Background – Greedy Algorithms

- "Greedy" algorithms tend to be very simple.
- They make locally-optimized decisions
  i.e., they make the most advantageous choice at any given point, without regard for any possible past or future knowledge about the problem.
- As such, the most appealing choice at the moment may turn out, in the long run, to be a poor choice
- Greedy algorithms never back up and "change their mind" based on what they later learn

Greedy Algorithms - Example

- The Huffman Tree-building algorithm is an example of a greedy algorithm
  - It always picks the two remaining items with the lowest weights to select next for merging into a new subtree
  - In the case of Huffman, the greedy approach does happen to build the optimal encoding tree (as Huffman proved)
  - That's why Cormen presents Huffman not in the section on Binary Trees, but in Chapter 16, ("Greedy Algorithms")

Greedy Algorithms – Another Example

- Counting change using standard US coins, with the constraint of using as few coins as possible
  - Coins: half-dollar, quarter, dime, nickel, penny
  - Being greedy, and wanting to meet the constraint of using as few coins as possible, we'll always try to use the largest coin we can.

Greedy Algorithms – Another Example

- Example: Count out 36 cents in change

  Step 1: The largest coin is half-dollar; too big. Move to something smaller
  Step 2: Next-largest coin is quarter; use a quarter
  Step 3: We still have 11 cents to count out. The half-dollar and quarter are too big; use a dime
  Step 4: The half-dollar, quarter, dime, and nickel are too big; use a penny
  Total change = 36 cents (QUARTER, DIME, PENNY); we're done with 3 coins

  This happens to be the optimal solution.

Greedy Algorithms – Another Example

- Example: Count out 16 cents worth of change, but assume there's a 12-cent coin in addition to the standard denominations:
  - Step 1: The half-dollar and quarter are too big
  - Step 2: Next-largest coin is the 12-cent; USE ONE
  - Step 3: We still have 4 cents to count out. We wind up using FOUR pennies
  - Total change = 16 cents; we're done with 5 coins
  - This is NOT the optimal solution (dime, nickel, penny takes two fewer coins).
    - The decision to use the 12-cent coin looked good at the time, but backed us into a non-optimal solution, and our algorithm has no provision for backing up and picking some other combination

Yet Another Greedy Algorithm

- The Discrete (0-1) Knapsack problem:
  - A thief with a knapsack is breaking into a jewelry store
  - If the weight in the knapsack exceeds some threshold, the sack will break, and the thief gets nothing
  - The thief wants to maximize the value of his loot
  - If the jewelry store contains n items from which the thief can choose, there are 2n combinations to consider: the thief may grow old while planning the heist!
  - The thief can't take part of an item; it's all-or-nothing

The Discrete Knapsack Problem (2)

- One obvious strategy: Take the item with the largest value first
  - The problem: The item with the largest value might also have a correspondingly high weight
  - Suppose we have three items, valued at $10, $9, and $9, with weights of 25, 10, and 10 pounds (respectively). Further, suppose the knapsack's weight limit is 30 pounds
  - If we start with the most valuable item ($10 and 25 pounds), we can't take another item, so we wind up with $10 in loot.
  - The optimal solution would be to take the two $9 (10-pound) items, for a total of $18 (and 20 lbs).

The Discrete Knapsack Problem (3)

- Another obvious strategy: Take the item with the smallest weight first
  - The problem: If the items that are light in weight could have small values compared with their weights (the thief fills the bag with “cheap fluff”)

The Discrete Knapsack Problem (4)

- A more sophisticated strategy: Take the item with the highest value/weight ratio first.

  Suppose we have three items:

  - Item 1: $50 / 5 lbs = $10/lb
  - Item 2: $60 / 10 lbs = $6/lb
  - Item 3: $140 / 20 lbs = $7/lb

  If we take the items in order of value per weight, we would take items 1 & 3 ($50 & $140, with weights of 5 & 20 lbs), for a total of $190 / 25 lbs

  But the optimal solution (which maximizes the total value) is to take items 2 & 3 ($60 & $140, with weights of 10 & 20 lbs), for a total of $200 / 30 lbs.

The Discrete Knapsack Problem (5)

- The greedy approach does not work (i.e., it can give us a non-optimal solution) on the discrete knapsack problem!
  - We may get lucky, and have it happen to give us an optimal solution for some particular set of data, but in general, we can't count on it (in the general case) to be optimal.

The Fractional Knapsack Problem

- A variant of the discrete knapsack problem is the Fractional Knapsack Problem:
  - The thief can take part of an item.
  - Think of the Discrete problem as gold/silver bars
  - Think of the Fractional problem as gold/silver dust
  - By being able to take part of an item, there is never any wasted capacity in the knapsack

The Fractional Knapsack Problem

- Consider the original three items:
  Item 1: $50 / 5 lbs = $10/lb
  Item 2: $60 / 10 lbs = $6/lb
  Item 3: $140 / 20 lbs = $7/lb
- The thief would take all of item 1 ($50 / 5 lb), all of item 3 ($140 / 20 lb), and have 5 lbs of knapsack space left, with which he could take 5 lbs of item #2 ($30)
- Total haul: $50 + 140 + 30 = $220
- Total haul: 5 + 20 + 5 = 30 lbs
- This IS the optimal solution

The Fractional Knapsack Problem

- Although the greedy approach does not necessarily provide an optimal solution to the discrete knapsack problem, it does (always) provide an optimal solution to the fractional knapsack problem.
- Now, on to spanning trees…

Motivation

- Given: a number of towns in a remote area, where roads are very expensive (per mile) to build
- Wanted: a way of getting to/from all towns in the region (not necessarily directly)
  - Going from town A to town B by way of some other town X is perfectly acceptable – there just has to be SOME path from A to B.
- Constraint: Given the cost of road building, minimize the total length of all roads required
- If we realize the towns can represent vertices and the roads are edges, then this is a graph problem

Motivation – Another Example

- Given: an electric circuit, in which a number of pins need to be connected together (a common point on the schematic)
- Wanted: a way of getting all such pins connected (not necessarily directly to each other), so that they're electrically equivalent
- Constraint: Minimize the total length of all connecting wire required

Spanning Trees

- We can model these graph problems using spanning trees
- Given a connected, undirected graph G = (V, E), and a weight
  w(u, v) for each edge (u, v) ? E, …
- We want to find a subset of E, which we'll call T (i.e., T ? E), such that

  is minimized

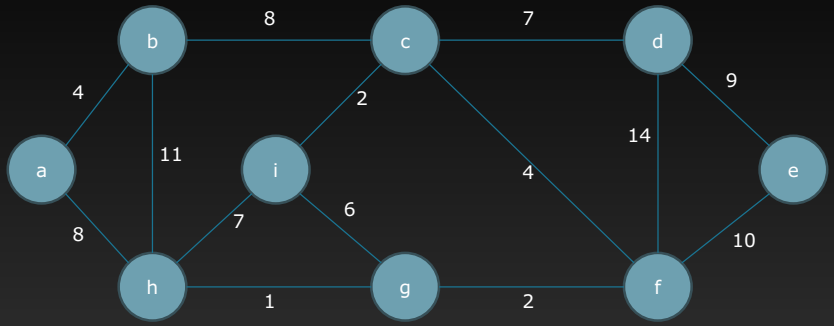
Spanning Trees

- Since T is acyclic, undirected, and it connects all of the vertices, it must form a (single) tree, which "spans" the graph, making it a spanning tree.
  - We can get a spanning tree directly from the DFS algorithm.
- The problem of finding the tree T that minimizes the total weights
  is called the minimum spanning tree problem.
  - Note that what we're minimizing is the total weight of all of the edges in the tree; not the size of the tree itself (the number of vertices or edges it contains). It's more precisely called the "minimum-weight spanning tree" problem.

