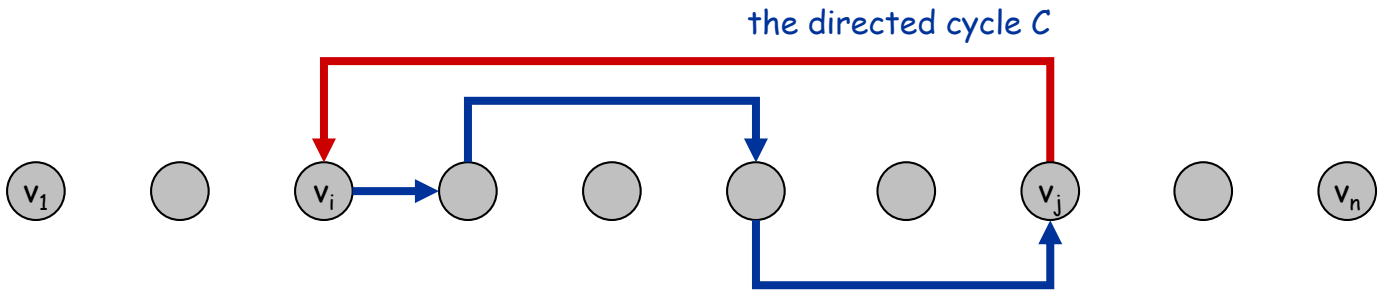


a topological ordering

Directed Acyclic Graphs

If G has a topological order, then G is a DAG.

- Topological order allows only edges from left to right
- With this we can never close a cycle.
- Closing a cycle would require an edge from right to left which is against the definition of topological order.



the supposed topological order: $v_1, \ldots, v_n$

Directed Acyclic Graphs

Q. Does every DAG have a topological ordering?

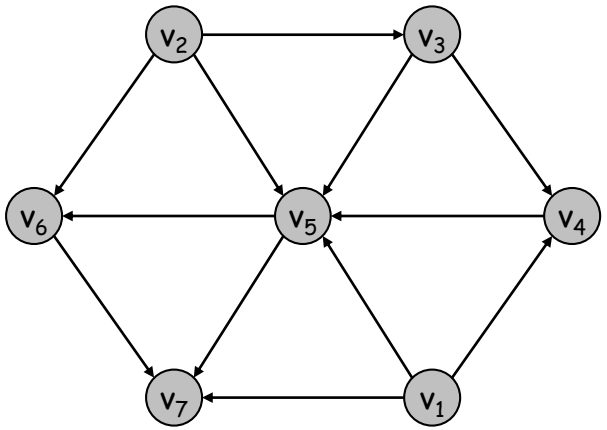
Q. If so, how do we compute one?

Stronger Version: G is a DAG if and only if G has a topological order

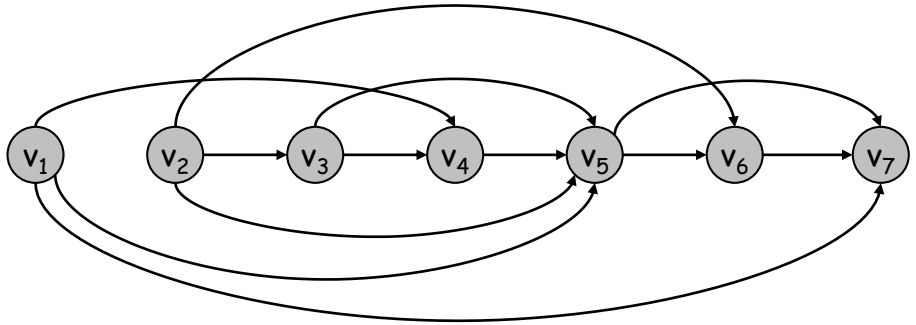
- If part (If G has a topological order, then G is a DAG.)... done already
- Only if: If G is a DAG, then G has a topological ordering.

Observation:

A topological ordering must start with a node with NO incoming edges.



a DAG



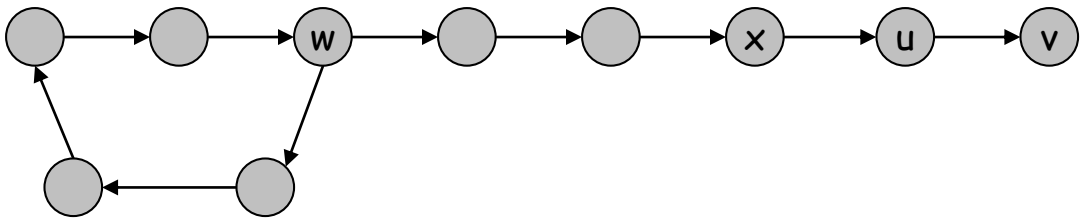
a topological ordering

Directed Acyclic Graphs

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. ▪



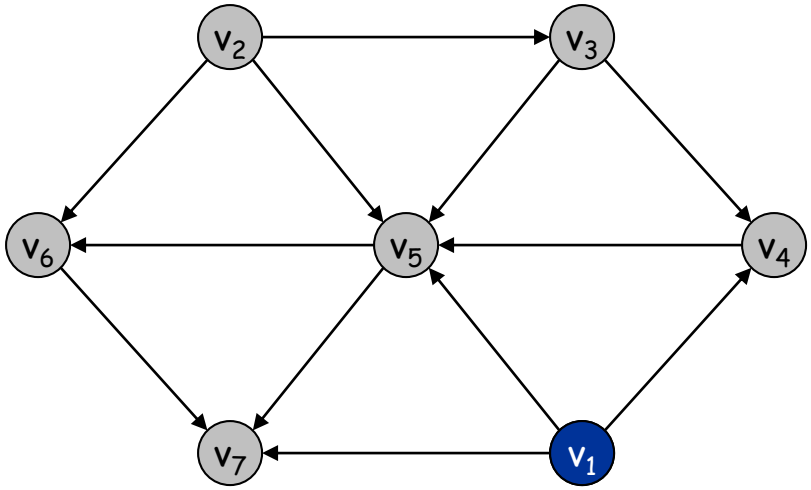
Directed Acyclic Graphs

If G is a DAG, then G has a topological ordering.

Pf.

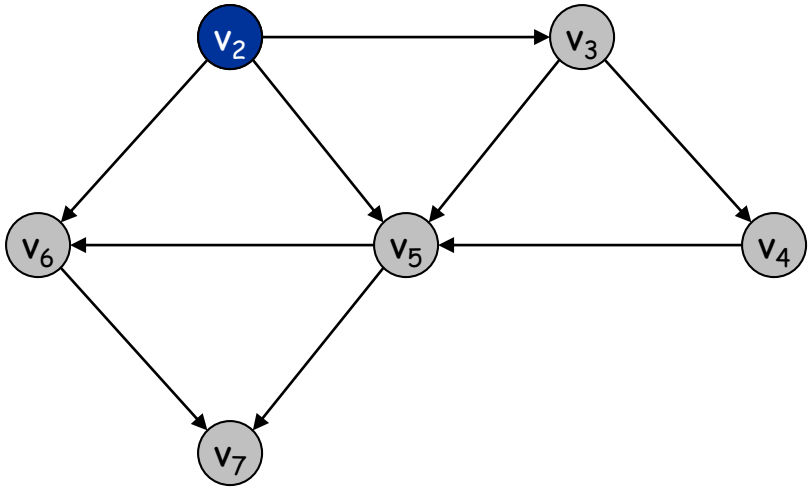
- Lets solve an example first: Given a DAG we find out topological ordering.
- Then, we proceed to the formal proof.

Topological Ordering Algorithm: Example



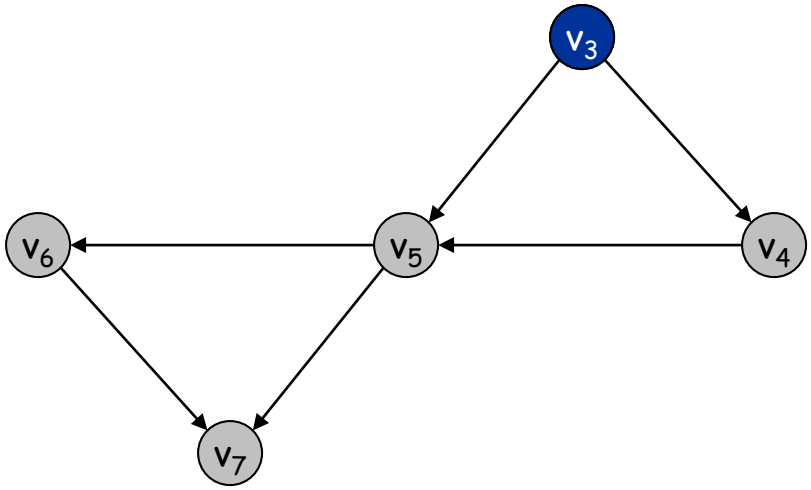
Topological order:

Topological Ordering Algorithm: Example



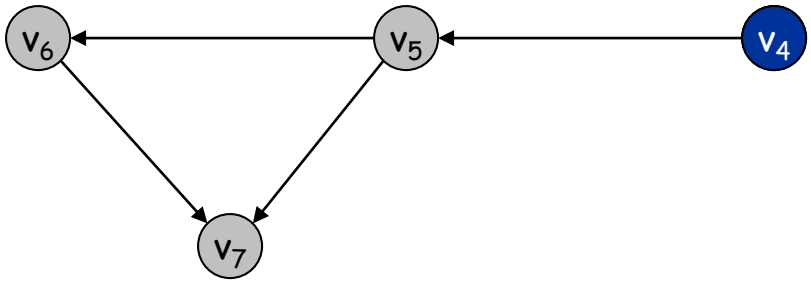
Topological order: v_1

Topological Ordering Algorithm: Example



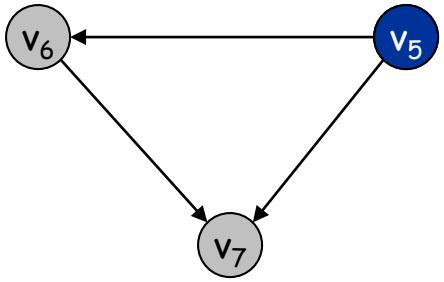
Topological order: v_1, v_2

Topological Ordering Algorithm: Example



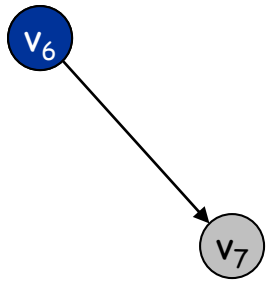
Topological order: v_1, v_2, v_3

Topological Ordering Algorithm: Example



Topological order: v_1, v_2, v_3, v_4

Topological Ordering Algorithm: Example

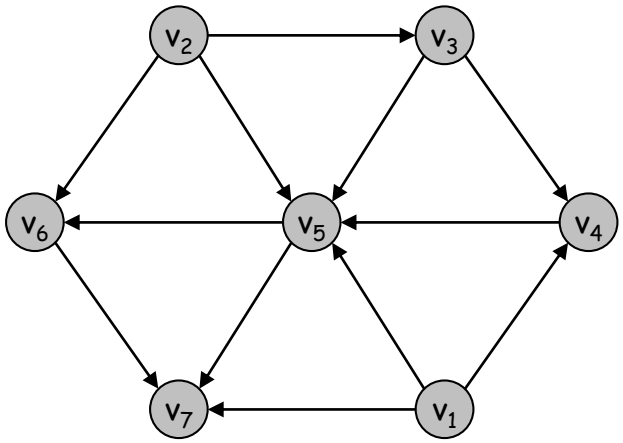


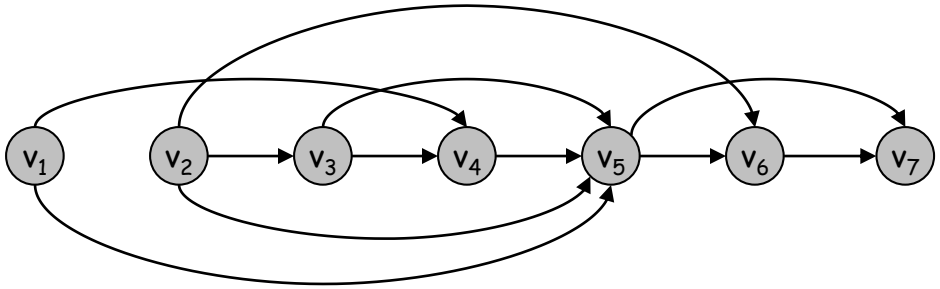
Topological order: v_1, v_2, v_3, v_4, v_5

Topological Ordering Algorithm: Example

Topological order: $v_1, v_2, v_3, v_4, v_5, v_6$

Topological Ordering Algorithm: Example





Topological order: $v_1, v_2, v_3, v_4, v_5, v_6, v_7$.

