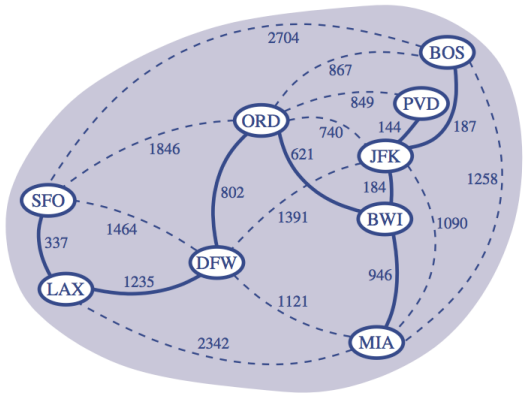
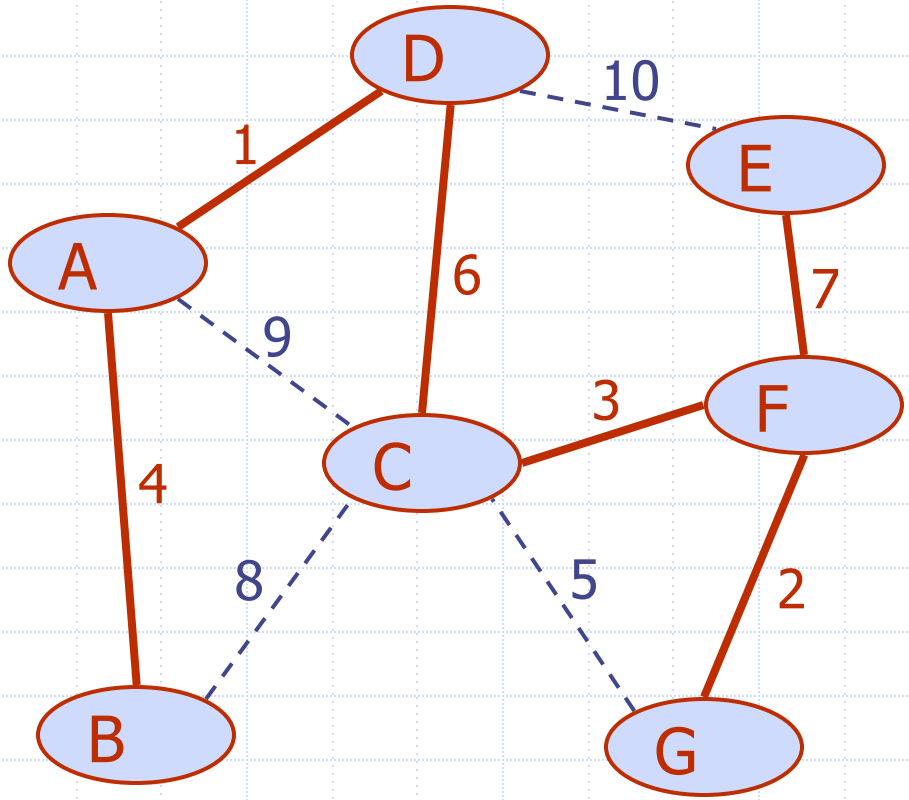
Minimum Spanning Trees



Minimum Spanning Trees

- Spanning subgraph
  - Subgraph of a graph G containing all the vertices of G
- Spanning tree
  - Spanning subgraph that is itself a (cycle free) tree
- Minimum spanning tree (MST)
  - Spanning tree of a weighted graph with minimum total edge weight
- Applications
  - Communications networks
  - Transportation networks



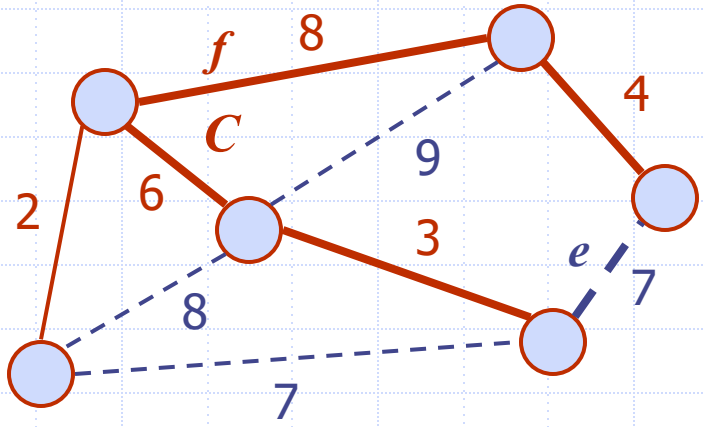
Cycle Property

Cycle Property:

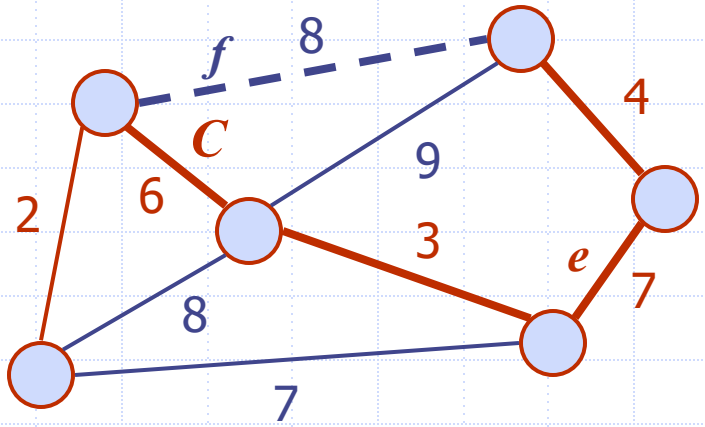
- Let T be a minimum spanning tree of a weighted graph G
- Let e be an edge of G that is not in T and C let be the cycle formed by e with T
- For every edge f of C, $weight(f) \leq weight(e)$