Minimum Spanning Trees (MST)

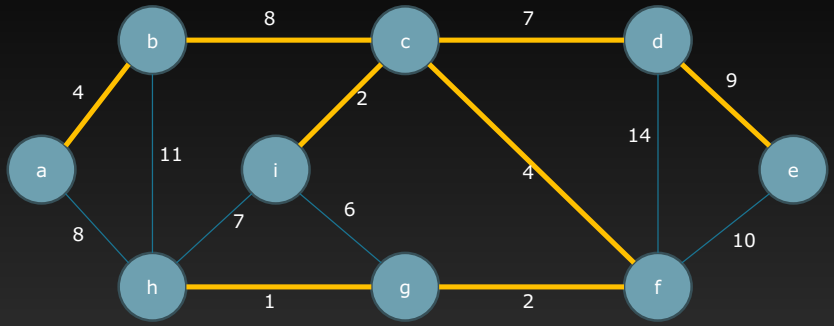
- The two algorithms we consider today, Prim's and Kruskal's, both find minimum-weight spanning trees by using a greedy algorithm.
- We've already seen that greedy algorithms are not always guaranteed to find the optimal solutions.
- For the minimum spanning tree problem, however, just like the fractional knapsack problem, they DO always find an optimal solution!

MST Example



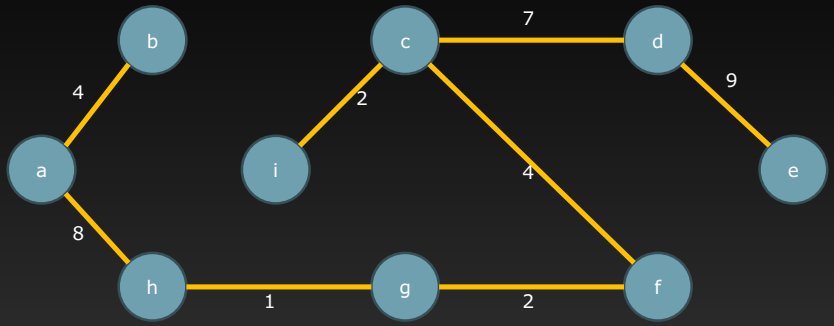
- Given this graph, with vertices a through i, and edges as shown, with labeled weights, find the (a) minimum-weight spanning tree.

MST Example



- You can get from any vertex to any other vertex using this tree. The sum of the weights is 37
- Once we've built our MST, we can eliminate all of the non-MST edges

MST Example



- If we drop (b, c) (whose weight is 8), replacing it with (a, h) (whose weight is also 8), we still have a spanning tree with a weight of 37. "The" MST may not be unique, but there IS a single minimum weight!
  - i.e., this graph has no spanning tree with a weight < 37

MST Properties

- Some properties of an MST:
  - It has |V| – 1 edges.
  - It has no cycles.
  - It might not be unique for a given graph
    - i.e., more than one tree with the minimum weight may exist

How To Grow an MST?

- Building up the solution:
  - We will build a set A of edges.
  - Initially, A has no edges.
  - As we add edges to A, maintain a loop invariant:
    - A is a subset of some MST.
  - Add only edges that maintain the invariant. If A is a subset of some MST, an edge (u, v) is safe for addition to A if and only if A $\cup$ {(u, v)} is also a subset of some MST.
  - So we will add only safe edges.
  - Determining what constitutes a "safe edge" is coming up.

The Generic MST Algorithm

```
Generic-MST(G, w)
1  A = ∅
2  while A does not form a spanning tree
3          find and edge (u, v) that is safe for A
4          A = A ∪ {(u, v)}
5  return A
```

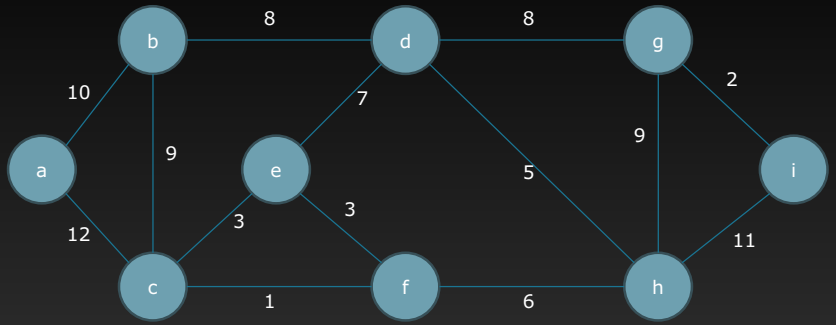
Loop Invariant and Safe Edges

- We can use the loop invariant (A is a subset of some MST) to show that this generic algorithm works.
- Initialization: The empty set trivially satisfies the loop invariant.
- Maintenance: Since we add only safe edges, A remains a subset of some MST.
- Termination: All edges added to A are in an MST, so when we stop, A is a spanning tree that is also an MST.

Loop Invariant and Safe Edges

- Obviously, the key to making this work lies in the details of "find an edge…that is safe for A"
- HOW?
- An example:

Loop Invariant and Safe Edges

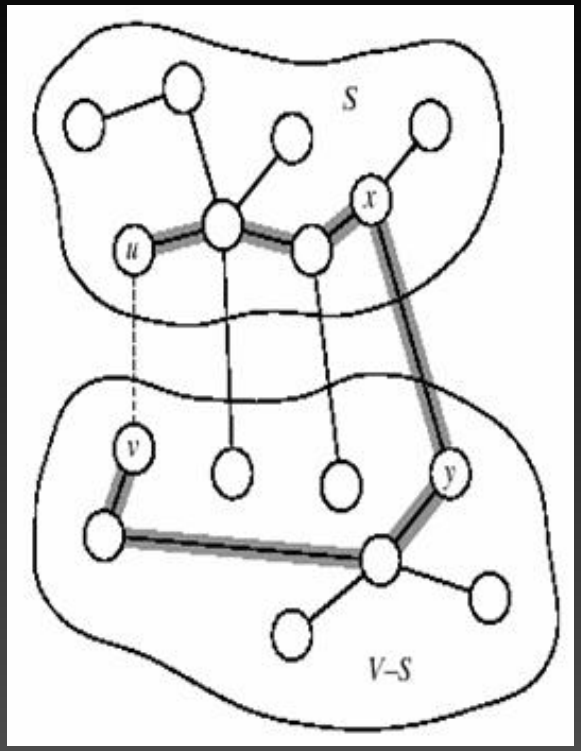


- Edge (c, f) has the lowest weight of any edge in the graph.
- Is this safe for A = [?]?

Safe Edges

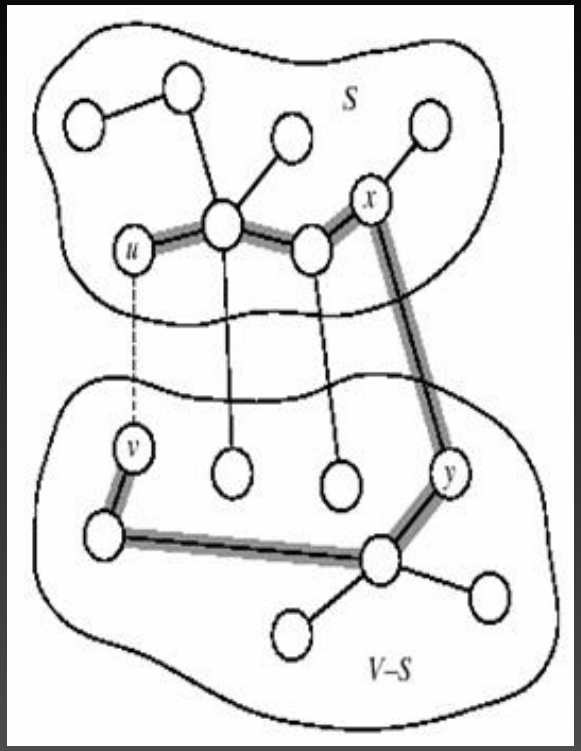
- Intuitively: Let $S \subseteq V$ be any set of vertices that includes c but not f (so that f is in $V - S$).
- In any MST, there has to be at least one edge that connects S with $V - S$.
- Why not choose the edge with minimum weight (which would be
  (c, f) in this case)?
- Some definitions:

Safe Edges – Definitions (1)



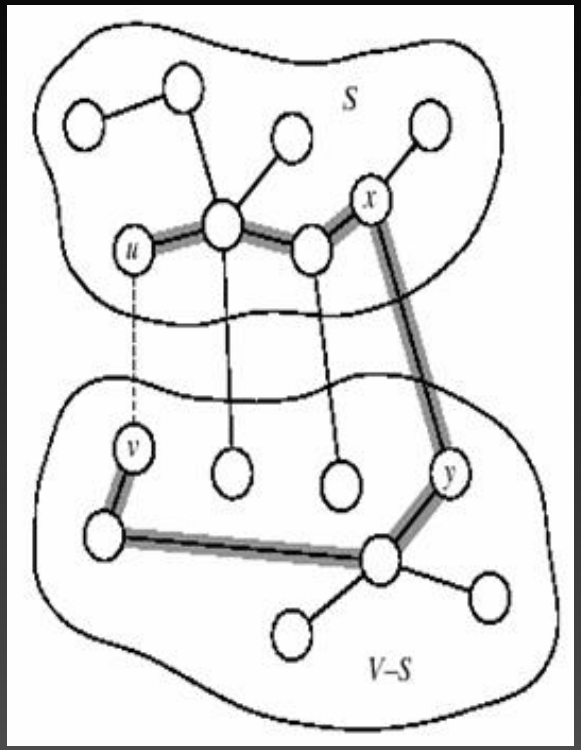
- Let S ⊆ V and A ⊆ E
- A cut (S, V – S) is a partition of vertices into disjoint sets V and V – S.
- Edge (u, v) ∈ E crosses the cut (S, V – S) if one endpoint is in S and the other is in V – S.
- A cut respects some set of edges A if and only if no edge in A crosses the cut.

Safe Edges – Definitions (2)



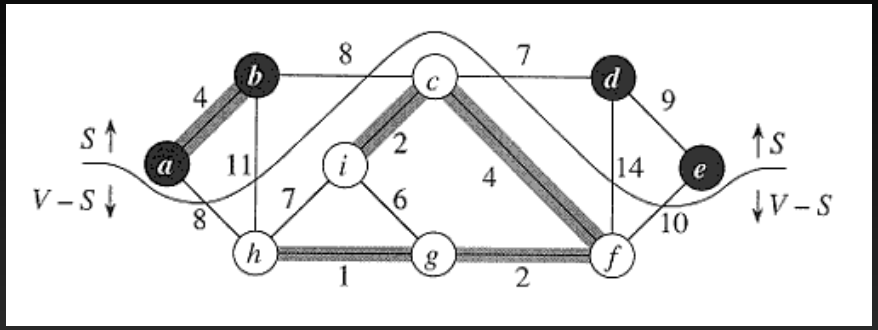
- An edge is a light edge crossing a cut if and only if its weight is minimum over all edges crossing the cut. For a given cut, there can be more than one light edge crossing it.
- If A is shown as shaded edges, the cut does not respect A; it cuts the edge
  (x, y)

Safe Edges – Definitions (3)



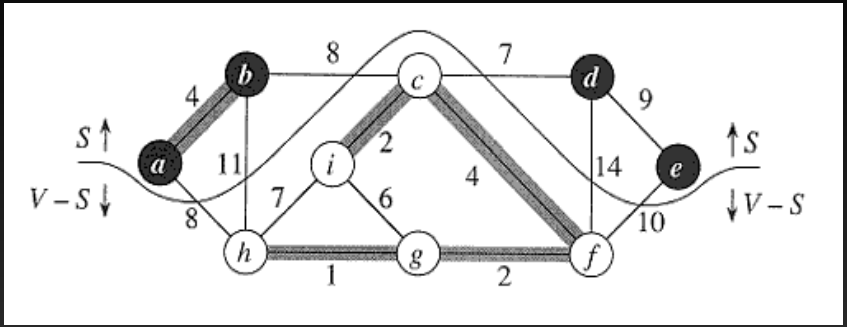
- If the weights of (u, v) & (x, y) are both 1, and the weights of the other two edges crossing the cut are 2, then (u, v) and (x, y) are both light edges.
- If w(u, v) = 2, w(x, y) = 1, and the weight of the other two edges that cross the cut are at least 2, then (x, y) is the only light edge for this cut

Safe Edges – Definitions (4)

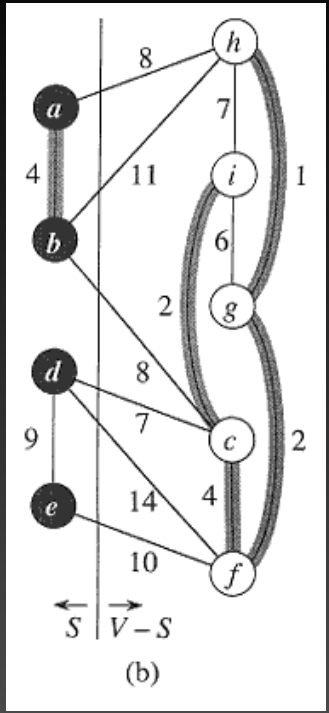


- Vertices in S are shown in black, vertices in V – S are shown in white.
- Edges crossing the cut connect a black vertex to a white one.
- (d, c) is the (unique) light edge crossing the cut
- A subset A of the edges is shaded.
- Note that the cut (S, V – S) respects A, as no edge of A crosses the cut

Safe Edges – Definitions (5)



- Another way of viewing this cut.
- An edge crosses the cut if it connects a vertex on the left (in S) with a vertex on the right (in V – S)



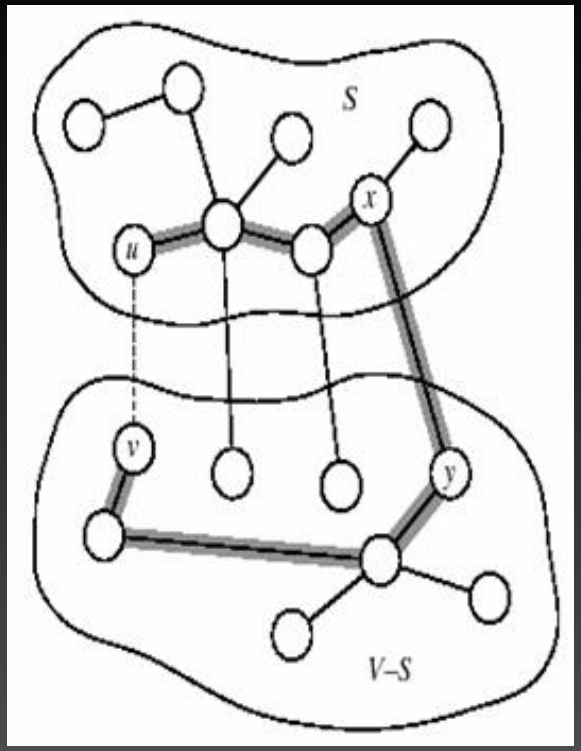
Theorem 23.1

- Let A be a subset of some MST, (S, V – S) be a cut that respects A, and (u, v) be a light edge crossing (S, V – S). Then (u, v) is safe for A.
- Proof
  - Let T be an MST that includes A.
  - If T contains (u, v), done.
  - Assume T does not contain (u, v). We'll construct a different MST T that includes A ∪ {(u, v)}

Theorem 23.1 (2)

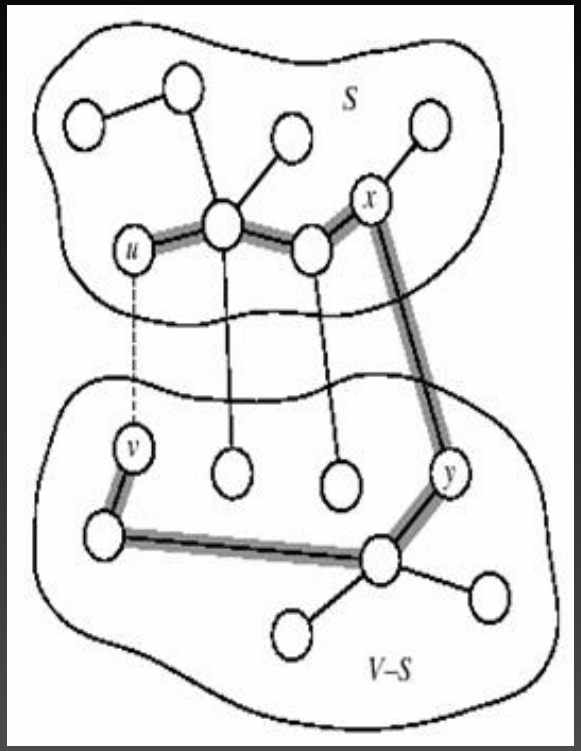
Recall: a tree has unique path between each pair of vertices.
Since T is an MST, it contains a unique path p between u and v.
Path p must cross the cut (S, V – S) at least once. Let (x, y) be an edge of p that crosses the cut. From how we chose (u, v), we must have w(u, v) ? w(x, y).
Since the cut respects A, edge (x, y) is not in A in the graph.