Directed Acyclic Graphs

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. ▪

Directed Acyclic Graphs

To compute a topological ordering of G: (already practiced on the example)

- 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

Seems like a Greedy Algorithm, however, nothing to optimize

- All we need to do is to chose an arbitrary node v with in-deg(v) = 0

How to efficiently maintain an updated degree for all current nodes in G?

- Queueing & Counting

Topological Sorting Algorithm: Running Time

Maintain the following information:

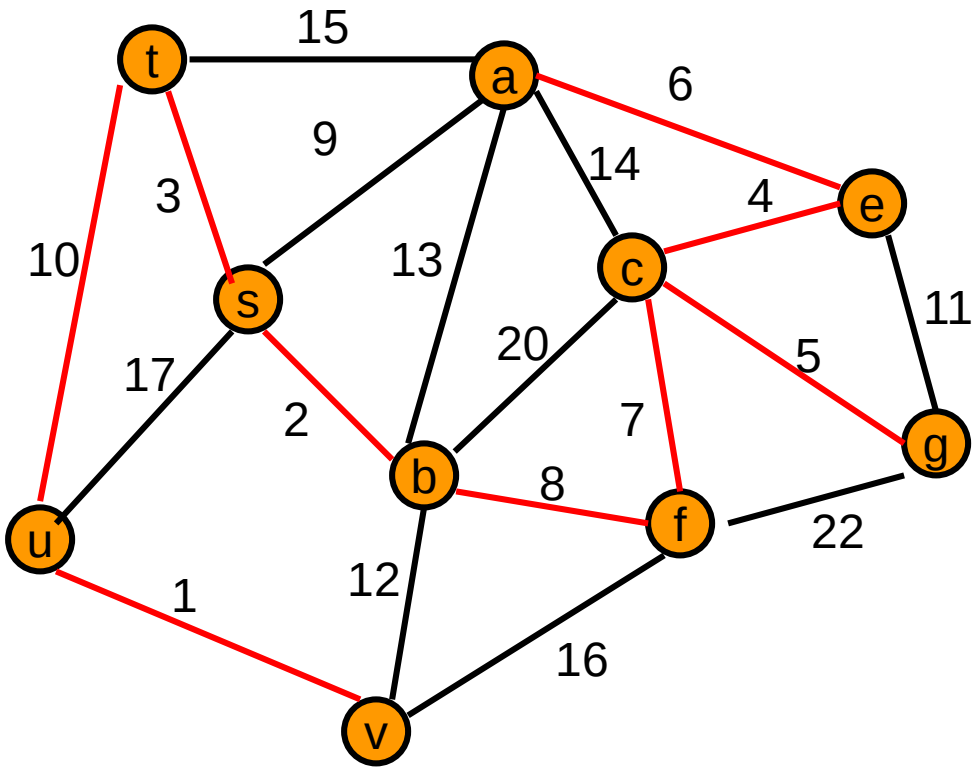
- count[v] = remaining number of incoming edges, in-degree(v)
- Q= queue of remaining nodes with no incoming edges, in-degree(v)=0

- Initialization: $O(m)$ via single scan through graph.

- Update: to delete u
  - remove u from Q
  - decrement count[v] for all edges from u to v, and
  - add v to Q if count[v] hits 0
  - this is $O(1)$ per edge ONCE
- Updating in-degrees costs $O(m)$ in total: G as Adjacency lists

  - The order of removal from Q gives the Topological Ordering

Minimum Spanning Trees

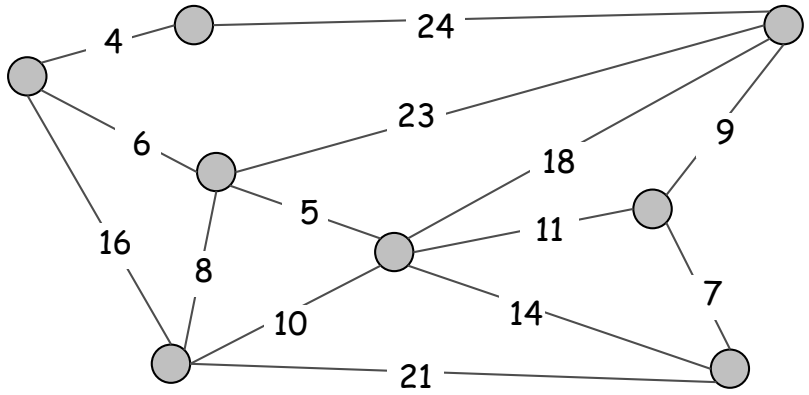


Minimum Spanning Tree (MST)

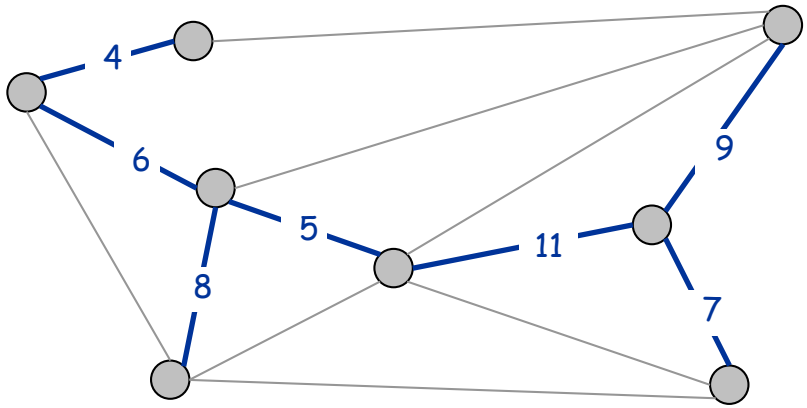
Given: a connected undirected graph $G = (V,E)$ where every edge has some positive cost c_e (also called weight)

Minimum spanning tree (MST): an MST is a subset of the edges $T \subseteq E$ such that T is a spanning tree (it "spans" all nodes in V) whose sum of edge weights is minimized.

Goal: Find an MST in G.



$G = (V, E)$



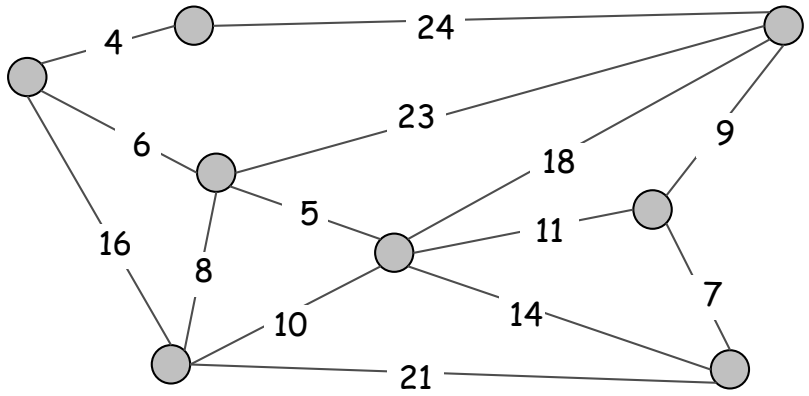
T, $\Sigma_{e \in T} c_e = 50$

Applications

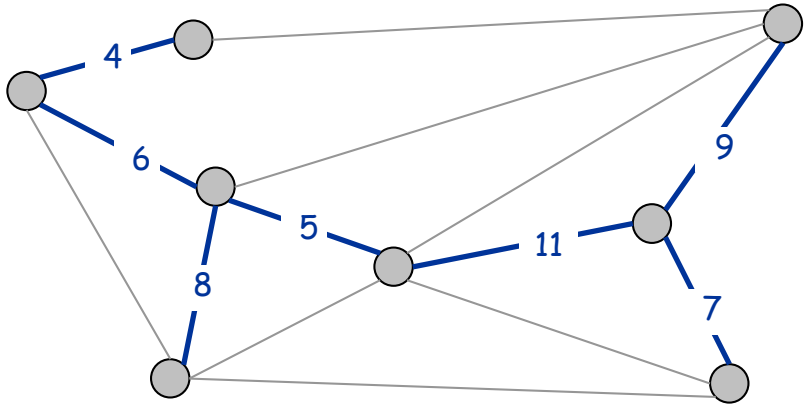
MST is fundamental problem with diverse applications.

- 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.

Minimum Spanning Tree (MST)



$G = (V, E)$



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

Cayley's Theorem. There are n^{n-2} spanning trees in a complete graph of n nodes

can't solve by brute force

Problem Analysis:

Which edges should be chosen by an MST?

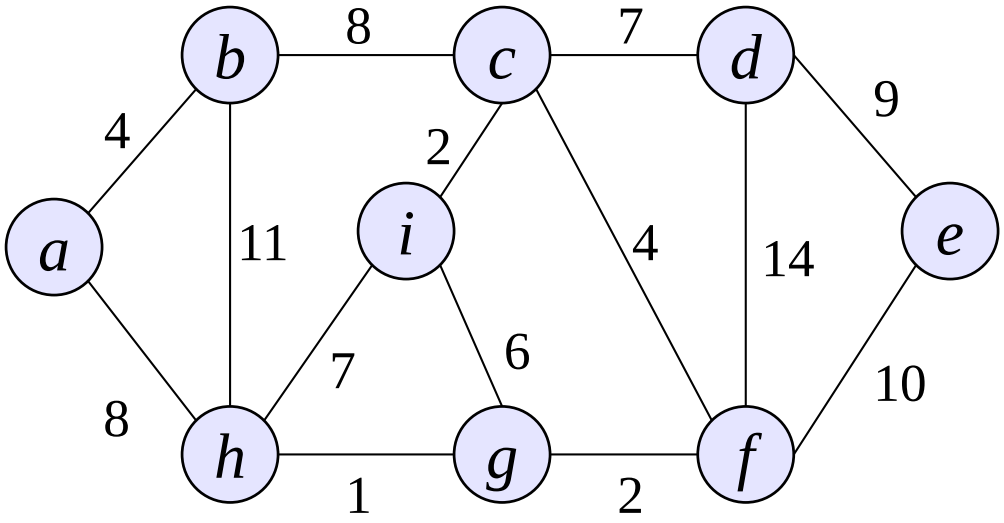
Intuitively: Those with min cost, to get an overall min.

Precisely: If e is a cheapest edge in G, then e belongs to some MST.

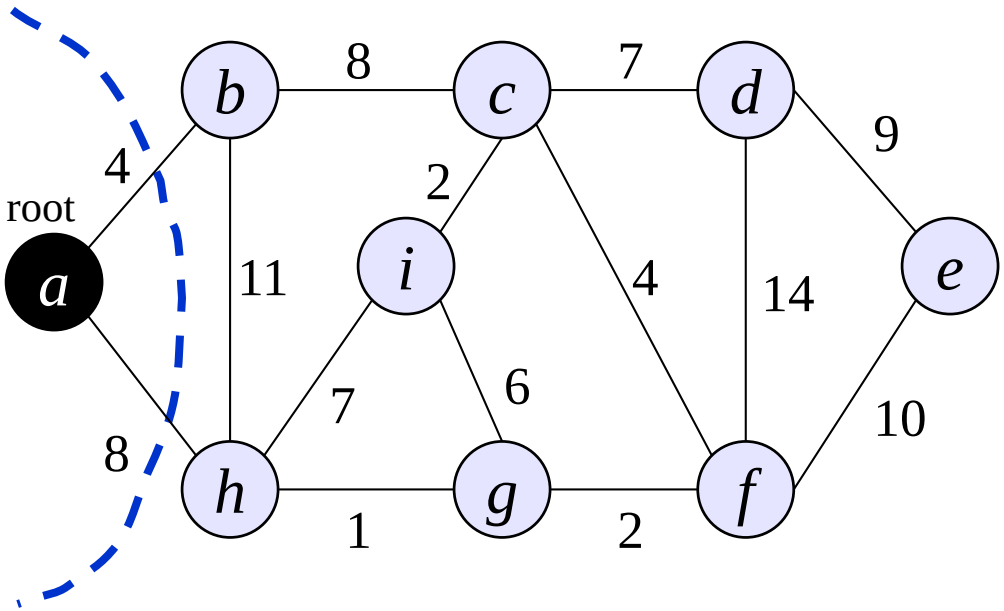
- Let T, an MST, does not include e.
- Add e to T $\Rightarrow$ T has a cycle C involving e.
- Remove some other edge from C, get a better T.