Proof:

- By contradiction
- If $weight(f) > weight(e)$ we can get a spanning tree of smaller weight by replacing e with f

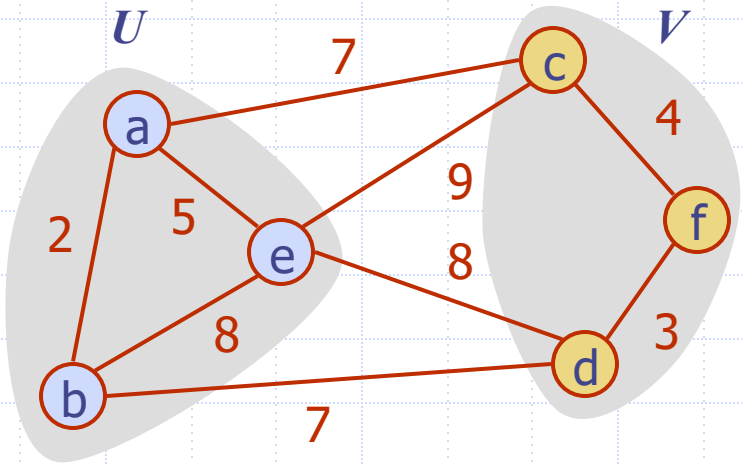


Replacing f with e yields a better spanning tree



Cuts

- A cut in G is a partition of the vertex set into two parts
  - Example:
    - U- {a, e, b}
    - V – {c, d, f}
  - Number of possible cuts
    - $2^{n-1} - 1$



- Edges in the cut
  - edges that connect vertices of one partition to vertices of the other partition.

Partition Property

Partition Property:

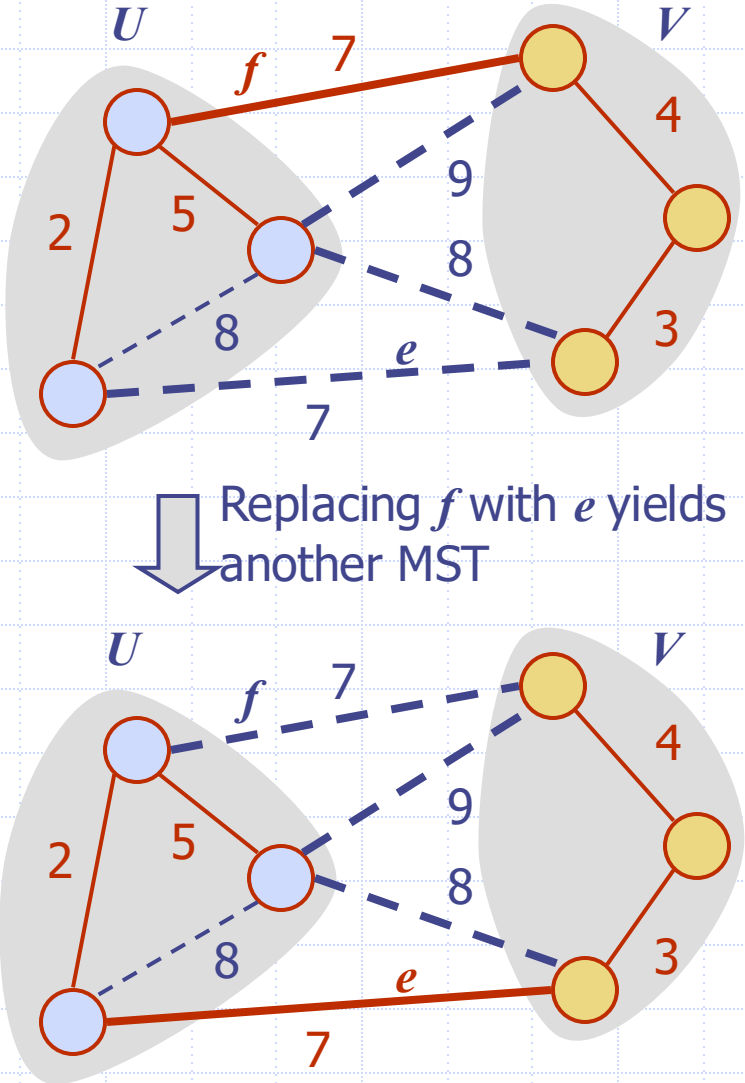
- Consider a partition of the vertices of G into subsets U and V
- Let e be an edge of minimum weight across the partition
- There is a minimum spanning tree of G containing edge e

Proof:

- Let T be an MST of G
- If T does not contain e, consider the cycle C formed by e with T and let f be an edge of C across the partition
- By the cycle property,