Theorem 23.1 (3)



- To form T' from T :
- Remove (x, y), which breaks T into two components.
- Add (u, v), reconnecting the trees.
- So T' = T - {(x, y)} ∪ {(u, v)}, and T' is a spanning tree.

Theorem 23.1 (4)



- w(T') = w(T) - w(x, y) + w(u, v) ? w(T) , since w(u, v) ? w(x, y).
- Since T' is a spanning tree, w(T') ? w(T), and T is an MST, then T' must be an MST.
- Need to show that (u, v) is safe for A:
- A ⊆ T and (x, y) not in A; thus, A ⊆ T , and A ∪ {(u, v)} ⊆ T'
- Since T is an MST, (u, v) is safe for A

Kruskal's & Prim's Algorithms

- Both are elaborations of Generic-MST.
- They differ in how they "select a safe edge" in line 3 of Generic-MST.
- In Kruskal's algorithm, the set A is a forest (multiple trees). The safe edge we add always connects two distinct trees
- In Prim's algorithm, the set A is a single tree. The safe edge we add is always a least-weight edge connecting the tree to a vertex not yet in the tree.

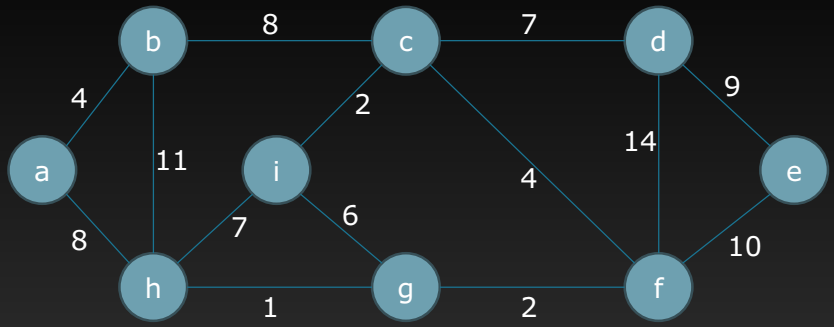
Kruskal's Algorithm

- Kruskal's algorithm depends on the existence of few set-based functions:
- Make-Set(x) creates a set containing the single item x.
- Find-Set(x) looks through the sets it is maintaining, and determines which set x belongs to
- Union(u, v) merges two sets (one containing u and the other containing v) into one set (u)

MST-Kruskal Algorithm

```
MST-Kruskal(G, w)
1  A = ?
2  for each vertex v ? G.V
3        Make-Set(v)
4  sort the edges of E by increasing w (weight)
   value
5  for each edge (u, v) ? E (taken in weight order)
6        if Find-Set(u) ? Find-Set(v)
7              A = A ? {(u, v)}
8              Union(u, v)
9  return A
```

Kruskal's Algorithm

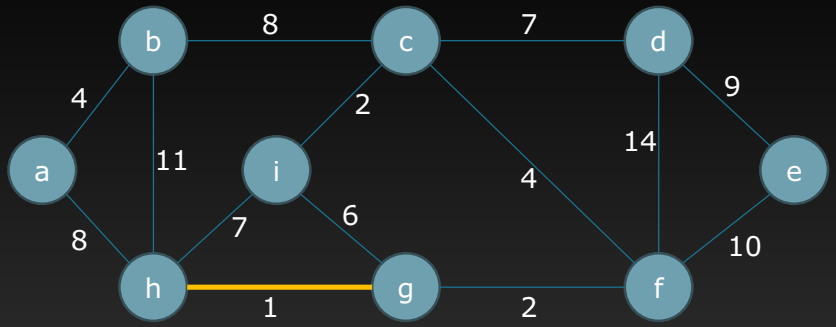


Sets:
{a}
{b}
{c}
{d}
{e}
{f}
{g}
{h}
{i}

Vertices:
{g, h}: 1
{c, i}: 2
{f, g}: 2
{a, b}: 4
{c, f}: 4
{g, i}: 6
{h, i}: 7
{c, d}: 7
{a, h}: 8
{b, c}: 8
{d, e}: 9
{e, f}: 10
{b, h}: 11
{d, f}: 14

- Start with every vertex in its own set
- Consider edges in increasing-weight order ONLY if the edge connects vertices in two different sets.
- If we include the edge, merge the two sets

Kruskal's Algorithm



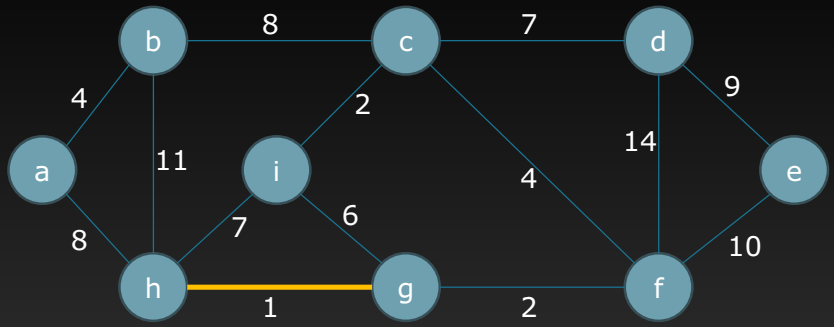
The lowest-weight edge is (g, h), and g and h are in different sets, so include the edge and merge the two sets

Sets:
{a}
{b}
{c}
{d}
{e}
{f}
{g}
{h}
{i}

Vertices:
{g, h}: 1
{c, i}: 2
{f, g}: 2
{a, b}: 4
{c, f}: 4
{g, i}: 6
{h, i}: 7
{c, d}: 7
{a, h}: 8
{b, c}: 8
{d, e}: 9
{e, f}: 10
{b, h}: 11
{d, f}: 14

- Start with every vertex in its own set
- Consider edges in increasing-weight order ONLY if the edge connects vertices in two different sets.
- If we include the edge, merge the two sets

Kruskal's Algorithm



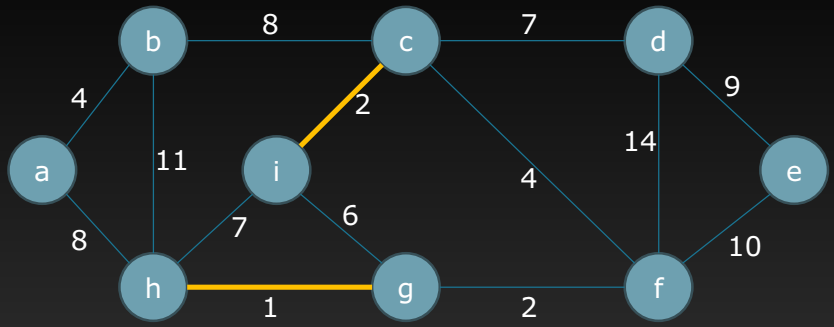
Sets:
{a}
{b}
{c}
{d}
{e}
{f}
{g, h}
{i}