Greedy Algorithm?

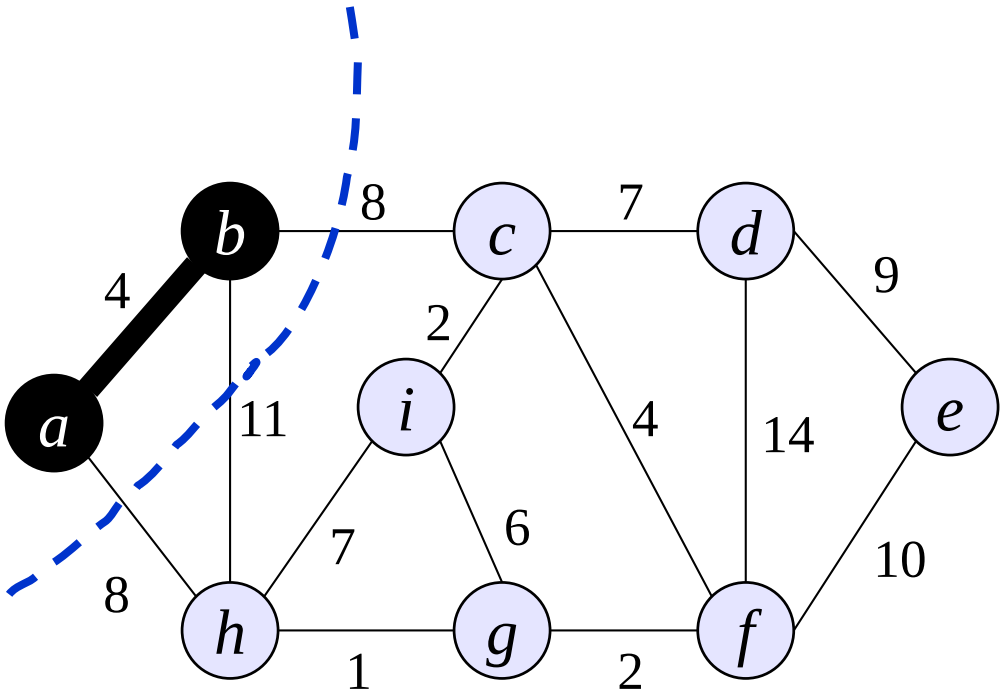
MST - Greedy Algorithms



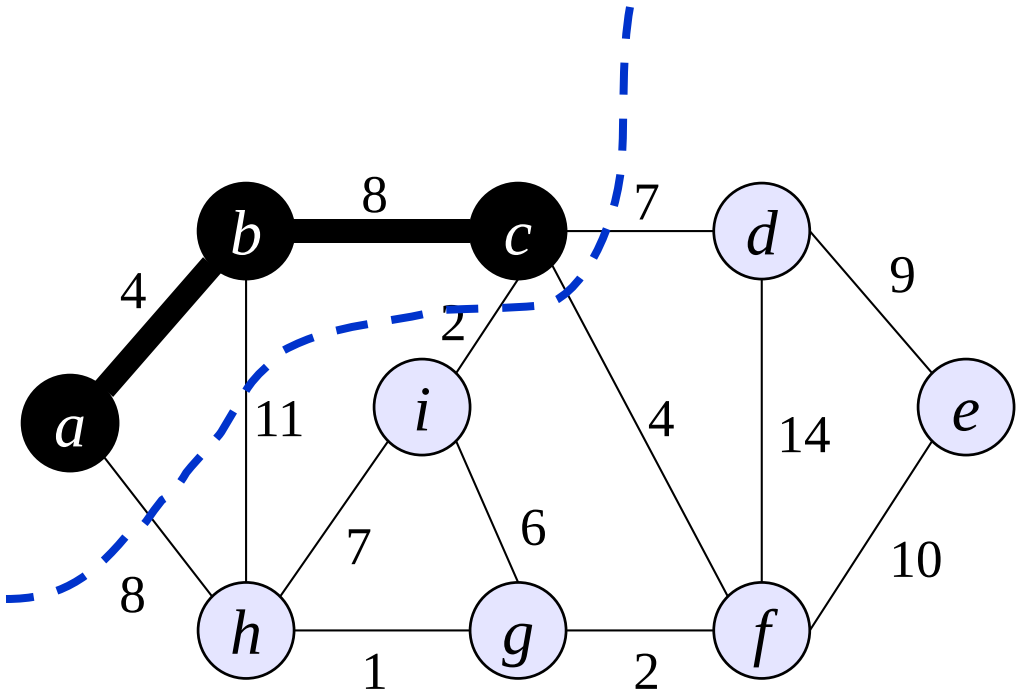
MST - Greedy Algorithms



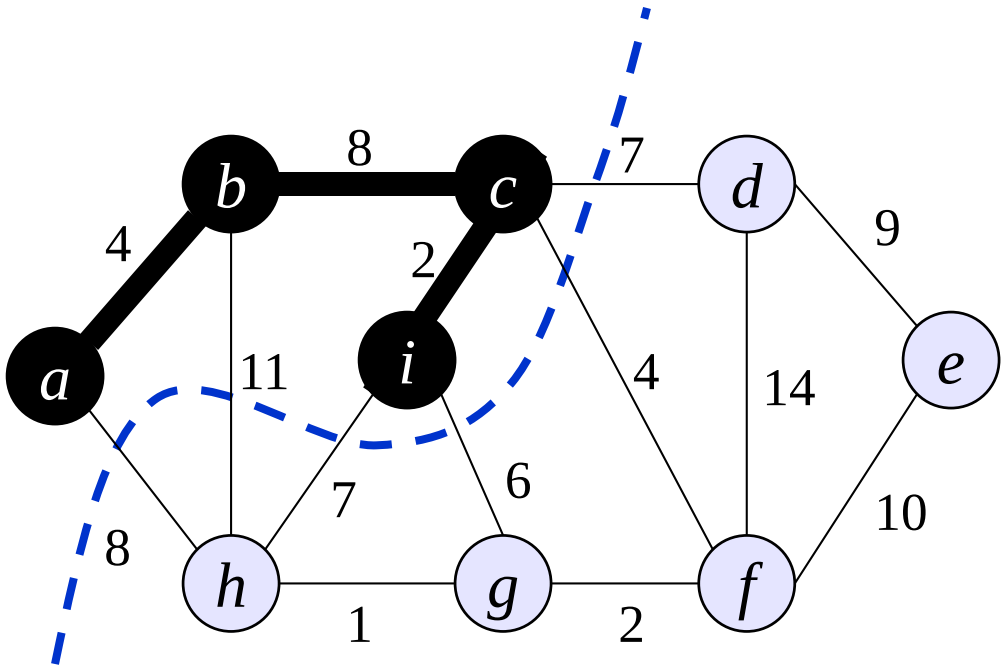
MST - Greedy Algorithms



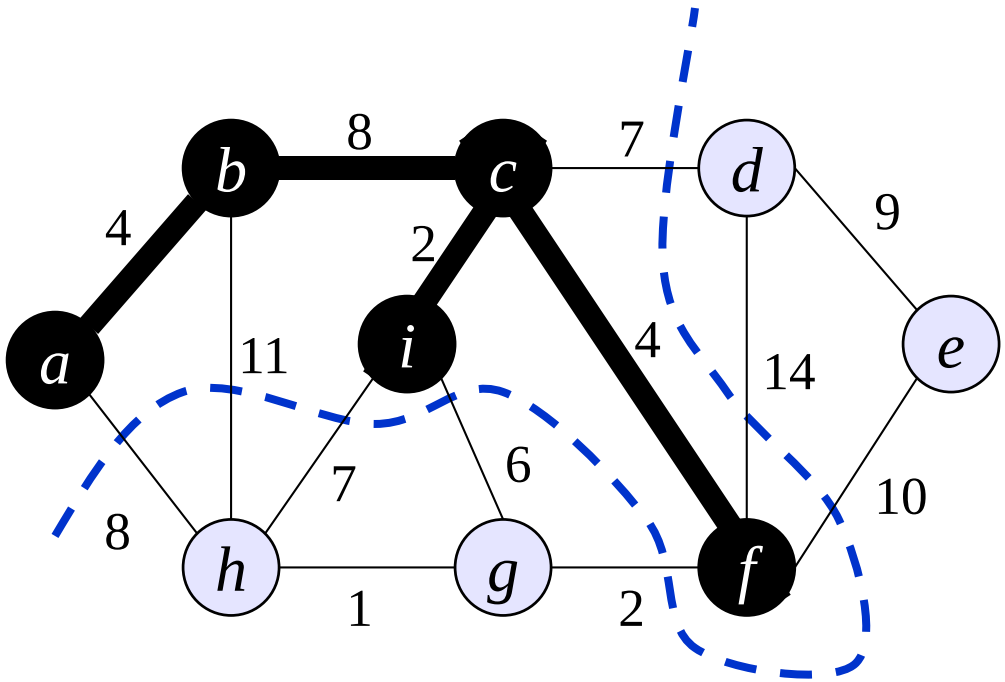
MST - Greedy Algorithms



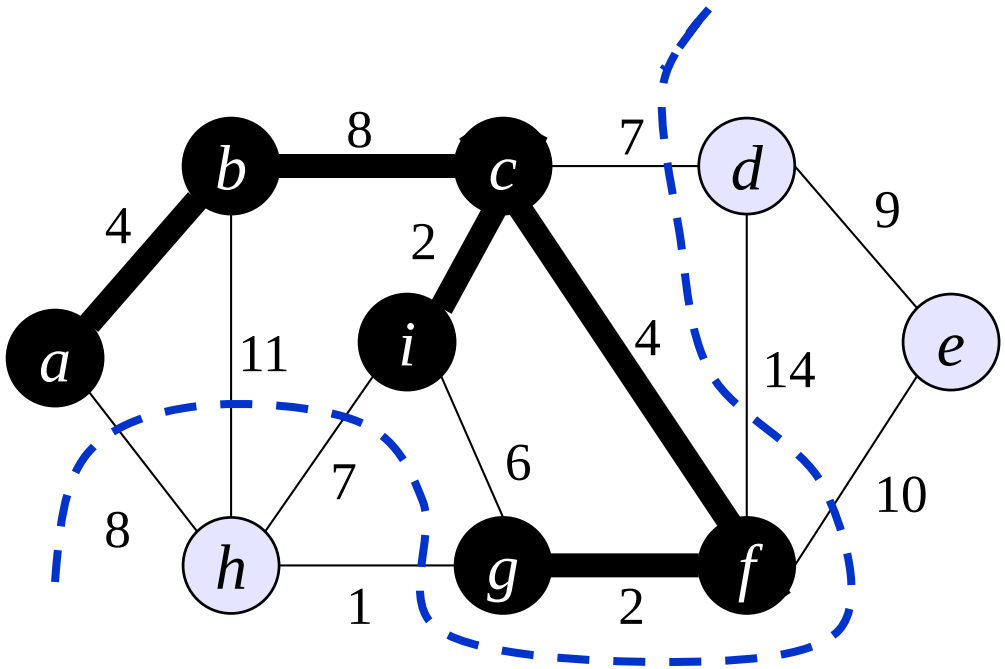
MST - Greedy Algorithms



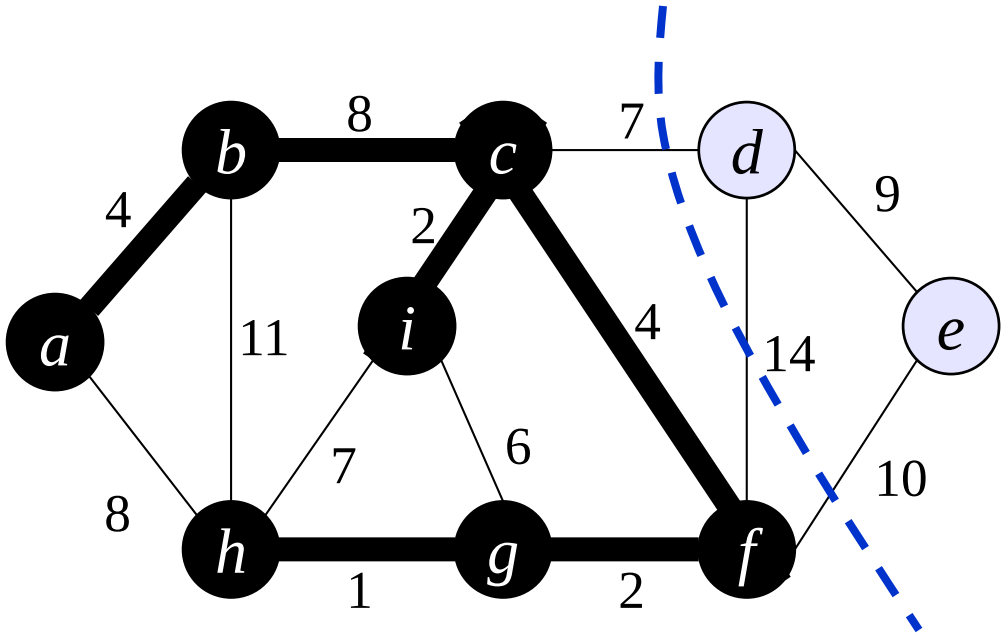
MST - Greedy Algorithms



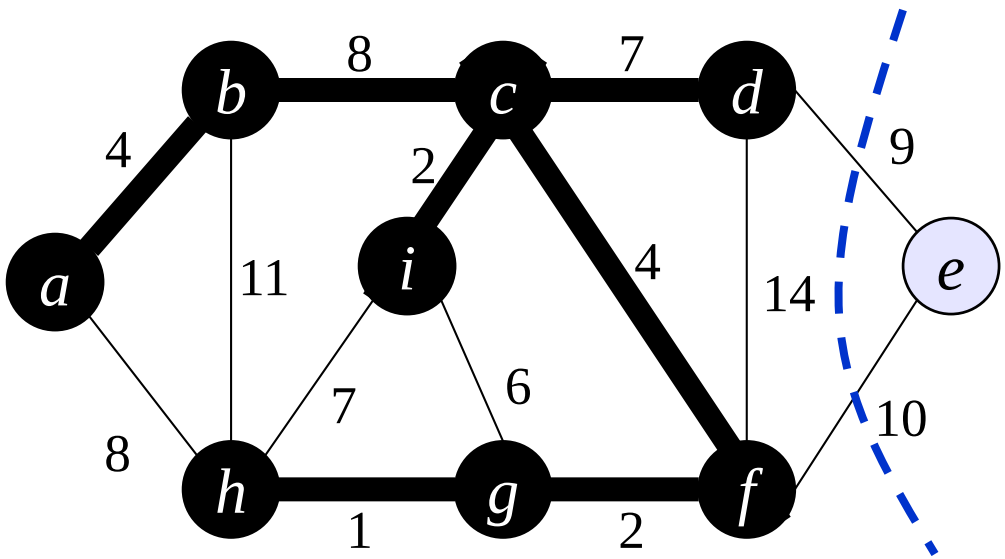
MST - Greedy Algorithms



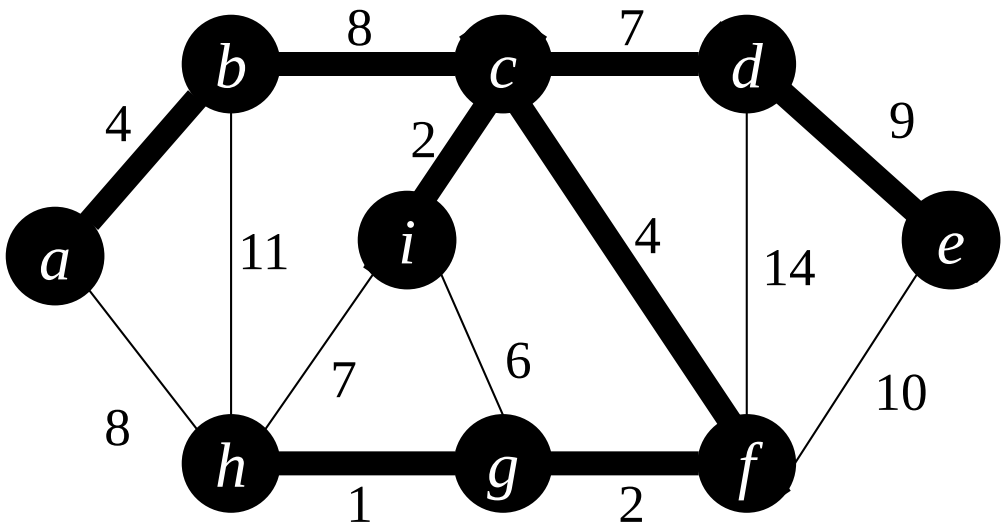
MST - Greedy Algorithms



MST - Greedy Algorithms



MST - Greedy Algorithms

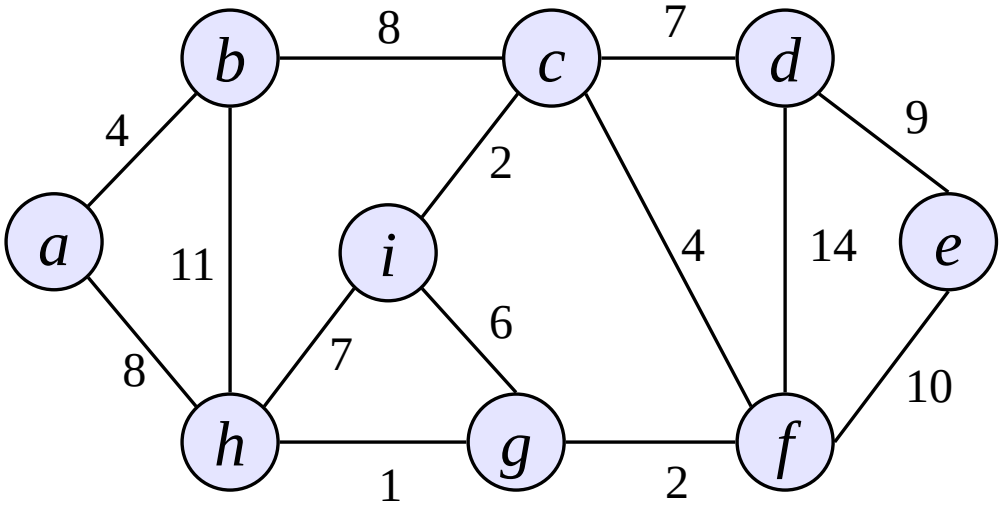


MST - Greedy Algorithms