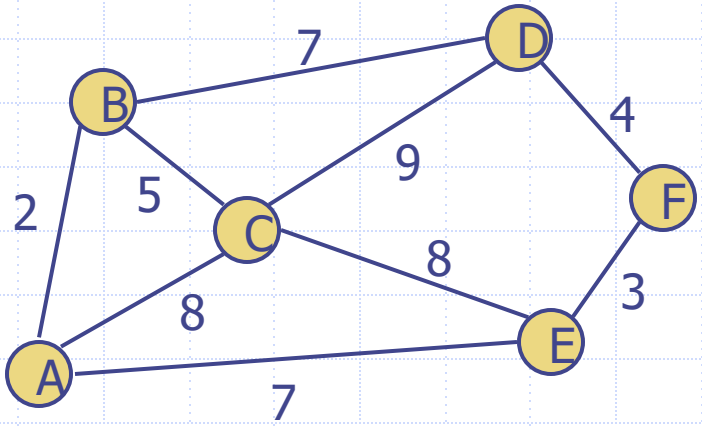
$$weight(f) \leq weight(e)$$

- Thus, $weight(f) = weight(e)$
- We obtain another MST by replacing f with e



Prim-Jarnik's Principle

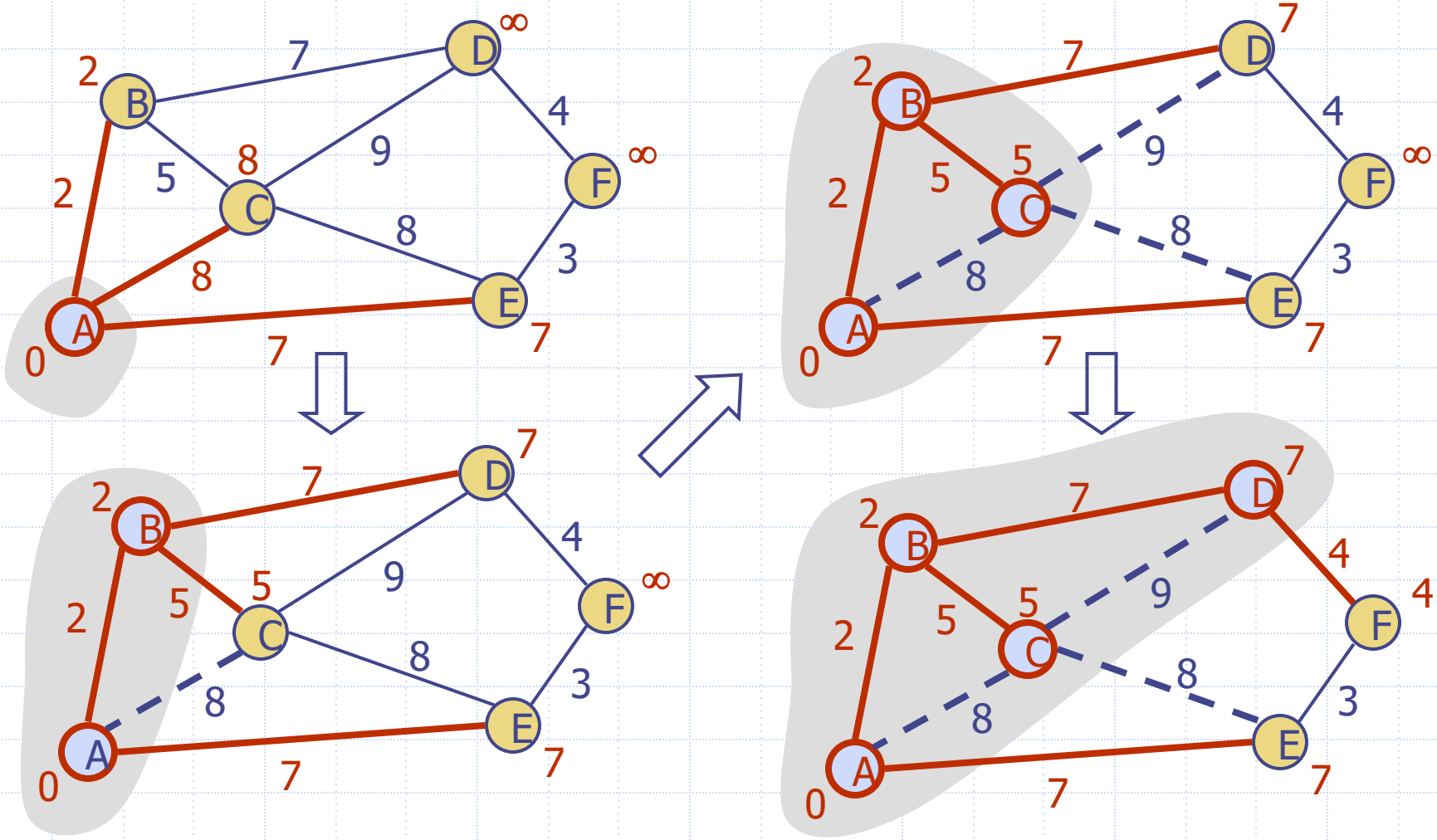
- Similar to Dijkstra's algorithm
- Utilizes the Partition Property
- We pick an arbitrary vertex s and we grow the MST as a cloud of vertices, starting from s