Vertices:
{g, h}: 1
{c, i}: 2
{f, g}: 2
{a, b}: 4
{c, f}: 4
{g, i}: 6
{h, i}: 7
{c, d}: 7
{a, h}: 8
{b, c}: 8
{d, e}: 9
{e, f}: 10
{b, h}: 11
{d, f}: 14

The lowest-weight edges are (c, i) and (f, g). Vertices c and i are in different sets, so include the edges and merge the sets

- Start with every vertex in its own set
- Consider edges in increasing-weight order ONLY if the edge connects vertices in two different sets.
- If we include the edge, merge the two sets

Kruskal's Algorithm



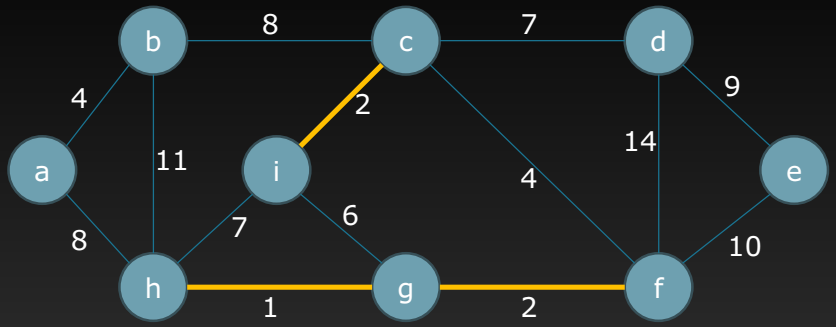
Sets:
{a}
{b}
{c, i}
{d}
{e}
{f}
{g, h}

Vertices:
{g, h}: 1
{c, i}: 2
{f, g}: 2
{a, b}: 4
{c, f}: 4
{g, i}: 6
{h, i}: 7
{c, d}: 7
{a, h}: 8
{b, c}: 8
{d, e}: 9
{e, f}: 10
{b, h}: 11
{d, f}: 14

The lowest-weight remaining edge is (f, g). Vertices f and g are in different sets, so include the edge and merge the sets

- Start with every vertex in its own set
- Consider edges in increasing-weight order ONLY if the edge connects vertices in two different sets.
- If we include the edge, merge the two sets

Kruskal's Algorithm



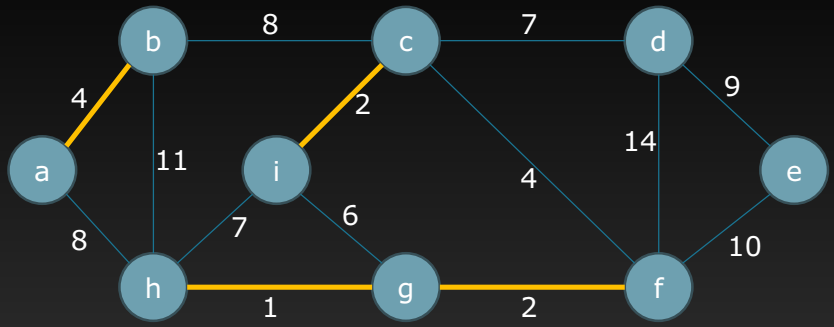
Sets:
{a}
{b}
{c, i}
{d}
{e}
{f, g, h}

Vertices:
{g, h}: 1
{c, i}: 2
{f, g}: 2
{a, b}: 4
{c, f}: 4
{g, i}: 6
{h, i}: 7
{c, d}: 7
{a, h}: 8
{b, c}: 8
{d, e}: 9
{e, f}: 10
{b, h}: 11
{d, f}: 14

The lowest-weight remaining edges are (a, b) and (c, f) Vertices a and b are in different sets, so include the edge and merge the sets

- Start with every vertex in its own set
- Consider edges in increasing-weight order ONLY if the edge connects vertices in two different sets.
- If we include the edge, merge the two sets

Kruskal's Algorithm



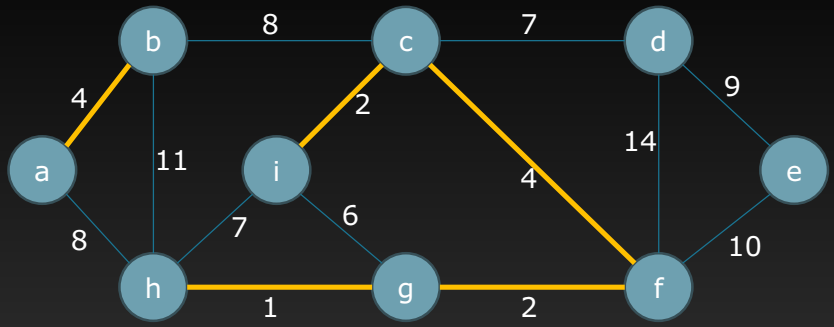
Sets:
{a, b}
{c, i}
{d}
{e}
{f, g, h}

Vertices:
{g, h}: 1
{c, i}: 2
{f, g}: 2
{a, b}: 4
{c, f}: 4
{g, i}: 6
{h, i}: 7
{c, d}: 7
{a, h}: 8
{b, c}: 8
{d, e}: 9
{e, f}: 10
{b, h}: 11
{d, f}: 14

The lowest-weight remaining edge is (c, f) Vertices c and f are in different sets, so include the edge and merge the sets

- Start with every vertex in its own set
- Consider edges in increasing-weight order ONLY if the edge connects vertices in two different sets.
- If we include the edge, merge the two sets

Kruskal's Algorithm



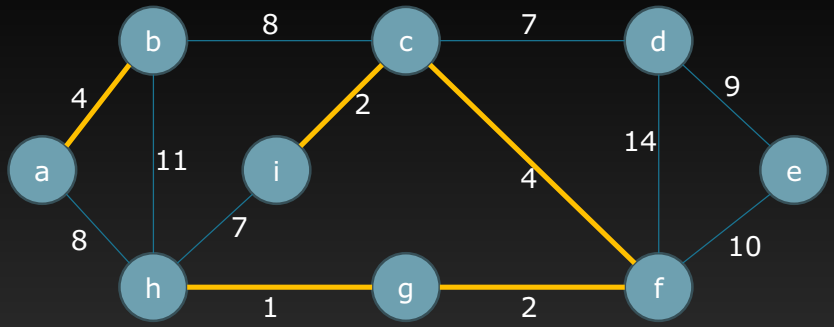
Sets:
{a, b}
{c, f, g, h, i}
{d}
{e}

Vertices:
{g, h}: 1
{c, i}: 2
{f, g}: 2
{a, b}: 4
{c, f}: 4
{g, i}: 6
{h, i}: 7
{c, d}: 7
{a, h}: 8
{b, c}: 8
{d, e}: 9
{e, f}: 10
{b, h}: 11
{d, f}: 14

The lowest-weight remaining edge is (g, i) Vertices g and i are in the same set, so we drop this edge from consideration, and keep going

- Start with every vertex in its own set
- Consider edges in increasing-weight order ONLY if the edge connects vertices in two different sets.
- If we include the edge, merge the two sets

Kruskal's Algorithm



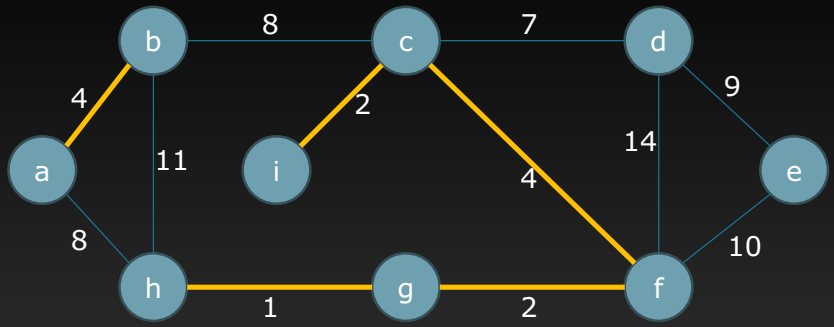
Sets:
{a, b}
{c, f, g, h, i}
{d}
{e}

Vertices:
{g, h}: 1
{c, i}: 2
{f, g}: 2
{a, b}: 4
{c, f}: 4
~~{g, i}: 6~~
{h, i}: 7
{c, d}: 7
{a, h}: 8
{b, c}: 8
{d, e}: 9
{e, f}: 10
{b, h}: 11
{d, f}: 14