The greedy rule that we followed in the demo on previous slides can be formulated as:

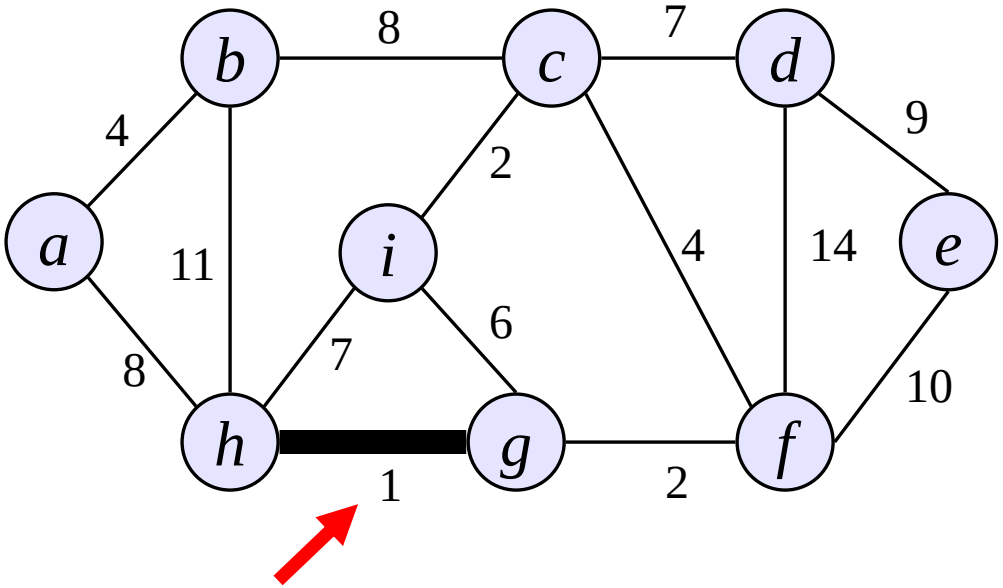
- Start at some root node s and greedily grow a tree T from s outward.
- At each step, add the cheapest edge e to T having one endpoint in T.
- Both edges in T would mean T gets a cycle.

Prim's Algorithm for finding an MST



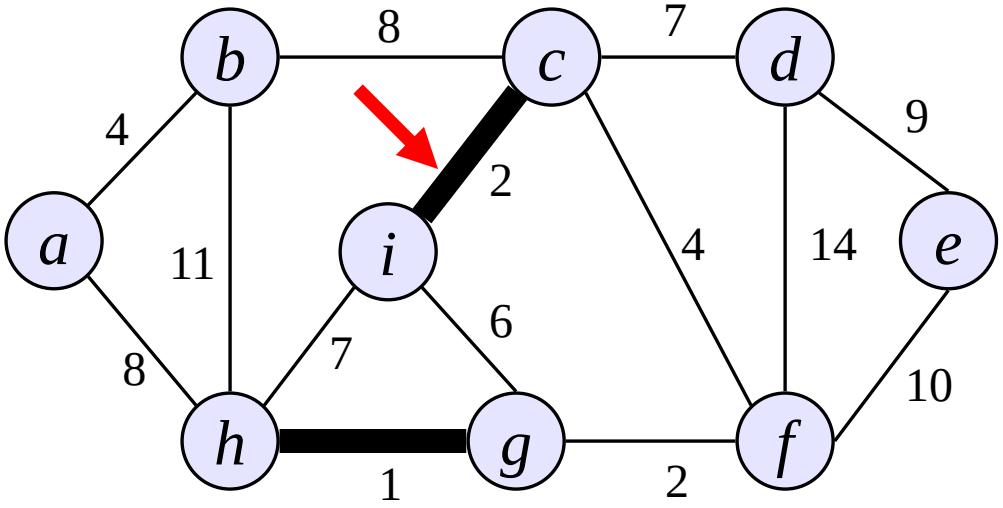
Initial sets = *{a}, {b}, {c}, {d}, {e}, {f}, {g}, {h}, {i}*

Edges	Weight
(g, h)	1
(c, i)	2
(f, g)	2
(a, b)	4
(c, f)	4
(g, i)	6
(c, d)	7
(h, i)	7
(a, h)	8
(b, c)	8
(d, e)	9
(e, f)	10
(b h)	11
(d, f)	14



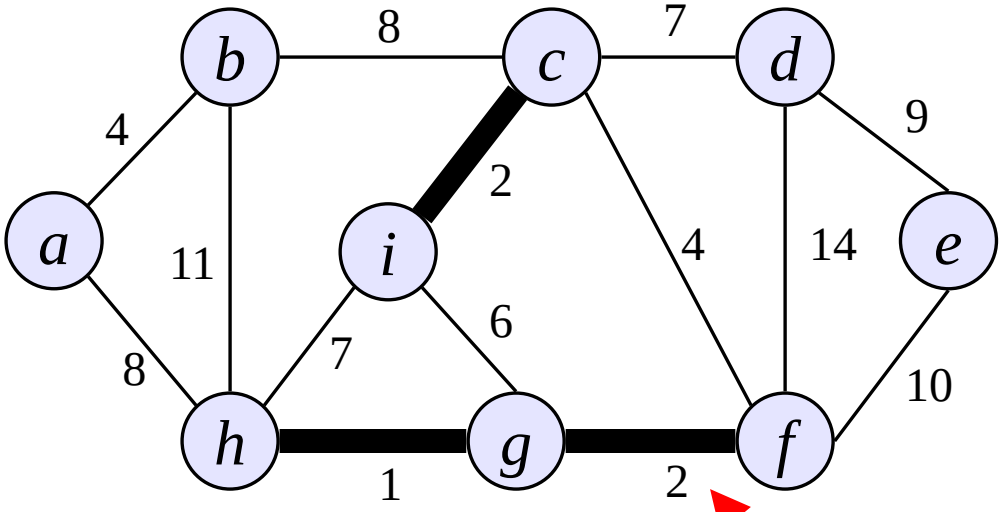
Initial sets = *{a}, {b}, {c}, {d}, {e}, {f}, {g}, {h}, {i}*