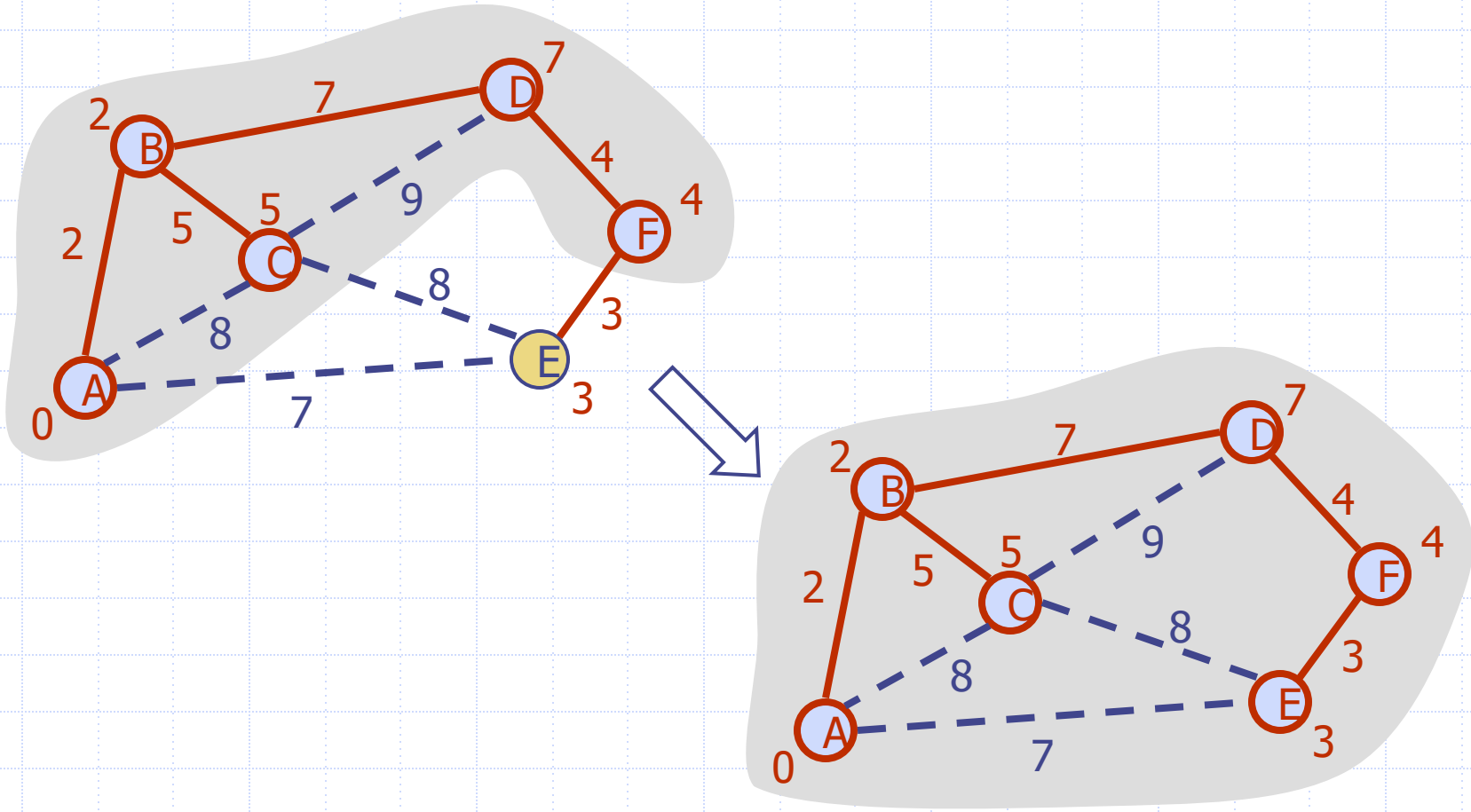
Prim-Jarnik's Algorithm - Implementation

- Similar to Dijkstra's algorithm
- We pick an arbitrary vertex s and we grow the MST as a cloud of vertices, starting from s
- We store with each vertex v label $d(v)$ representing the smallest weight of an edge connecting v to a vertex in the cloud
- At each step:
  - We add to the cloud the vertex u outside the cloud with the smallest distance label
  - We update the labels of the vertices adjacent to u

Example


A
B
C
D
E
F
0
2
7
∞
2
5
8
9
4
∞
8
3
7
7
A
B
C
D
E
F
0
2
7
7
2
5
5
9
4
∞
8
8
3
7
7
A
B
C
D
E
F
0
2
7
7
2
5
5
9
4
∞
8
8
3
7
7
A
B
C
D
E
F
0
2
7
7
2
5
5
9
4
4
8
8
3
7
7

Example (contd.)



Prim-Jarnik Pseudo-code

Algorithm PrimJarnik(G):

Input: An undirected, weighted, connected graph G with n vertices and m edges

Output: A minimum spanning tree T for G

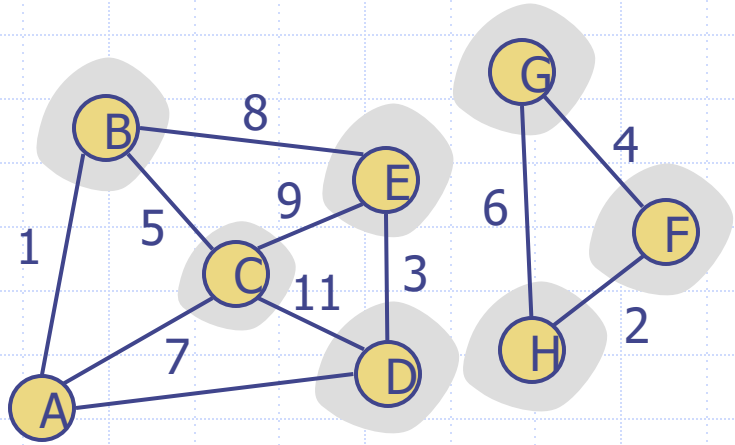
Pick any vertex s of G
$D[s] = 0$
for each vertex $v \neq s$ **do**
  $D[v] = \infty$
Initialize $T = \emptyset$.
Initialize a priority queue Q with an entry $(D[v], (v, \text{None}))$ for each vertex v, where $D[v]$ is the key in the priority queue, and (v, None) is the associated value.
while Q is not empty **do**
  $(u, e) =$ value returned by Q.remove_min()
  Connect vertex u to T using edge e.
  for each edge $e' = (u, v)$ such that v is in Q **do**
    {check if edge (u, v) better connects v to T}
    if $w(u, v) < D[v]$ **then**
      $D[v] = w(u, v)$
      Change the key of vertex v in Q to $D[v]$.
      Change the value of vertex v in Q to (v, e').
return the tree T