The lowest-weight remaining edges are (h, i) and (c, d). Vertices h and i are in the same set, so we drop this edge from consideration, and keep going

- Start with every vertex in its own set
- Consider edges in increasing-weight order ONLY if the edge connects vertices in two different sets.
- If we include the edge, merge the two sets

Kruskal's Algorithm



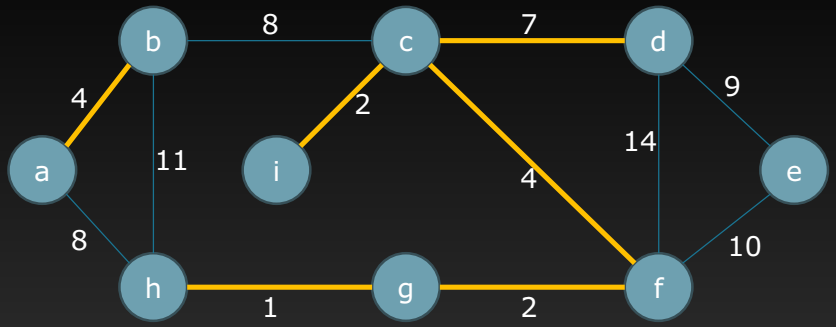
Sets:
{a, b}
{c, f, g, h, i}
{d}
{e}

Vertices:
{g, h}:
1
{c, i}: 2
{f, g}:
2
{a, b}:
4
{c, f}:
4
~~{g, i}:~~
6
~~{h, i}:~~
7
{c, d}:
7
{a, h}:
8
{b, c}:
8
{d, e}:
9
{e, f}:
10
{b, h}:
11
{d, f}:
14

The lowest-weight remaining edge is (c, d). Vertices c and d are in different sets, so include this edge and merge the sets

- Start with every vertex in its own set
- Consider edges in increasing-weight order ONLY if the edge connects vertices in two different sets.
- If we include the edge, merge the two sets

Kruskal's Algorithm



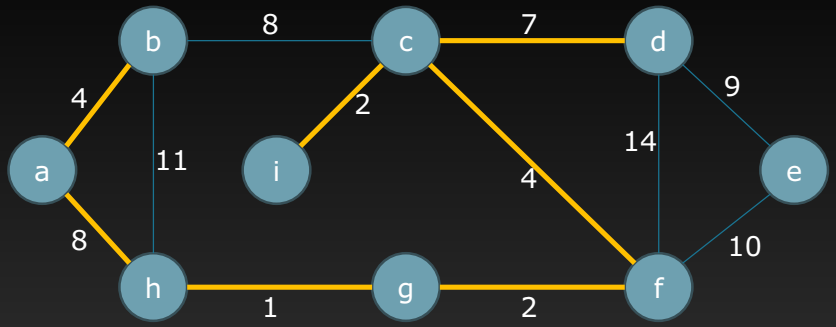
Sets:
{a, b}
{c, d, f, g, h, i}
{e}

Vertices:
{g, h}:
1
{c, i}: 2
{f, g}:
2
{a, b}:
4
{c, f}:
4
~~{g, i}:~~
~~6~~
~~{h, i}:~~
~~7~~
{c, d}:
7
{a, h}:
8
{b, c}:
8
{d, e}:
9
{e, f}:
10
{b, h}:
11
{d, f}:
14

The lowest-weight remaining edges are (a, h) and (b, c). Vertices a and h are in different sets, so include this edge and merge the sets

- Start with every vertex in its own set
- Consider edges in increasing-weight order ONLY if the edge connects vertices in two different sets.
- If we include the edge, merge the two sets

Kruskal's Algorithm



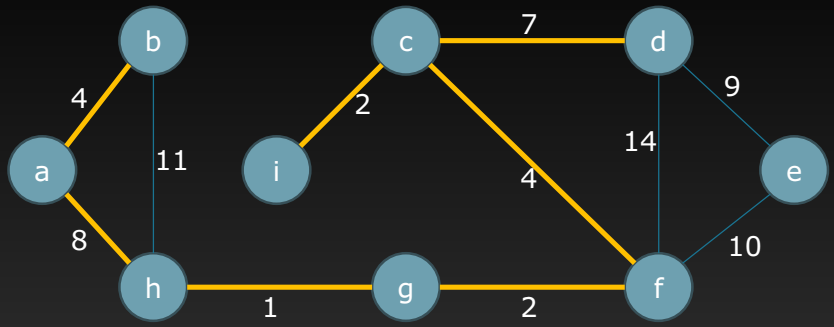
Sets:
{a, b, c, d, f, g, h, i}
{e}

Vertices:
{g, h}: 1
{c, i}: 2
{f, g}: 2
{a, b}: 4
{c, f}: 4
~~{g, i}: 6~~
~~{h, i}: 7~~
{c, d}: 7
{a, h}: 8
{b, c}: 8
{d, e}: 9
{e, f}: 10
{b, h}: 11
{d, f}: 14

The lowest-weight remaining edge is (b, c). Vertices b and c are in the same set, so drop this edge from consideration and keep going

- Start with every vertex in its own set
- Consider edges in increasing-weight order ONLY if the edge connects vertices in two different sets.
- If we include the edge, merge the two sets

Kruskal's Algorithm



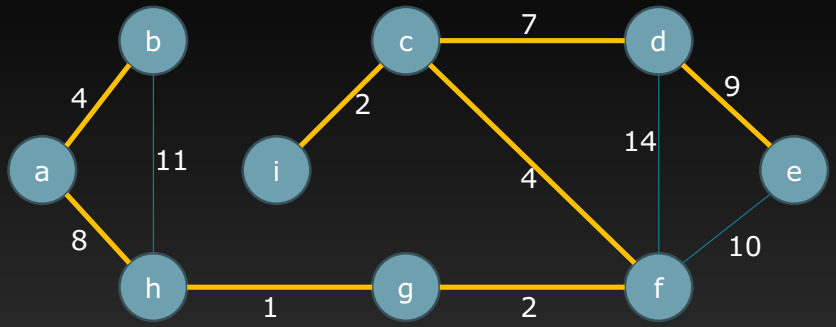
Sets:
{a, b, c, d, f, g, h, i}
{e}

Vertices:
{g, h}: 1
{c, i}: 2
{f, g}: 2
{a, b}: 4
{c, f}: 4
~~{g, i}: 6~~
~~{h, i}: 7~~
{c, d}: 7
{a, h}: 8
~~{b, c}: 8~~
{d, e}: 9
{e, f}: 10
{b, h}: 11
{d, f}: 14

The lowest-weight remaining edge is (d, e). Vertices d and e are in different sets, so include this edge and merge the two sets

- Start with every vertex in its own set
- Consider edges in increasing-weight order ONLY if the edge connects vertices in two different sets.
- If we include the edge, merge the two sets

Kruskal's Algorithm



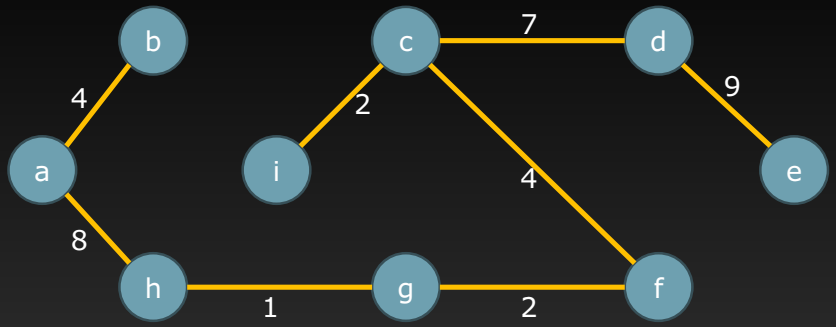
Sets:
{a, b, c, d, e, f, g, h, i}

Vertices:
{g, h}: 1
{c, i}: 2
{f, g}: 2
{a, b}: 4
{c, f}: 4
~~{g, i}: 6~~
~~{h, i}: 7~~
{c, d}: 7
{a, h}: 8
~~{b, c}: 8~~
{d, e}: 9
{e, f}: 10
{b, h}: 11
{d, f}: 14

We will next consider the edges (e, f), (b, h), and (d, f) (in that order). Because they are all already in the same set, we will drop them all.