Final sets = *{a}, {b}, {c}, {d}, {e}, {f}, {g, h}, {i}*



Initial sets = *{a}, {b}, {c}, {d}, {e}, {f}, {g, h}, {i}*

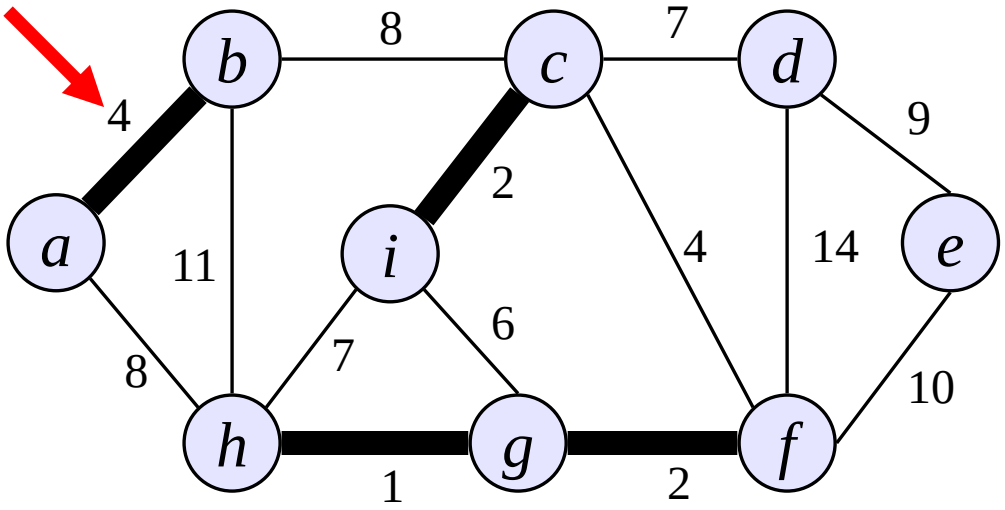
Final sets = *{a}, {b}, {c, i}, {d}, {e}, {f}, {g, h}*



Initial sets = *{a}, {b}, {c, i}, {d}, {e}, {f}, {g, h}*

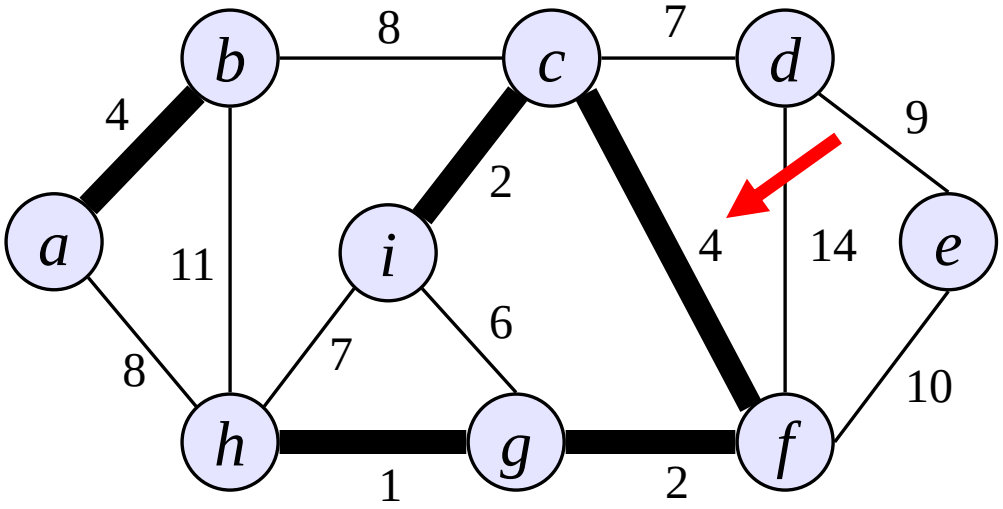
Final sets = *{a}, {b}, {c, i}, {d}, {e}, {f, g, h}*

- In intermediate steps, we don't get a tree actually.
- A collection of connected components of edges, called a "Forest"



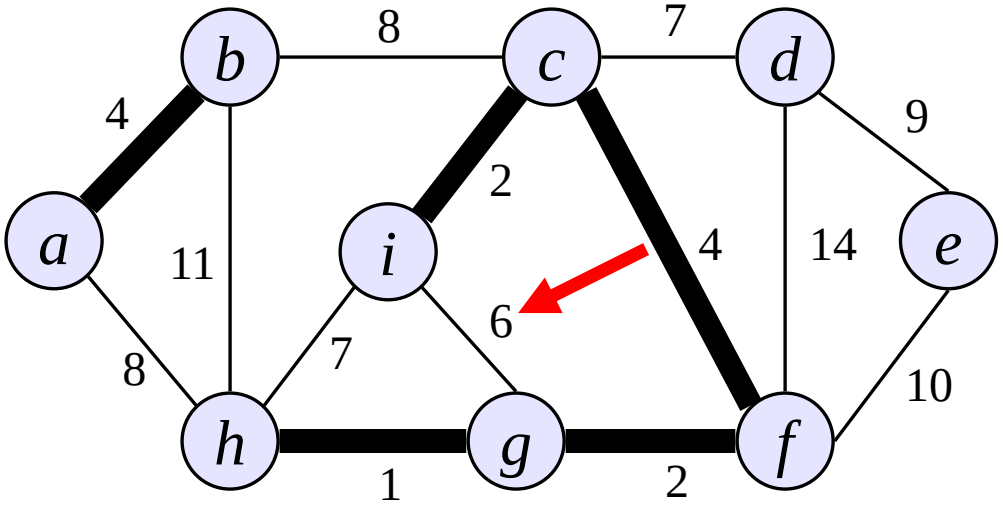
Initial sets = $\{\underline{a}\}, \{\underline{b}\}, \{c, i\}, \{d\}, \{e\}, \{f, g, h\}$

Final sets = $\{a, b\}, \{c, i\}, \{d\}, \{e\}, \{f, g, h\}$



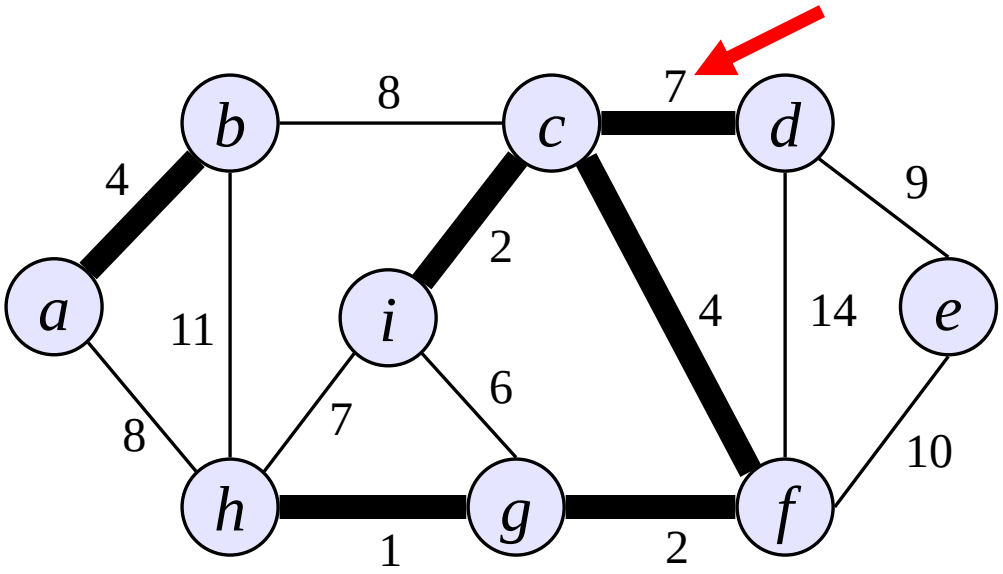
Initial sets = {*a*, *b*}, {*c*, *i*}, {*d*}, {*e*}, {*f*, *g*, *h*}

Final sets = {*a*, *b*}, {*c*, *f*, *g*, *h* , *i*}, {*d*}, {*e*}



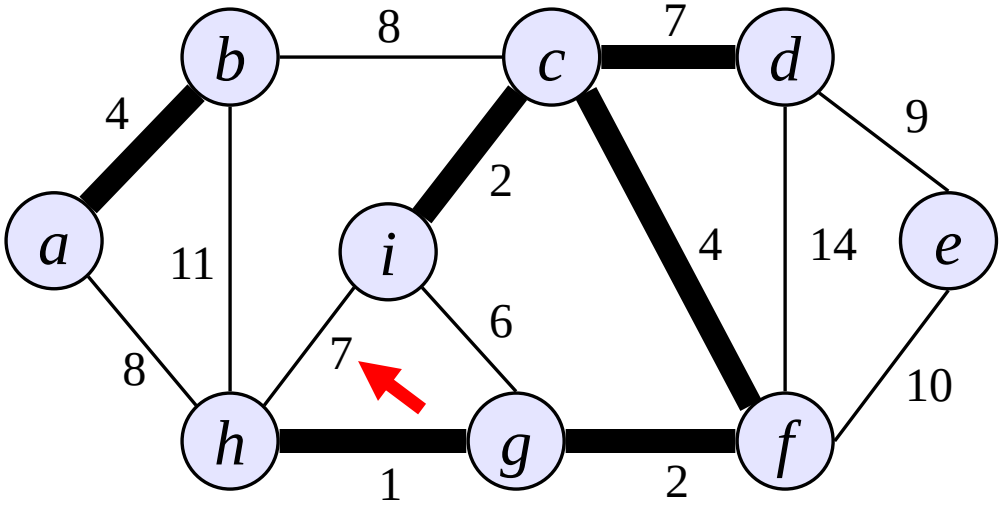
Initial sets = {*a*, *b*}, {*c*, *f*, *g*, *h*, *i*}, {*d*}, {*e*}

Final sets = {*a*, *b*}, {*c*, *f*, *g*, *h*, *i*}, {*d*}, {*e*}



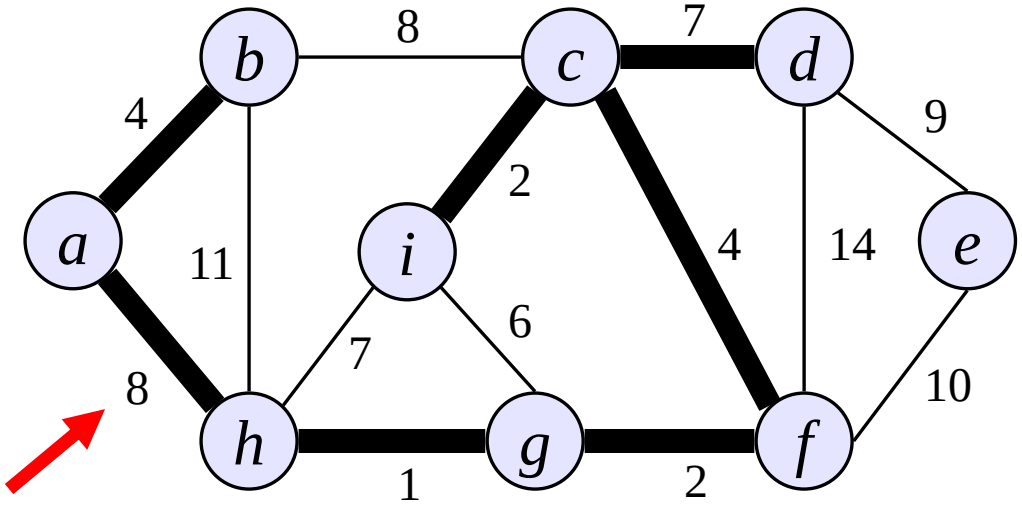
Initial sets = *{a, b}, {c, f, g, h, i}, {d}, {e}*