Analysis

- Graph operations
  - We cycle through the incident edges once for each vertex
- Label operations
  - We set/get the distance, parent and locator labels of vertex z $O(\deg(z))$ times
  - Setting/getting a label takes $O(1)$ time
- Priority queue operations
  - Each vertex is inserted once into and removed once from the priority queue, where each insertion or removal takes $O(\log n)$ time
  - The key of a vertex w in the priority queue is modified at most $\deg(w)$ times, where each key change takes $O(\log n)$ time
- Prim-Jarnik's algorithm runs in $O((n + m) \log n)$ time provided the graph is represented by the adjacency list structure
  - Recall that $\sum_v \deg(v) = 2m$
- The running time is $O(m \log n)$ since the graph is connected

Kruskal's Principle

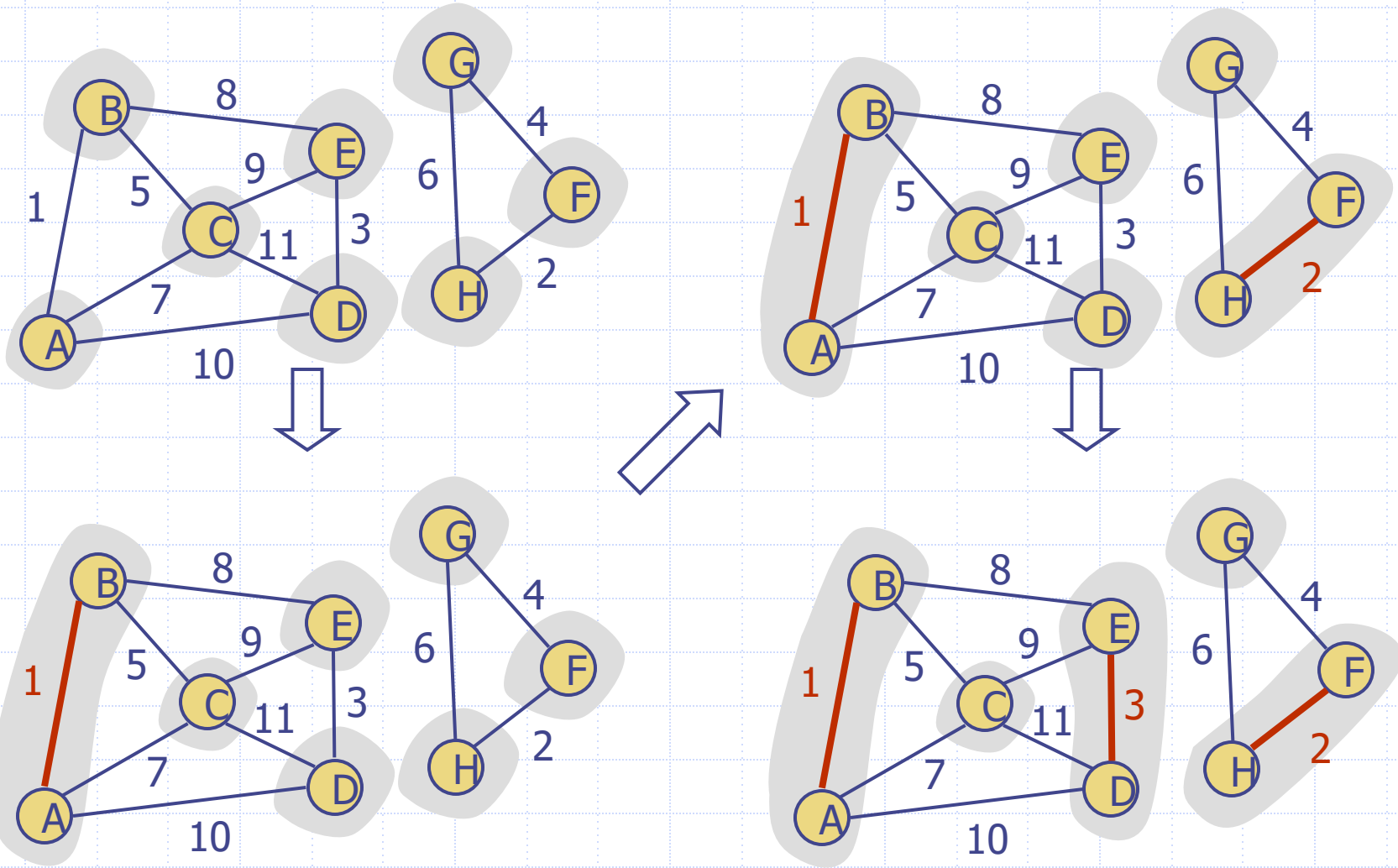
- Greedy Algorithm
  - greedily add the edge with minimum weight to the MST
  - ensure that the added edge does not form a cycle



