Week 10
Graphs
Selection Algorithms

Algorithms and Data Structures

COMP3506/7505

Week 10 – Graphs & Selection

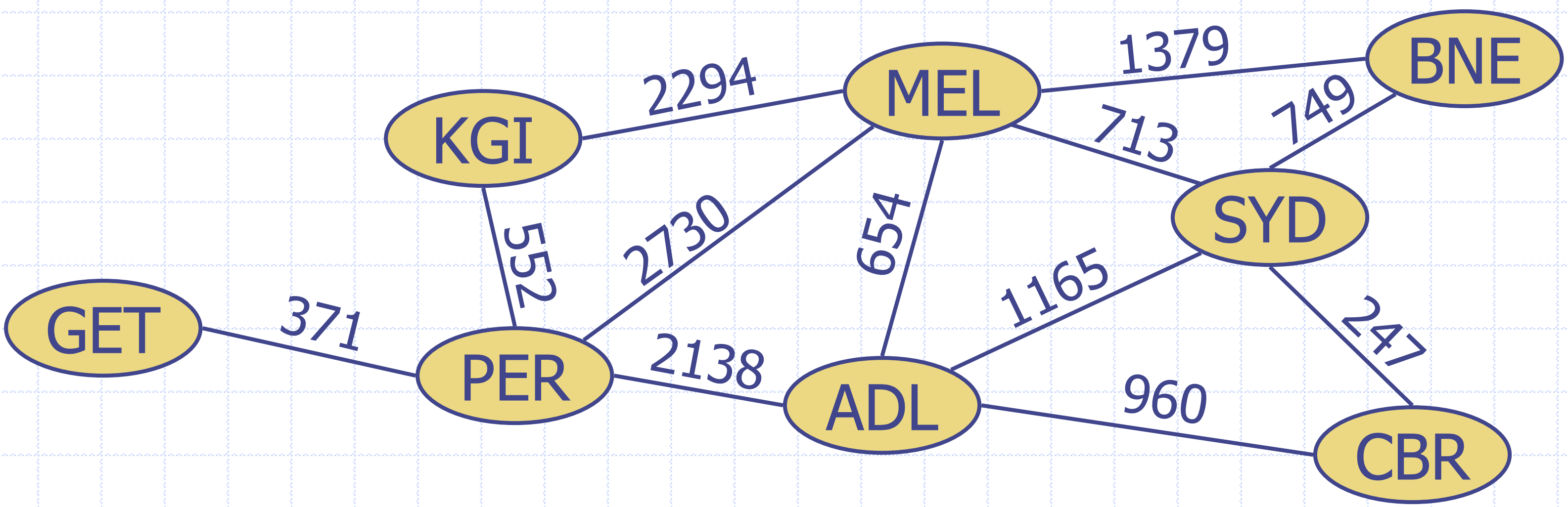
1. Shortest path algorithms
2. Minimum spanning trees
3. Selection algorithms

