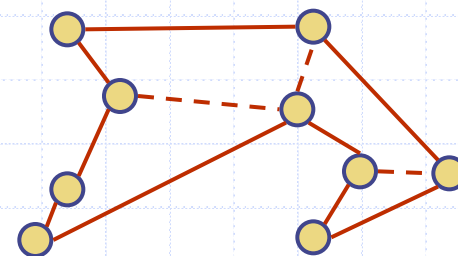
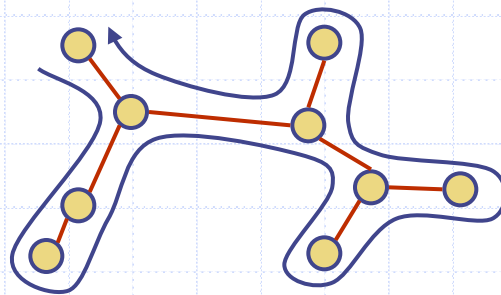


Campus Tour



Graph Assignment

◆ Goals

- Learn and implement the adjacency matrix structure and Kruskal's minimum spanning tree algorithm
- Understand and use the decorator pattern and various JDSL classes and interfaces

◆ Your task

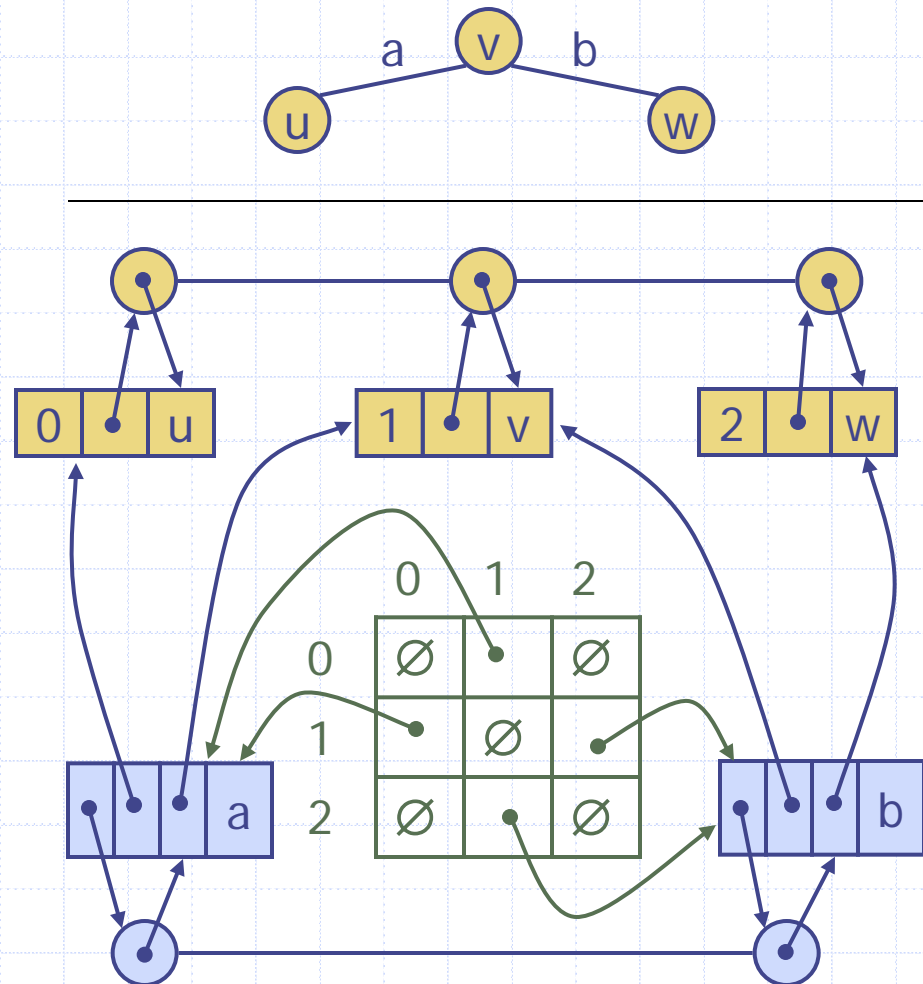
- Implement the adjacency matrix structure for representing a graph
- Implement Kruskal's MST algorithm