Final sets = *{a, b}, {c, d, f, g, h, i}, {e}*



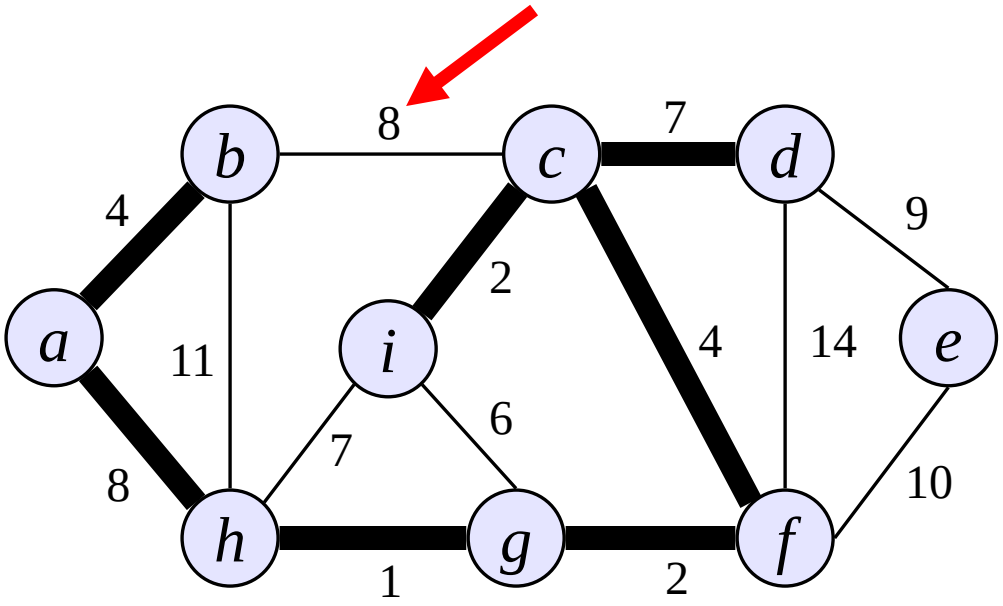
Initial sets = *{a, b}, {c, d, f, g, h , i}, {e}*

Final sets = *{a, b}, {c, f, d, g, h , i}, {e}*



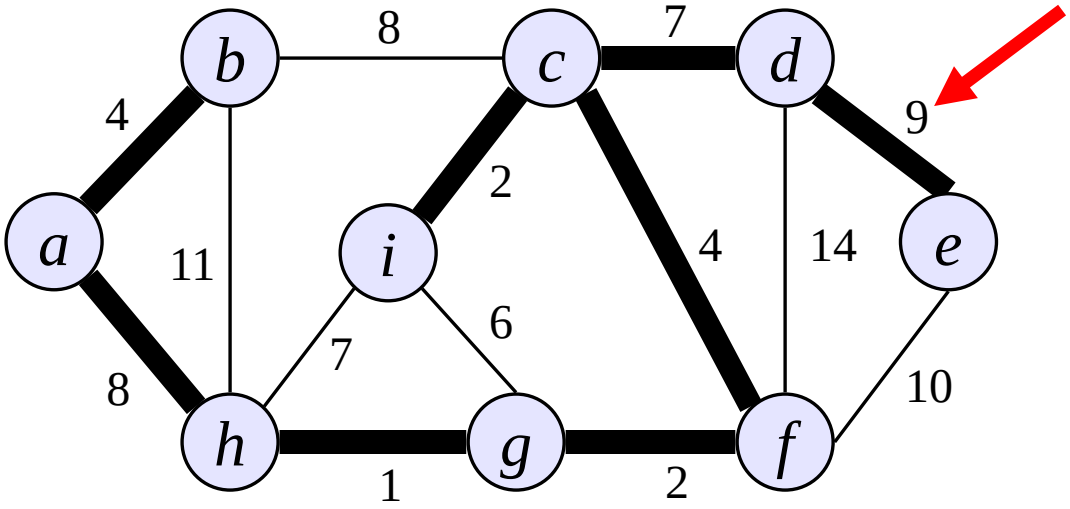
Initial sets = {a, b}, {c, d, f, g, h, i}, {e}

Final sets = {a, b, c, d, f, g, h , i}, {e}



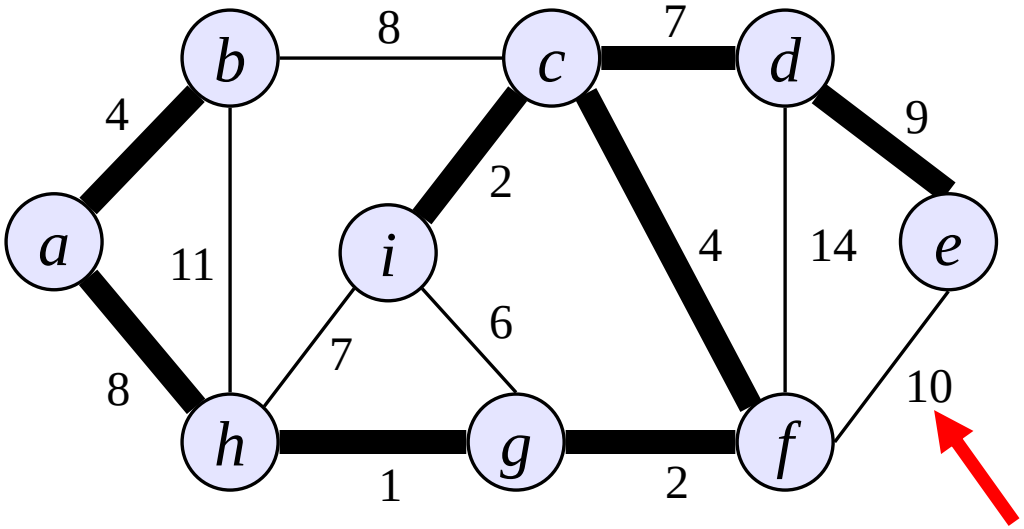
Initial sets = {*a*, *b*, *c*, *d*, *f*, *g*, *h* , *i*}, {*e*}

Final sets = {*a*, *b*, *c*, *d*, *f*, *g*, *h* , *i*}, {*e*}



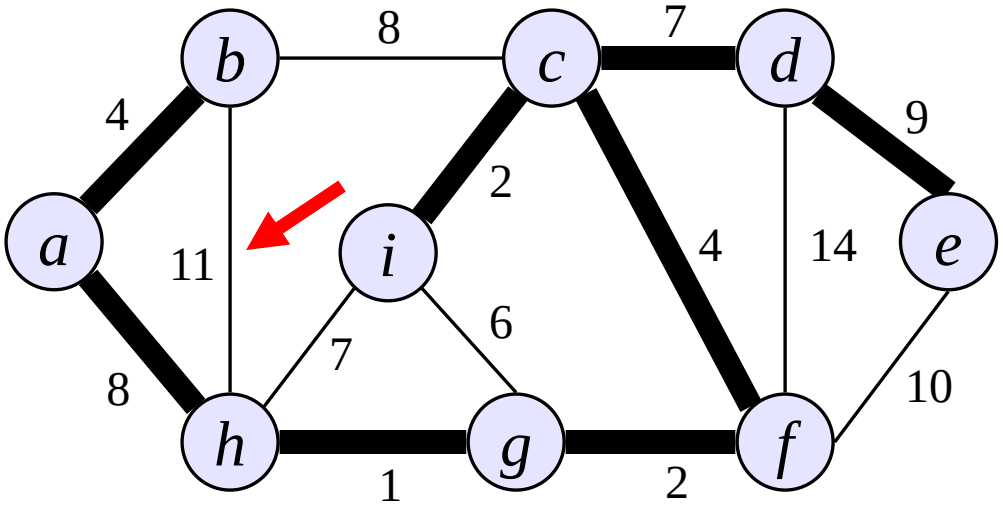
Initial sets = {*a*, *b*, *c*, *d*, *f*, *g*, *h* , *i*}, {*e*}

Final sets = {*a*, *b*, *c*, *d*, *e*, *f*, *g*, *h* , *i*}



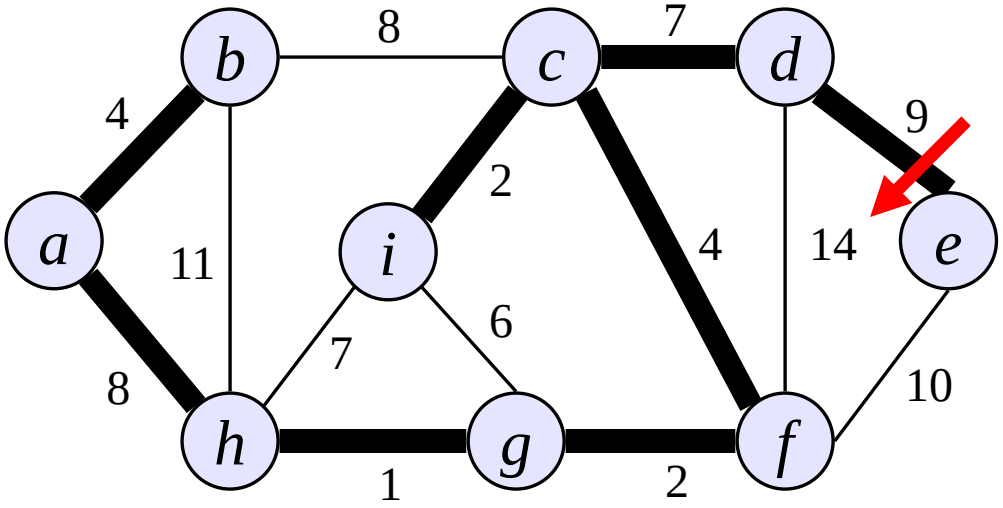
Initial sets = {*a*, *b*, *c*, *d*, *e*, *f*, *g*, *h* , *i*}

Final sets = {*a*, *b*, *c*, *d*, *e*, *f*, *g*, *h* , *i*}



Initial sets = {*a*, *b*, *c*, *d*, *e*, *f*, *g*, *h*, *i*}

Final sets = {*a*, *b*, *c*, *d*, *e*, *f*, *g*, *h*, *i*}



Initial sets = {*a*, *b*, *c*, *d*, *e*, *f*, *g*, *h*, *i*}

Final sets = {*a*, *b*, *c*, *d*, *e*, *f*, *g*, *h* , *i*}

MST - Greedy Algorithms

The previous demo follows what we call: Kruskal's algorithm.

- Start with T = ϕ. Consider edges in ascending order of cost.
- Insert edge e in T unless doing so would create a cycle.

- Like Prim's, this algorithm also produce an MST.

MST - Greedy Algorithms

Correctness?

Our "the cheapest edge" argument does not hold for every step.

Let's extend it carefully

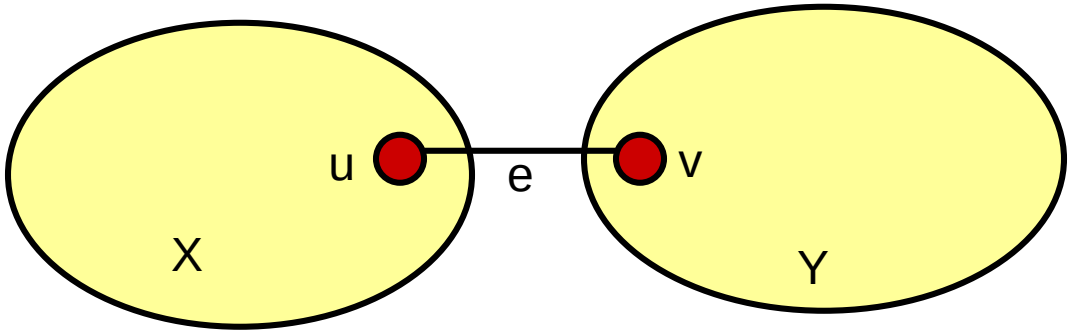
Simplification: All edge costs c_e are distinct.

A generalized version of cheapest edge inclusion property of MSTs:

If we partition the node set V arbitrarily in two non-empty sets X and Y, then the cheapest edge among all edges connecting X and Y must belong to any MST.

Edge inclusion lemma:

Let X = S and Y = V\S, and suppose e = (u, v) is the minimum cost edge of E, between X and Y.