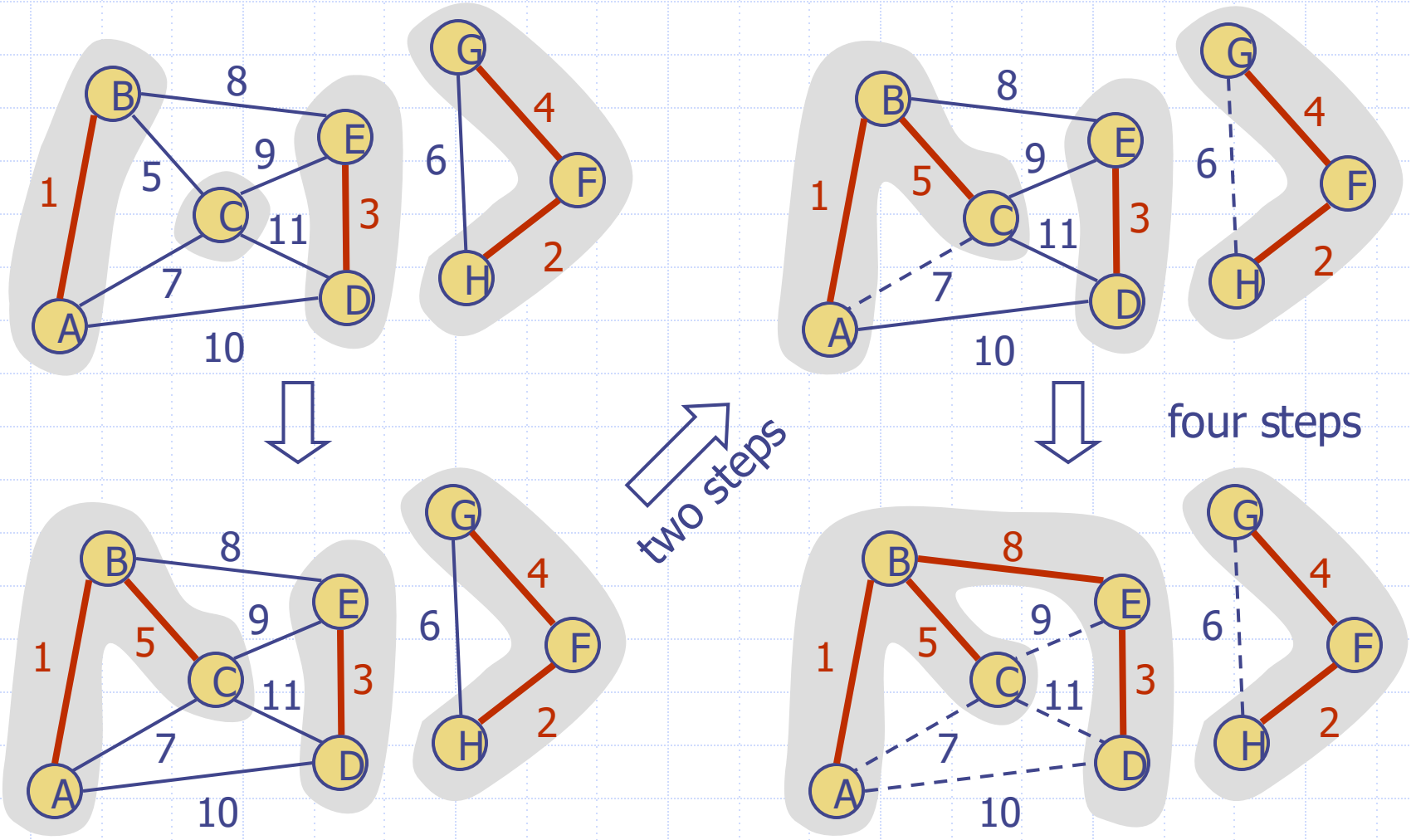
Kruskal's Algorithm

- Maintain a partition of the vertices into clusters
  - Initially, single-vertex clusters
  - Keep an MST for each cluster
  - Merge "closest" clusters and their MSTs
- A priority queue stores the edges outside clusters
  - Key: weight
  - Element: edge
- At the end of the algorithm
  - One cluster and one MST

Example



Example (contd.)



Kruskal's Algorithm

Algorithm Kruskal(G):

Input: A simple connected weighted graph G with n vertices and m edges

Output: A minimum spanning tree T for G

for each vertex v in G **do**

Define an elementary cluster $C(v) = \{v\}$.

Initialize a priority queue Q to contain all edges in G, using the weights as keys.

$T = \emptyset$ {T will ultimately contain the edges of the MST}

while T has fewer than $n-1$ edges **do**

(u,v) = value returned by Q.remove_min()

Let $C(u)$ be the cluster containing u, and let $C(v)$ be the cluster containing v.

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

Add edge (u,v) to T.

Merge $C(u)$ and $C(v)$ into one cluster.

return tree T

Correctness of Kruskal's Algorithm

- Let all the edges in the graph have distinct weights
- Let the tree obtained from Kruskal's algorithm contain the edges $g_1<g_2<g_3\ldots<g_{n-1}$
- Let the Optimum MST have the edges $f_1<f_2<f_3\ldots<f_{n-1}$
  - Let i be the first edge that is different between g's and f's.
    - case1: $g_i<f_i$
    - case2: $f_i<g_i$

Is the MST unique?