◆ Frontend

- Computation and visualization of an approximate traveling salesperson tour

Adjacency Matrix Structure

- ◆ Edge list structure
- ◆ Augmented vertex objects
 - Integer key (index) associated with vertex
- ◆ 2D-array adjacency array
 - Reference to edge object for adjacent vertices
 - Null for non adjacent vertices



Kruskal's Algorithm

- ◆ The vertices are partitioned into clouds
 - We start with one cloud per vertex
 - Clouds are merged during the execution of the algorithm
- ◆ Partition ADT:
 - **makeSet**(*o*): create set {*o*} and return a locator for object *o*
 - **find**(*l*): return the set of the object with locator *l*
 - **union**(*A*,*B*): merge sets *A* and *B*

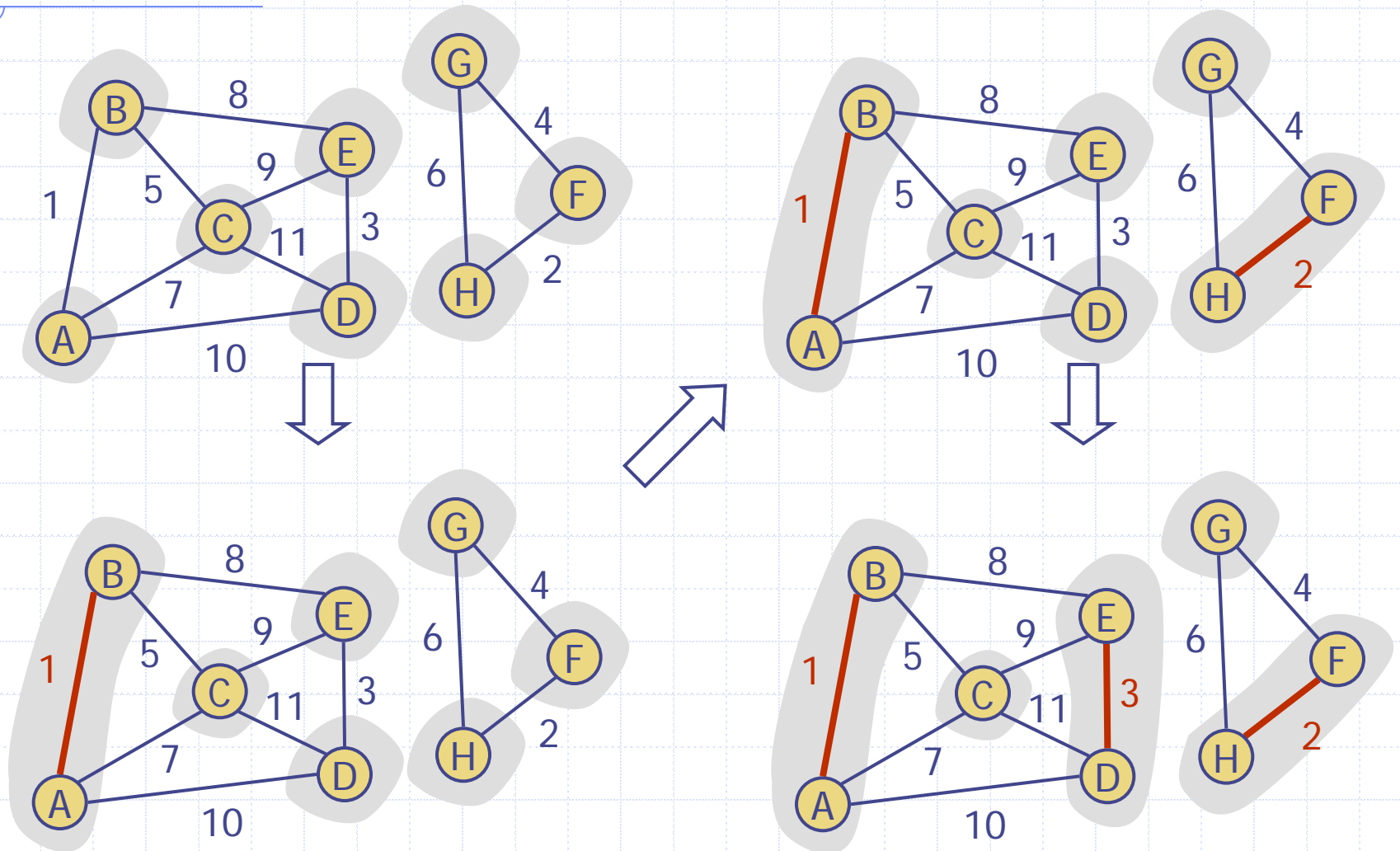
Algorithm **KruskalMSF**(*G*)