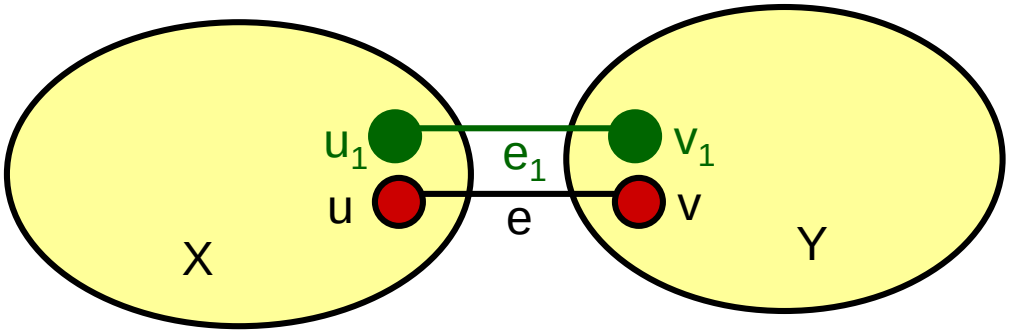
We want to show:

- e is in every minimum spanning tree of G
  - Or equivalently, if e is not in T, then T is not an MST.

Edge inclusion lemma

Pf. Suppose T is an MST that does not contain e

- Add e to T, this creates a cycle
- The cycle must have some edge $e_1 = (u_1, v_1)$ with u_1 in X and v_1 in Y



$T^* = T \cup \{e\} - \{e_1\}$ is a spanning tree with lower cost

- by assumption, e is the minimum cost edge between X and Y

Hence, T is not a minimum spanning tree (a contradiction).

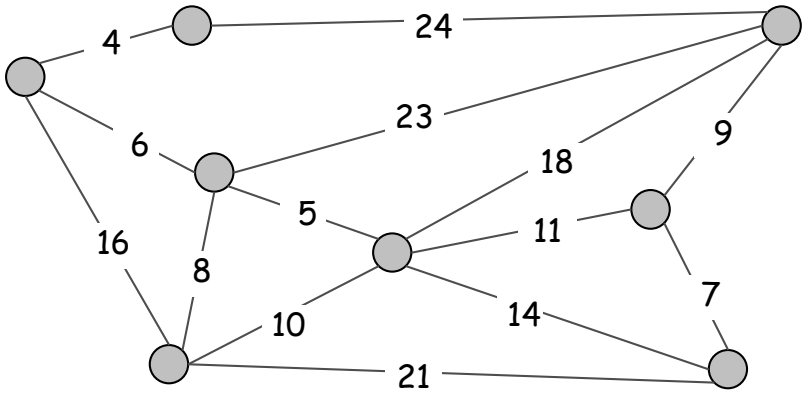
- In the presence of assumption (All edge costs c_e are distinct), above proof also says that: the MST is uniquely determined.

Optimality Proofs

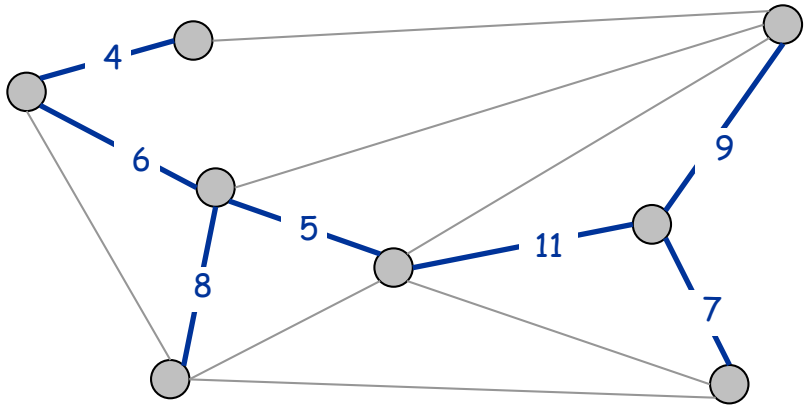
Prim's (Kruskal's) algorithm computes a Tree which is an MST

Show that:

- When an edge is added to the MST by Prim or Kruskal, the edge is the minimum cost edge between S and V\S for some set S.
- All selected edges together form a tree (NO Cycles).
- The tree spans the entire set V.



G = (V, E)



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

Optimality Proofs

Prim's Algorithm: Without loosing any detail, we can write it as follows.

```
S = { };    T = { };
while S != V
        choose the minimum cost edge
                e = (u,v), with u in S, and v in V\S
        add e to T
        add v to S
endwhile
```

This ensures all nodes are included in the tree

Ensures Cycle Free

Edge inclusion lemma ensures every e chosen here actually belongs to an MST

Optimality Proofs

Kruskal's Algorithm:

Let $C = \{\{v_1\}, \{v_2\}, \ldots, \{v_n\}\}$, $T = \{\ \}$

while $|C| > 1$ ← This ensures all nodes are included in the tree

Let e = (u, v) with u in C_i and v in C_j be the minimum cost edge joining distinct sets in C ← Ensures Cycle Free

Replace C_i and C_j by $C_i \cup C_j$

Add e to T

endwhile

Edge inclusion lemma ensures every e chosen here actually belongs to an MST

Yet another Greedy Algorithm

Reverse-Delete Algorithm:

Delete the most expensive edge that does not disconnect the graph.

Relies on the following lemma:

- The most expensive edge on a cycle is never in a minimum spanning tree
- Correctness is proved along the same lines.

Remark: For dense graphs it is slower than the others, as it has to delete most edges.

Dealing with the assumption of no equal cost edges

Perturbation Argument:

Force the edge costs to be distinct to break the ties

- Add small distinct quantities to the edges having equal costs

Apply Prim's /Kruskal's algorithm

- the resulting edge set is an MST also for G with the original edge costs
  - the sum of perturbations should be small enough

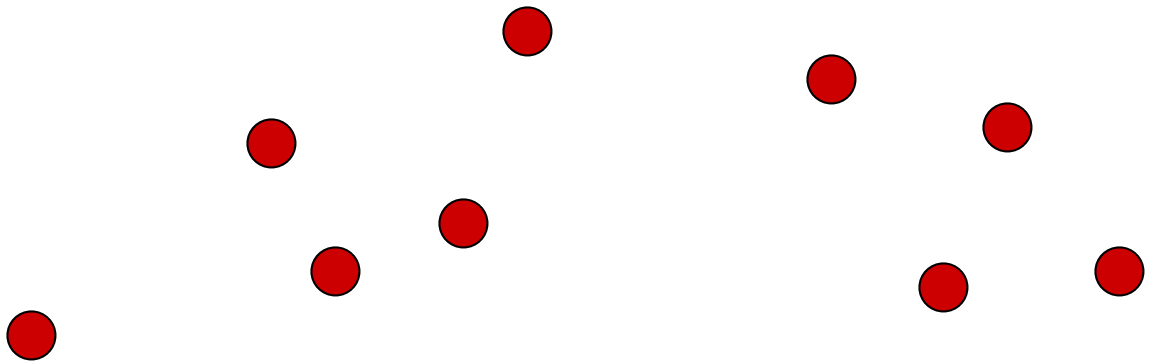
Application: Clustering

Clustering: a partitioning of a set of (data) points into disjoint subsets of points, called clusters.

How: Some distance function is defined between the points.

Spacing: the minimum distance of two clusters
(or equivalently, the minimum distance of any two points from different clusters).

Given : a collection of n points in a geometrical space, and an integer $k < n$.
The pairwise distances of points are known, or they can be easily computed.



Goal: Construct a clustering with k clusters and maximum spacing.

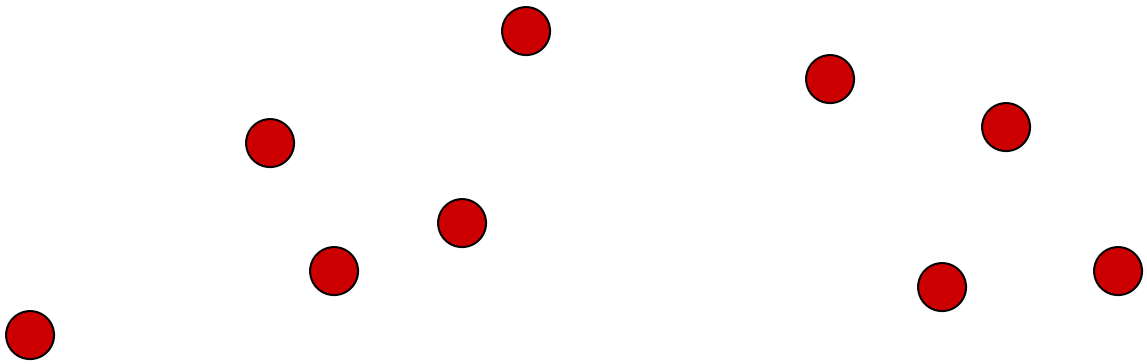
Distance clustering

Divide the data set into 2 subsets and maximum spacing

- dist (S_1, S_2) = min {dist(x, y) | x in S_1, y in S_2}

Motivations:

Clustering in general has many applications in data reduction, pattern recognition, classifcation, data mining, and related fields.



Distance Clustering Algorithm

Kruskal's Algorithm:

Let $C = \{\{v_1\}, \{v_2\}, \ldots, \{v_n\}\}$; $T = \{\ \}$

while $|C| > K$

Let $e = (u, v)$ with u in C_i and v in C_j

be the minimum cost edge joining distinct sets in C

Replace C_i and C_j by $C_i \cup C_j$

endwhile

The K nodes sets so obtained are the K clusters.

To prove: The obtained spacing is optimal.

The proof follows from the fact that Kruskal's algorithm always joins two disjoint subsets (into a single cluster) via a min cost edge having endpoints in both.

Implementation

Which one is faster Prim's Algorithm or Kruskal's Algorithms?

Prim's:

```
S = { };    T = { };
while S != V
        choose the minimum cost edge
                e = (u,v), with u in S, and v in V\S
        add e to T
        add v to S
endwhile
```

How we implement this step, is going to decide the running time.

Naïve implementation

- (n-1) iterations with O(m) every time to choose min cost e $\Rightarrow$ O (nm)

Implementation

Which one is faster Prim's Algorithm or Kruskal's Algorithms?

Prim's:

```
S = { };   T = { };
while S != V
        choose the minimum cost edge
                e = (u,v), with u in S, and v in V\S
        add e to T
        add v to S
endwhile
```

How we implement this step, is going to decide the running time.

Improving the Naïve:

- For each v in V\S, maintain attachment cost
  a[v] = cost of cheapest edge e = (u,v), with u in S
- Choosing the min cost edge is restricted to the n edges represented by a.
  When "add v to S" happens, update array a as:

```
foreach (edge e = (v, u) incident to v)
        if ((u ∉ S) and (c_e < a[u]))
                decrease cost of a[u] to c_e
```

$\Rightarrow O(n)$

Inside the while: $O(n + n) = O(n)$

Altogether: (n-1) iterations of while with $O(n)$ every time $\Rightarrow O(n^2)$

Implementation

Which one is faster Prim's Algorithm or Kruskal's Algorithms?

Prim's:

```
S = { };    T = { };
while S != V
        choose the minimum cost edge
                e = (u,v), with u in S, and v in V\S
        add e to T
        add v to S
endwhile
```

How we implement this step, is going to decide the running time.

Choosing a better data structure:

- Let F be the set of edges between S, and V\S.
  - Choose the minimum cost e from F
- when "add v to S" happens, we update F:
  - every e enters and leaves F exactly once $\Rightarrow O(m)$
- If F is a priority Queue, insert/remove(find min) is achievable using $O(\log m)$
- Altogether $\Rightarrow O(m \log m) \Rightarrow O(m \log n)$

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

Implementation

Which one is faster Prim's Algorithm or Kruskal's Algorithms?

Prim's:

```
S = { };    T = { };
while S != V
        choose the minimum cost edge
                e = (u,v), with u in S, and v in V\S
        add e to T
        add v to S
endwhile
```

Both implementations of Prim's algorithm are justified:

- $O(n^2)$ is somewhat faster if the graph is dense (has a quadratic number of edges),
- but otherwise, $O(m \log n)$ is considerably faster.

Implementation

Which one is faster Prim's Algorithm or Kruskal's Algorithms?