- Start with every vertex in its own set
- Consider edges in increasing-weight order ONLY if the edge connects vertices in two different sets.
- If we include the edge, merge the two sets

Kruskal's Algorithm



Sets:
{a, b, c, d, e, f, g, h, i}

Vertices:
{g, h}: 1
{c, i}: 2
{f, g}: 2
{a, b}: 4
{c, f}: 4
~~{g, i}: 6~~
~~{h, i}: 7~~
{c, d}: 7
{a, h}: 8
~~{b, c}: 8~~
{d, e}: 9
~~{e, f}: 10~~
~~{b, h}: 11~~
~~{d, f}: 14~~

All nodes are connected. Total MST weight is 4 + 8 + 1 + 2 + 4 + 2 + 7 + 9 = 37

- Start with every vertex in its own set
- Consider edges in increasing-weight order ONLY if the edge connects vertices in two different sets.
- If we include the edge, merge the two sets

MST-Kruskal Algorithm - Analysis

```
MST-Kruskal(G, w)
1  A = ?
2  for each vertex v ? G.V
3        Make-Set(v)
4  sort the edges of G.E by increasing w (weight) value
5  for each edge (u, v) ? G.E (taken in weight order)
6       if Find-Set(u) ? Find-Set(v)
7            A = A ? {(u, v)}
8            Union(u, v)
9  return A
```

Total time: O(E lg V) (see p. 633)

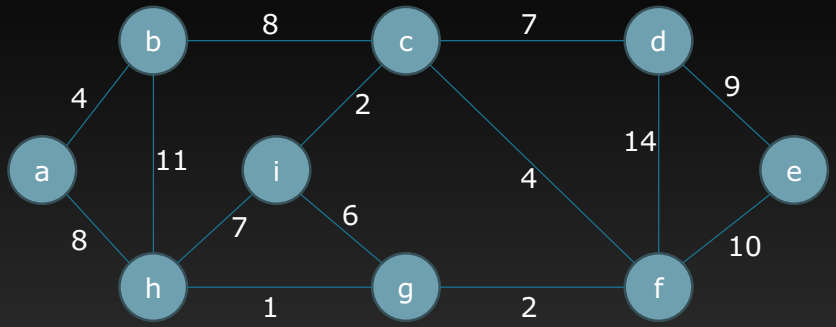
Prim's Algorithm

- In Prim's algorithm, the set A is one tree. The safe edge we add is always a least-weight edge connecting the tree to a vertex not yet in the tree.
- Start with an empty subset of the edges (F), and a subset of the vertices (Y) containing only a single (arbitrarily chosen) vertex, v1.
- A vertex nearest to Y (separated by the lowest weight edge) is a vertex in (V–Y) connected to a vertex in Y by an edge of minimum weight.
- The vertex nearest to Y is added to Y, and the edge is added to F. Repeat until Y = V

MST-Prim Algorithm

```
MST-Prim(G, w, r)        // r is an arbitrarily chosen vertex
 1  for each u ? G.V
 2       u.key = ?
 3       u.? = NIL
 4  r.key = 0
 5  Q = G.V                    // Q is a min-priority queue
 6  while Q ≠ ?
 7      u = Extract-min(Q)
 8      for each v ? Adj[u]
 9          if v ? Q and w(u, v) < v.key
10              v.? = u
11              v.key = w(u, v)
```

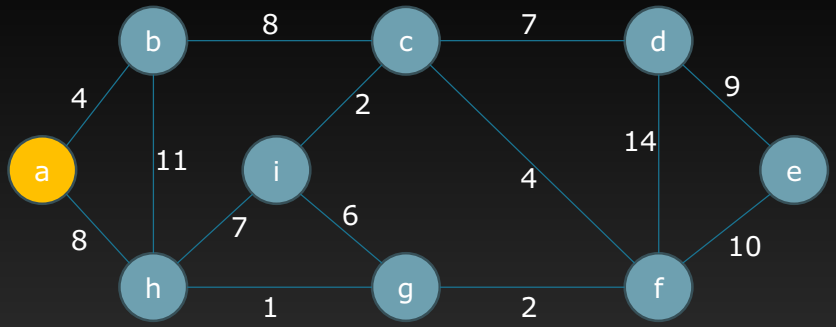
Prim's Algorithm



Y = {a}
F = {}

- Let's start (arbitrarily) from a.

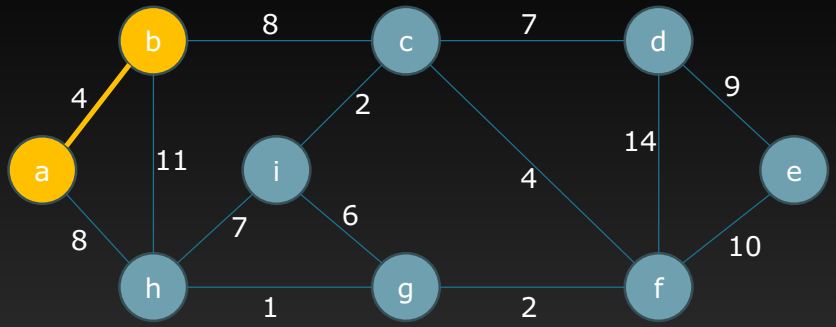
Prim's Algorithm



Y = {a}
F = {}

- Let's start (arbitrarily) from a.
- The lowest-weight edge connecting Y to (V–Y) is (a, b)
- Include this edge in F, and include b in Y.

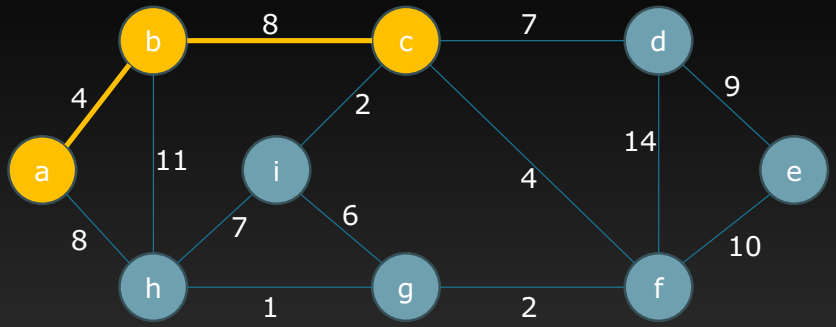
Prim's Algorithm



Y = {a, b}
F = {(a, b)}

- The lowest-weight edges connecting Y to (V–Y) are (b, c) and (a, h). We will arbitrarily choose (b, c).
- Include this edge in F, and include c in Y.

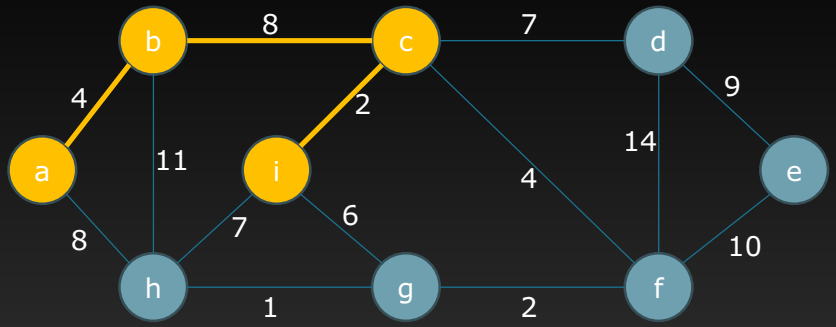
Prim’s Algorithm



Y = {a, b, c}
F = {(a, b), (b, c)}

- The lowest-weight edge connecting Y to (V–Y) is (c, i)
- Include this edge in F, and include i in Y.