Input weighted graph *G*

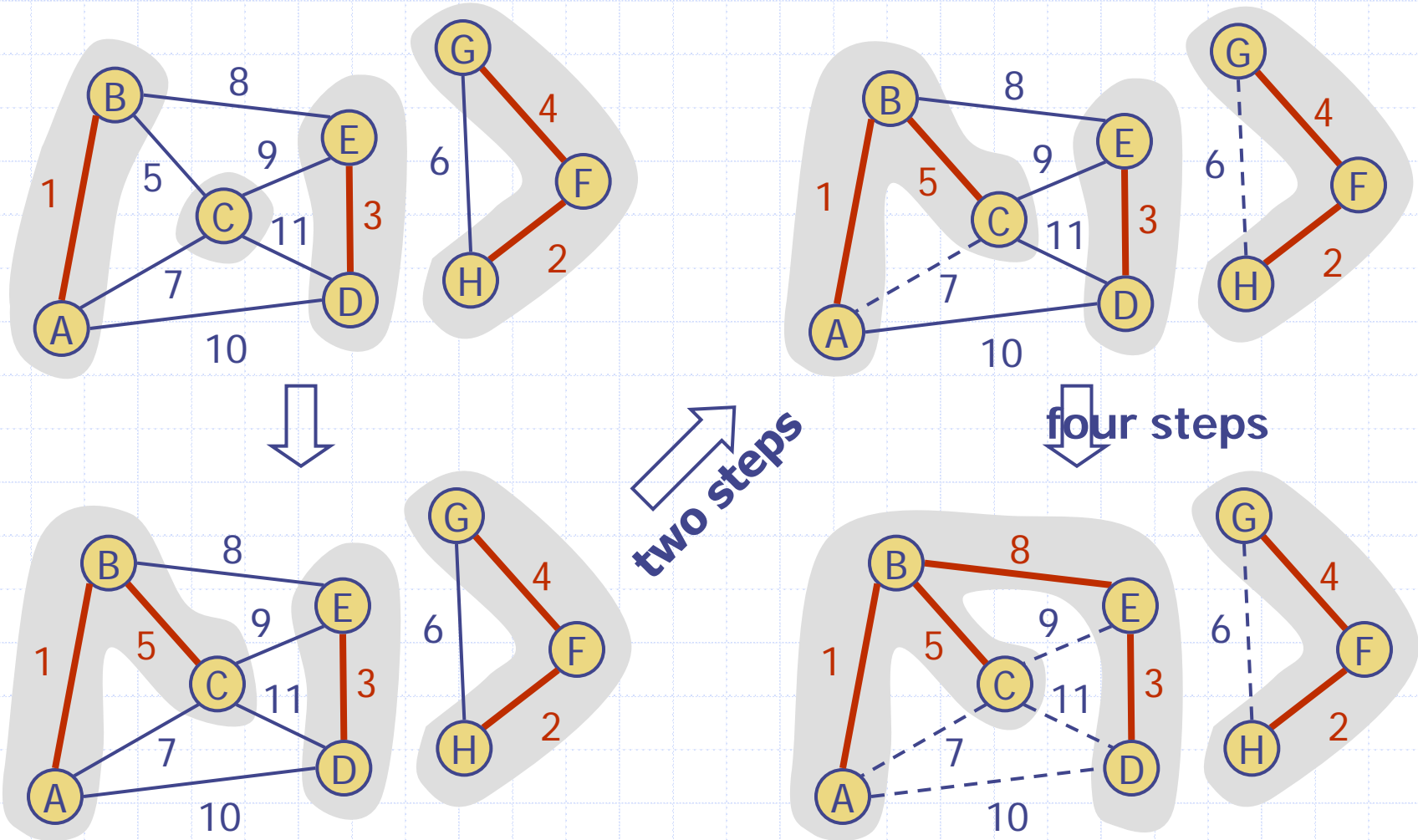
Output labeling of the edges of a minimum spanning forest of *G*

```
Q ← new heap-based priority queue
for all v ∈ G.vertices() do
    l ← makeSet(v) { elementary cloud }
    v.setLocator(l)
for all e ∈ G.edges() do
    Q.insert(e.weight(), e)
while ¬Q.empty()
    e ← Q.removeMin()
    [u,v] ← e.endVertices()
    A ← find(u.getLocator())
    B ← find(v.getLocator())
    if A ≠ B
        setMSFedge(e)
        { merge clouds }
        union(A, B)
```

Example



Example (contd.)