Kruskal's:

Let $C = \{\{v_1\}, \{v_2\}, \ldots, \{v_n\}\}$, $T = \{\}$
while $|C| > 1$
    Let $e = (u, v)$ with u in C_i and v in C_j be the minimum cost edge joining distinct sets in C
    Replace C_i and C_j by $C_i \cup C_j$
    Add e to T
endwhile

The running time depends on the implementation of these steps.

We need efficient way to find a "global" min cost edge:

- sort the edges wrt cost $\Rightarrow O(m \log m) \Rightarrow O(m \log n)$
- if current e creates a cycle, move to the next in the sorted list.

Avoiding cycles efficiently depends how we:

- Make sure for $e = (u, v)$, u and v belong to disjoint sets
- Maintain C after every "Replace C_i and C_j by $C_i \cup C_j$"
- A new data structure: Union-and-Find

Union-and-Find

- Maintains partitions of a set (here: V) into subsets
- Each subset has a label

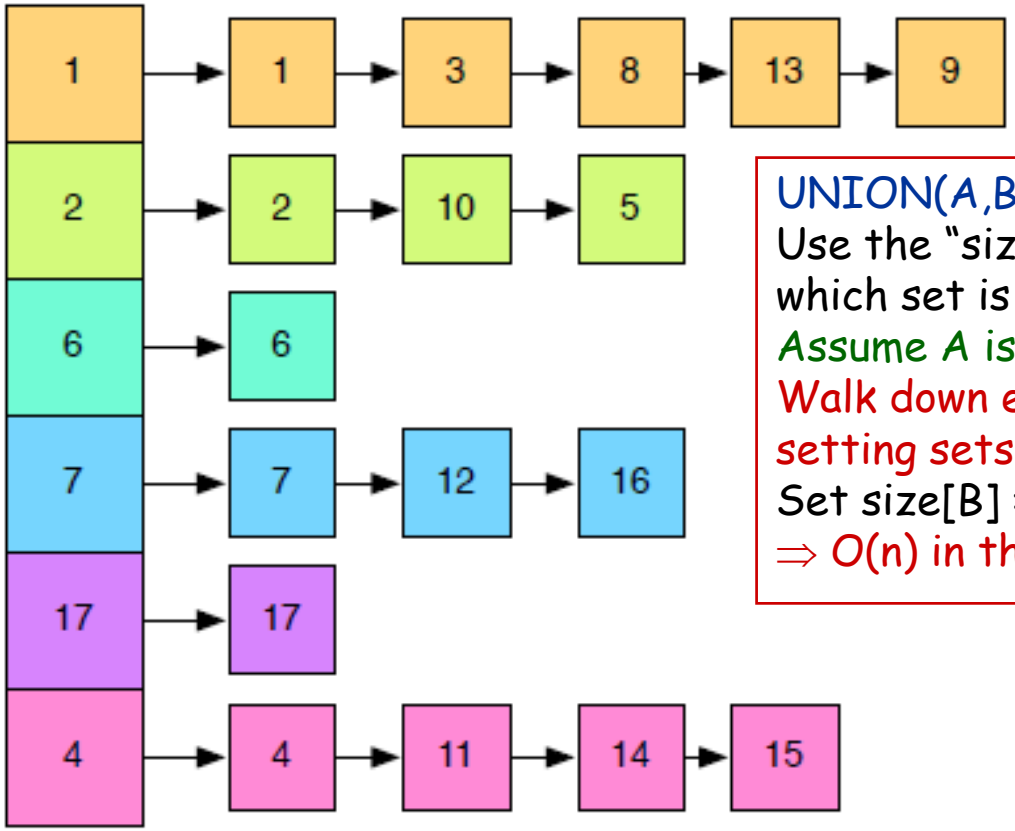
Supports the following operations:

FIND(i): returns the label of the subset that contains element i.

UNION(A,B): merge the subsets with labels A and B, that is,
replace these sets with A U B and give it a label.

Union-and-Find

UF Items:



UNION(A,B):
Use the "size" array to decide which set is smaller.
Assume A is smaller.
Walk down elements i in set A, setting sets[i] = B.
Set size[B] = size[B] + size[A]
⇒ O(n) in the worst case

UF Sizes:

1	5
2	3
6	1
7	3
17	1
4	4

FIND(i): return UF.sets[i].
- Takes a constant amount of time. ⇒ O(1)

UF Sets Array:

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
1	2	1	4	2	6	7	1	1	2	4	7	1	4	4	7	17

Reference: https://www.cs.cmu.edu/~ckingsf/class/02713-s13/lectures/lec03-othermst.pdf

Union-and-Find

FIND(i): $O(1)$

UNION(A,B): $O(n)$ in the worst case

Kruskal's:

Let $C = \{\{v_1\}, \{v_2\}, \ldots, \{v_n\}\}$, $T = \{\}$
while $|C| > 1$
    Let $e = (u, v)$ with u in C_i and v in C_j be the minimum cost edge joining distinct sets in C
    Replace C_i and C_j by $C_i \cup C_j$
    Add e to T
endwhile

2 FIND calls per e
$\Rightarrow O(m)$

(n-1) UNION calls
$\Rightarrow O(mn)$
using $O(n)$ per call

- Overall: $O(mn)$ dominates the sorting cost
- Why considering $O(n)$ which is worst case for ONE call to UNION?
- Can we do better by bounding the total for all UNION operations?
  - What is the time for $k < n$ call to UNION?

Union-and-Find

Rethinking "Time" for UNION(A,B):

Elementary operation:

- how often every element is relabeled and moved to other sets.

What invokes relabeling of an element?

- if it belongs to the smaller set in a UNION operation.

What happens then?

- the element now belongs to a new set of at least double the size.

What is the size of the larger set after k UNION operations?

- All sets are of size 1. After first UNION, the largest is 2
- After the second UNION, the largest can have at most 3 elements
- …, after k UNION operation, the largest can have at most k +1 size.

Every "elementary operation" moves the element from a set of size x to 2x

- every element is relabeled at most $\log_2(k+1)$ times.
- For all elements $\Rightarrow O(k \log k)$

Union-and-Find

- Sorting: $O(m \log m) \Rightarrow O(m \log n)$

  $m \le n^2 \Rightarrow \log m$ is $O(\log n)$

Kruskal's:

Let $C = \{\{v_1\}, \{v_2\}, \ldots, \{v_n\}\}$, $T = \{\ \}$
while $|C| > 1$
    Let $e = (u, v)$ with u in C_i and v in C_j be the minimum cost edge joining distinct sets in C
    Replace C_i and C_j by $C_i \cup C_j$
    Add e to T
endwhile

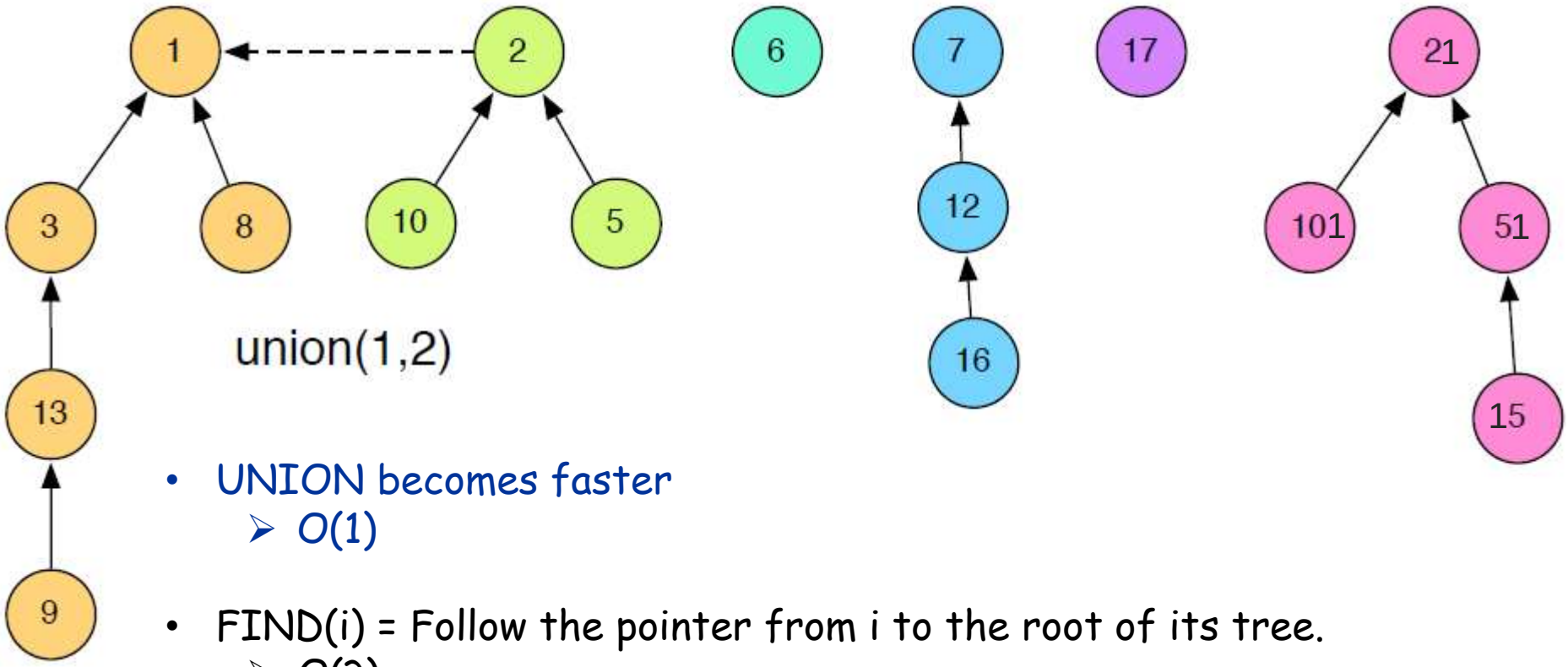
2 FIND calls per e
$\Rightarrow O(m)$

(n-1) UNION calls
$\Rightarrow O(n \log n)$

- Sorting dominates $\Rightarrow O(m \log n)$

Union-and-Find

- Another Implementation of Union-and-Find
  - Simpler and Faster in practice



- UNION becomes faster
  - O(1)
- FIND(i) = Follow the pointer from i to the root of its tree.
  - O(?)

Union-and-Find

FIND(i) requires $O(\log n)$ in a tree-based union-and-find data structure containing n items.

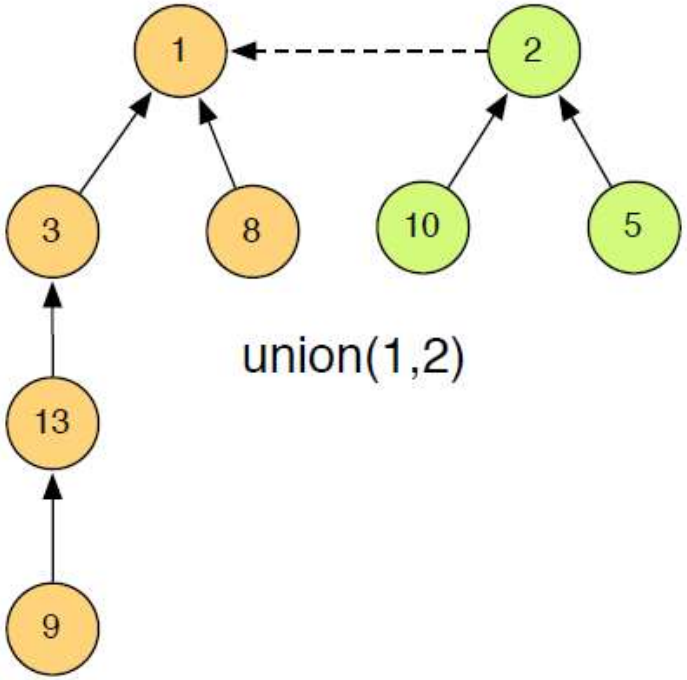
Pf.

What is the depth of an element v in its tree?

- the number of times the set containing it is renamed.

How many times a set can change its name?

- the set containing v at least doubles every time when its name is changed.



The largest number of times the size can double is $\log_2 n$. •

Kruskal's Algorithm:

Sorting $O(m \log n)$, FIND: $O(m \log n)$, UNION: $O(n)$

- Same running time as using the array-based union-and-find
- Simpler and Faster in practice