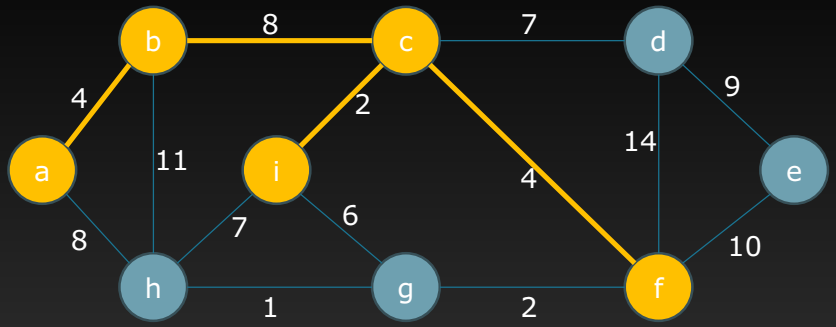
Prim's Algorithm



Y = {a, b, c, i}
F = {(a, b), (b, c), (c, i)}

- The lowest-weight edge connecting Y to (V–Y) is (c, f)
- Include this edge in F, and include f in Y.

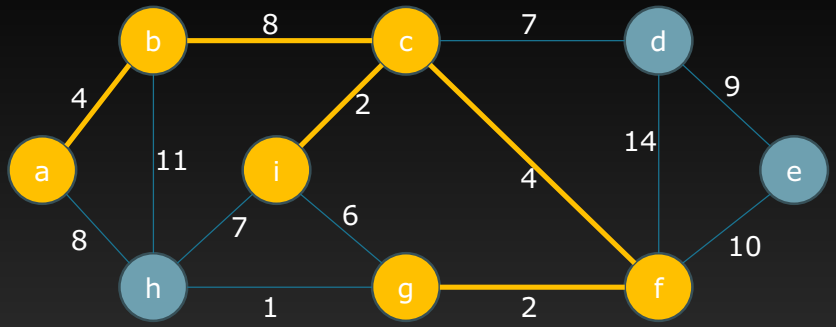
Prim's Algorithm



Y = {a, b, c, f, i}
F = {(a, b), (b, c), (c, i), (c, f)}

- The lowest-weight edge connecting Y to (V–Y) is (f, g)
- Include this edge in F, and include g in Y.

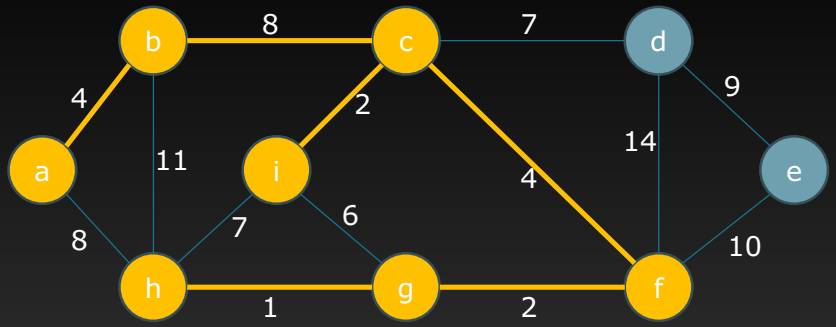
Prim's Algorithm



Y = {a, b, c, f, g, i}
F = {(a, b), (b, c), (c, i), (c, f), (f, g)}

- The lowest-weight edge connecting Y to (V–Y) is (g, h)
- Include this edge in F, and include h in Y.

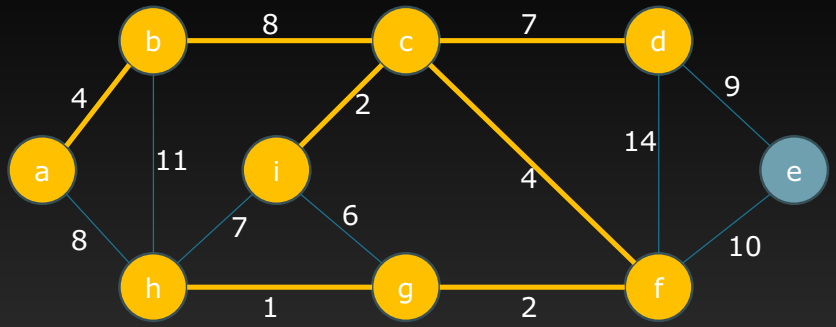
Prim's Algorithm



Y = {a, b, c, f, g, h, i}
F = {(a, b), (b, c), (c, i), (c, f), (f, g), (g, h)}

- The lowest-weight edge connecting Y to (V–Y) is (c, d)
- Include this edge in F, and include d in Y.

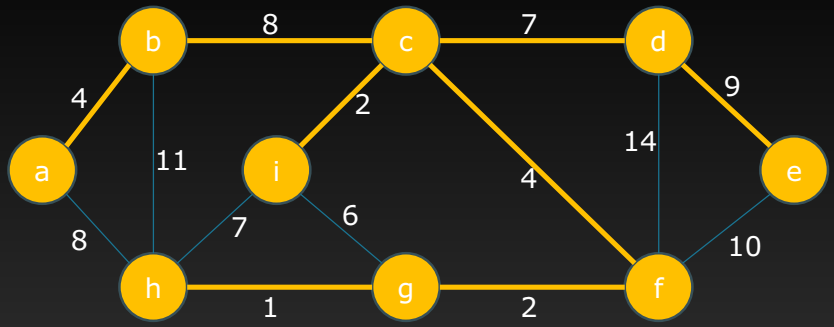
Prim's Algorithm



Y = {a, b, c, d, f, g, h, i}
F = {(a, b), (b, c), (c, i), (c, f), (f, g), (g, h), (c, d)}

- The lowest-weight edge connecting Y to (V–Y) is (d, e)
- Include this edge in F, and include e in Y.

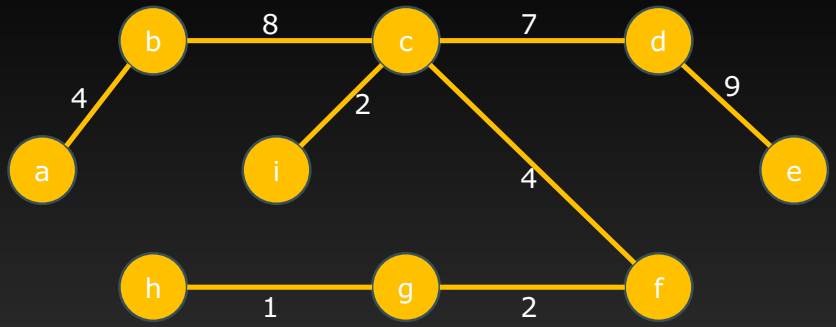
Prim's Algorithm



Y = {a, b, c, d, e, f, g, h, i}
F = {(a, b), (b, c), (c, i), (c, f), (f, g), (g, h), (c, d), (d, e)}

- Now Y = V (we have added all of the vertices to Y)
- Remove all remaining edges

Prim's Algorithm



Y = {a, b, c, d, e, f, g, h, i}
F = {(a, b), (b, c), (c, i), (c, f), (f, g), (g, h), (c, d), (d, e)}

- Now Y = V (we have added all of the vertices to Y)
- Remove all remaining edges
- MST Weight = 4 + 8 + 7 + 9 + 2 + 4 + 1 + 2 = 37

Prim's Algorithm

- Total run time: O(E lg V) (like Kruskal)
- Depends on how the priority queue is implemented. A Min-Heap is a good way, but a Fibonacci-heap (Chapter 19) is even better, but we're not covering them in this class.