Partition Implementation

◆ Partition implementation

- A set is represented the sequence of its elements
- A position stores a reference back to the sequence itself (for operation *find*)
- The position of an element in the sequence serves as locator for the element in the set
- In operation *union*, we move the elements of the smaller sequence into to the larger sequence

◆ Worst-case running times

- *makeSet, find*: $O(1)$
- *union*: $O(\min(n_A, n_B))$

◆ Amortized analysis

- Consider a series of k Partiton ADT operations that includes n *makeSet* operations
- Each time we move an element into a new sequence, the size of its set at least doubles
- An element is moved at most $\log_2 n$ times
- Moving an element takes $O(1)$ time
- The total time for the series of operations is $O(k + n \log n)$

Analysis of Kruskal's Algorithm

◆ Graph operations

- Methods *vertices* and edges are called once
- Method *endVertices* is called m times

◆ Priority queue operations

- We perform m *insert* operations and m *removeMin* operations

◆ Partition operations

- We perform n *makeSet* operations, $2m$ *find* operations and no more than $n - 1$ *union* operations

◆ Label operations

- We set vertex labels n times and get them $2m$ times

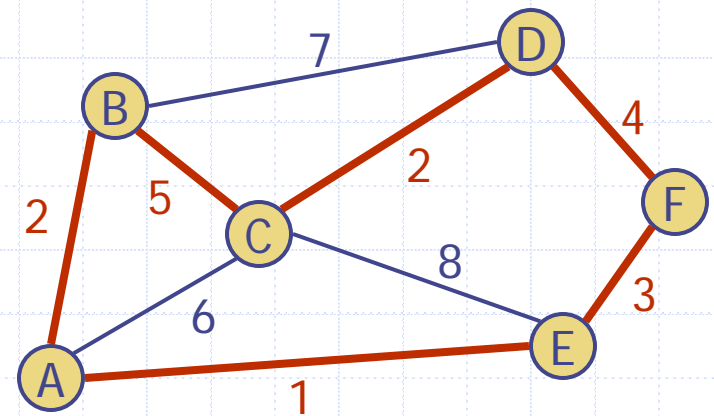
◆ Kruskal's algorithm runs in time $O((n + m) \log n)$ time provided the graph has no parallel edges and is represented by the adjacency list structure

Decorator Pattern

- ◆ Labels are commonly used in graph algorithms
 - Auxiliary data
 - Output
- ◆ Examples
 - DFS: unexplored/visited label for vertices and unexplored/ forward/back labels for edges
 - Dijkstra and Prim-Jarnik: distance, locator, and parent labels for vertices
 - Kruskal: locator label for vertices and MSF label for edges
- ◆ The decorator pattern extends the methods of the Position ADT to support the handling of attributes (labels)
 - **has**(*a*): tests whether the position has attribute *a*
 - **get**(*a*): returns the value of attribute *a*
 - **set**(*a*, *x*): sets to *x* the value of attribute *a*
 - **destroy**(*a*): removes attribute *a* and its associated value (for cleanup purposes)
- ◆ The decorator pattern can be implemented by storing a dictionary of (attribute, value) items at each position

Traveling Salesperson Problem

- ◆ A tour of a graph is a spanning cycle (e.g., a cycle that goes through all the vertices)
- ◆ A traveling salesperson tour of a weighted graph is a tour that is simple (i.e., no repeated vertices or edges) and has minimum weight
- ◆ No polynomial-time algorithms are known for computing traveling salesperson tours
- ◆ The traveling salesperson problem (TSP) is a major open problem in computer science
 - Find a polynomial-time algorithm computing a traveling salesperson tour or prove that none exists



Example of traveling salesperson tour (with weight 17)

TSP Approximation

- ◆ We can approximate a TSP tour with a tour of at most twice the weight for the case of Euclidean graphs
 - Vertices are points in the plane
 - Every pair of vertices is connected by an edge
 - The weight of an edge is the length of the segment joining the points
- ◆ Approximation algorithm
 - Compute a minimum spanning tree
 - Form an Eulerian circuit around the MST
 - Transform the circuit into a tour