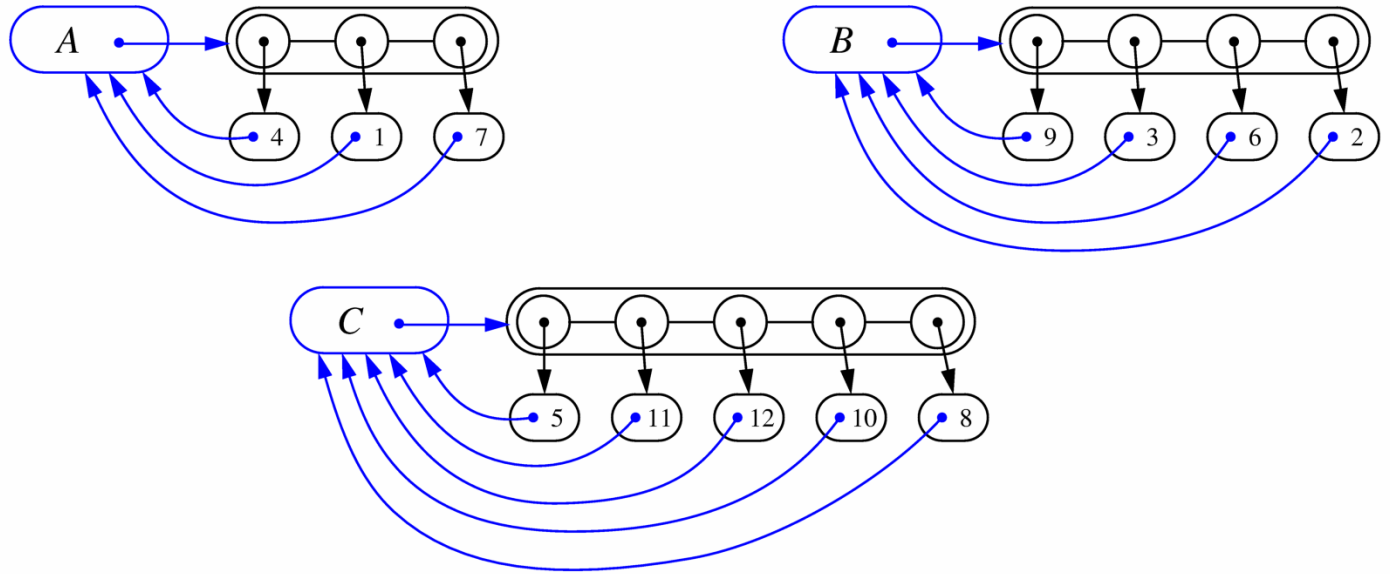
- yes, if the edge weights are distinct.
- no, otherwise.

Data Structure for Kruskal's Algorithm

- The algorithm maintains a forest of trees
- A priority queue extracts the edges by increasing weight
- An edge is accepted it if connects distinct trees
- We need a data structure that maintains a partition, i.e., a collection of disjoint sets, with operations:
  - makeSet(u): create a set consisting of u
  - find(u): return the set storing u
  - union(A, B): replace sets A and B with their union
- Total time
  - O(m log n + time for Union* n+ time for Find* m)

List-based Implementation

- Each set is stored in a sequence represented with a linked-list
- Each node should store an object containing the element and a reference to the set name

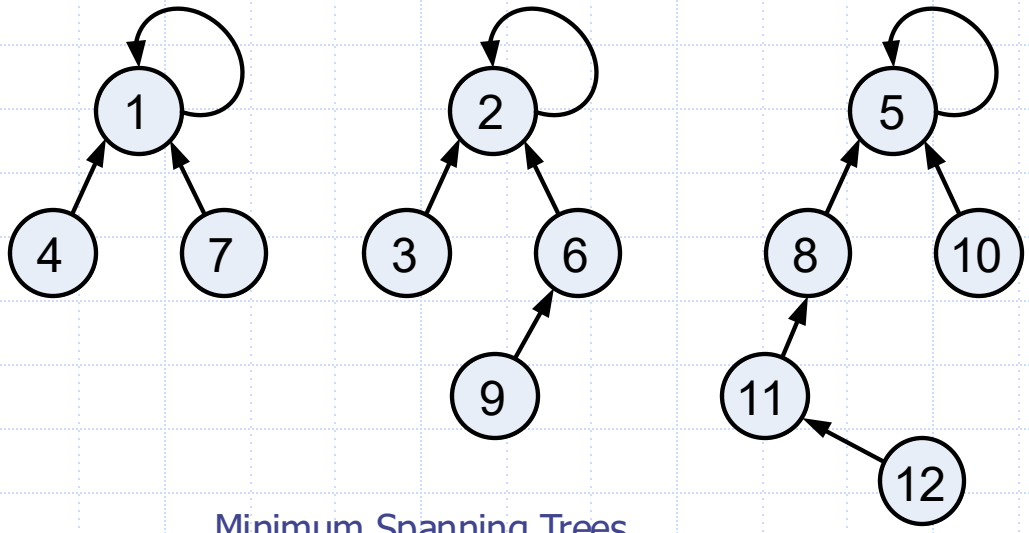


Analysis of List-based Representation

- Union
  - merging of two linked lists
  - O(n)
- Find
  - search through the list
  - O(n)
- Total time
  - O(m log n)+O(n^2)+O(mn).

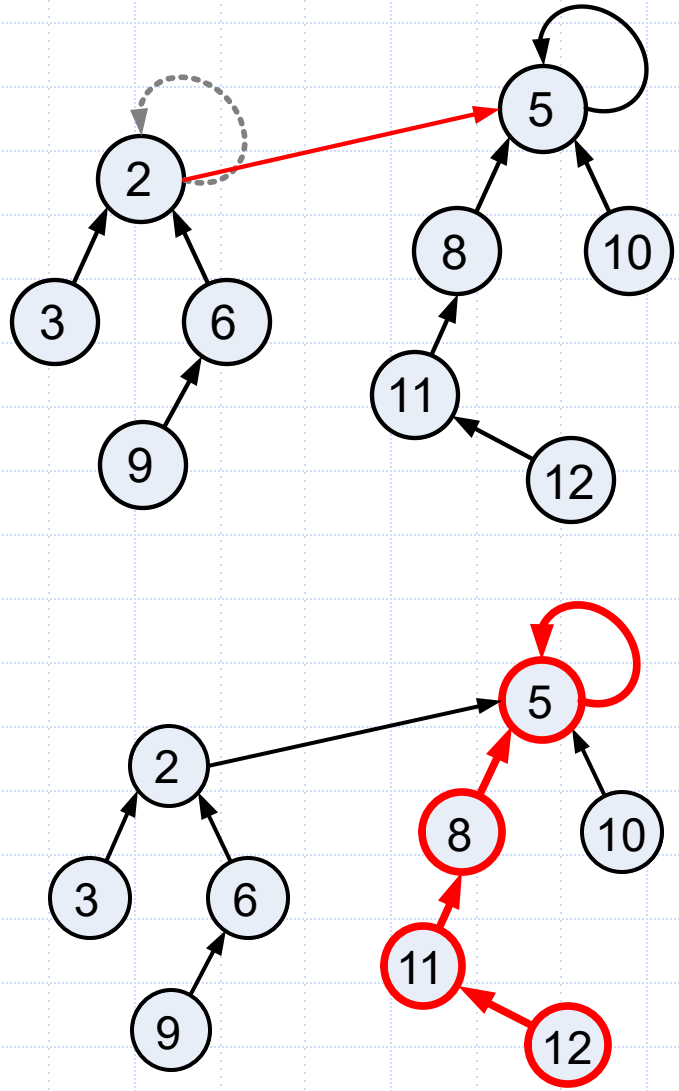
Tree-based Implementation

- Each element is stored in a node, which contains a pointer to a set name
- A node v whose set pointer points back to v is also a set name
- Each set is a tree, rooted at a node with a self-referencing set pointer
- For example: The sets “1”, “2”, and “5”:



Union-Find Operations

- To do a union, simply make the root of one tree point to the root of the other
- To do a find, follow set-name pointers from the starting node until reaching a node whose set-name pointer refers back to itself
- Amortized time is O(n log*n)



The End!

Course Schedule

week 1 - Introduction OOP, Arrays, and Linked Lists
week 2 - Analysis Tools and Recursion
week 3 - Stacks and Queues
week 4 - Tree Structures and Heaps
week 5 - Sorting and Priority Queues
week 6 - Maps, Hash Tables
week 7 - Search Tree Structures I, **Quiz 1**
week 8 - Search Tree Structures II
week 9 - Search Tree Structures III
week 10 - Buffer week
week 11 - String Processing
week 12 - Dynamic Programming, **Quiz 2**
week 13 - Graphs I
week 14 - Graphs II
week 15 - Graphs III
week 16 - Buffer week

Arrays and Linked Lists

- Chapter 3, Sections 3.1-3.4
- Arrays
  - Insertion sort
- Singly Linked Lists
- Doubly Linked Lists

Analysis of Algorithms

- Chapter 4, Sections 4.1-4.4
- Writing and analyzing pseudo codes
- Big Oh Notation
  - asymptotic algorithm analysis
- Big Omega and Big Theta notations
- Proof methods
- mathematical functions and their relationships

Recursion

- Chapter 5, Section 5.1-5.5
- Defining a recursion pattern
- Analysis of recursion
- Types of recursion
  - Linear, binary and multiple
- Pitfalls of recursion

Stacks and Queues

- Chapter 6, Sections 6.1-6.3
- ADT
- Stacks
  - ADT definition
  - Implementations
  - Applications
- Queues
  - ADT definition
  - Implementations
  - Applications
  - Circular Queues

Tree Structures

- Chapter 8, Sections 8.1-8.4
- Generic Tree ADT
- Binary Trees
  - properties
- Tree Representations
- Tree Traversals
  - preorder
  - postorder
  - inorder
  - breadth first

Heaps and Priority Queues

- Chapter 9, Sections 9.1-9.4
- Prority Queue
  - ADT Definition
  - Implementation using sorted/unsorted lists
- Heaps
  - Definition
  - Priority Queue as a heap
  - Analysis of Upheap and Downheap procedures
  - Heap construction methods
    - analysis
- Sorting with a priority queue
  - selection sort
  - insertion sort
  - heap sort

Searching and Search Tree Structures (chapter 11)

- Binary Search Trees
  - insertion and deletion operations
- Balanced Search Trees
- AVL Trees
  - operations
- (2,4) Trees
  - multi-way search trees

Hash Tables and Maps

- Chapter 10, Sections 10.1-10.3
- Map
  - ADT
  - applications
- Hashing
  - Hash functions
  - Collision handling schemes
  - load factors and efficiency

String Processing and Dynamic Programming (Chapter 12)

- Pattern Matching Algorithms
  - The Boyer-Moore Algorithm
    - Bad character and good suffix rules
- Tries
  - Standard Tries
  - Compressed Tries
  - Suffix Tries
    - Prefix Matching
- Dynamic Programming
  - Methodology

Sorting (Chapter 13)

- Divide and Conquer Paradigm
- Merge Sort
  - algorithm
  - complexity analysis
- Quick Sort/Randomized Quick Sort
  - algorithm
  - complexity analysis
  - In-Place partitioning
- Radix Sort
  - Radix Exchange Sort
  - Straight Radix Sort
    - Stable way
- Bucket sort

Graphs (Chapter 14)

- Graphs
  - Terminology
- Data Structures for representing graphs
  - edge list, adjacency list, adjacency matrix
- Graph Traversals
  - BFS, DFS
- Directed Graphs
- Transitive Closure
- Directed Acyclic Graphs
  - topological order
- Shortest Paths
  - weighted graphs, Dijkstra's algorithm, Bellman-ford algorithm
- Minimum Spanning Trees
  - Prim-Jarnik Algorithm, Kruskal's Algorithm, Data structure for Kruskal's algorithm.

Good Luck