Week 10 – Graphs & Selection

1. Shortest path algorithms
2. Minimum spanning trees
3. Selection algorithms

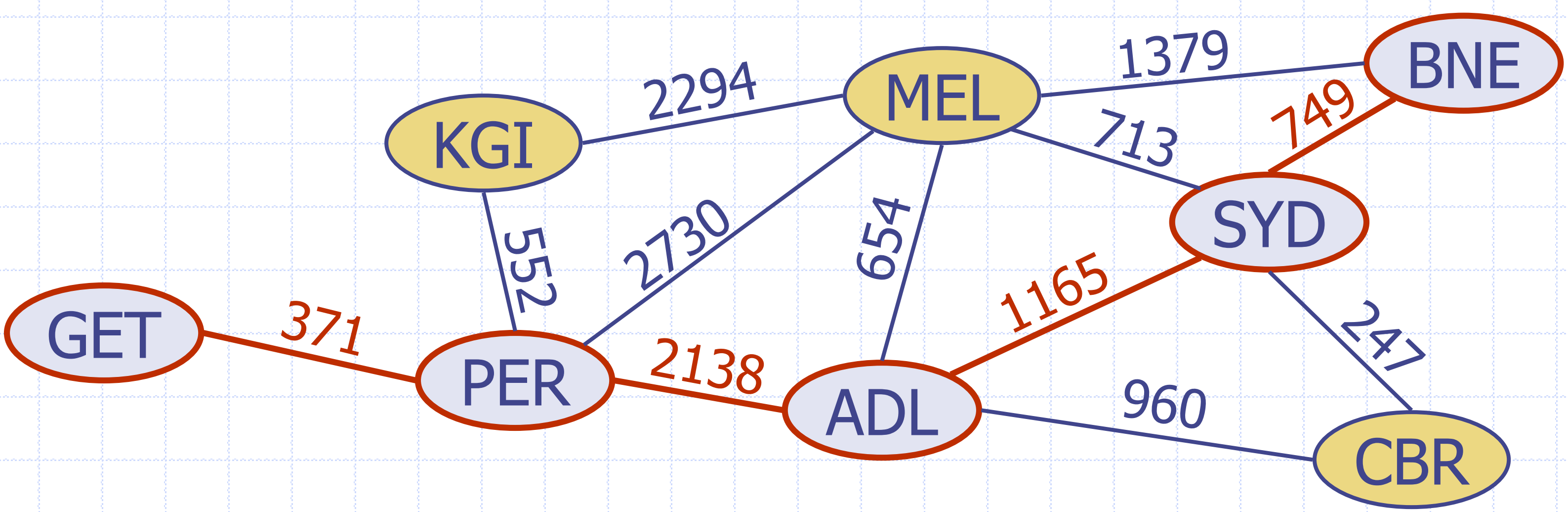
Weighted Graphs

- Each edge has an associated numerical value, called the weight of the edge
- Edge weights may represent, distances, costs, etc.
  - e.g. flight route graph: weight of an edge represents the distance between the endpoint airports



Shortest Path

- Given a weighted graph and two vertices u and v, find a path of minimum total weight between u and v
  - length of a path is the sum of the weights of its edges
- Example:
  - shortest path between Brisbane and Geraldton



Shortest Path Properties

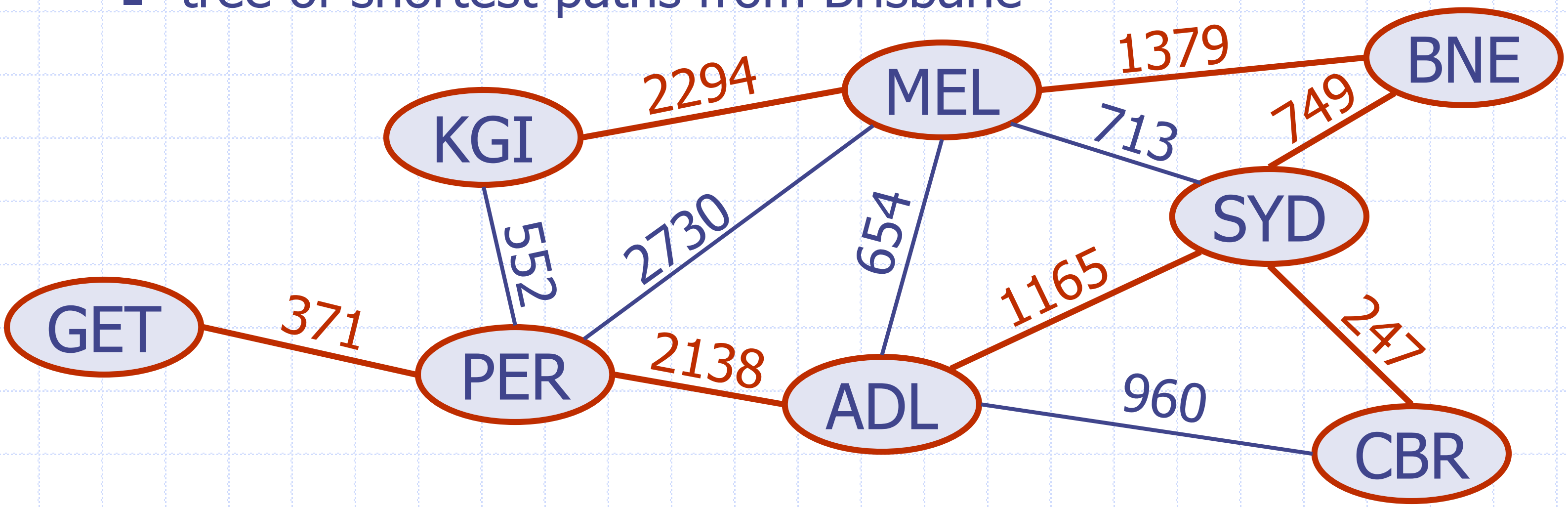
Property 1:

Subpath of a shortest path is itself a shortest path

Property 2:

There is a tree of shortest paths from a start vertex to all other vertices

- tree of shortest paths from Brisbane



Dijkstra’s Algorithm

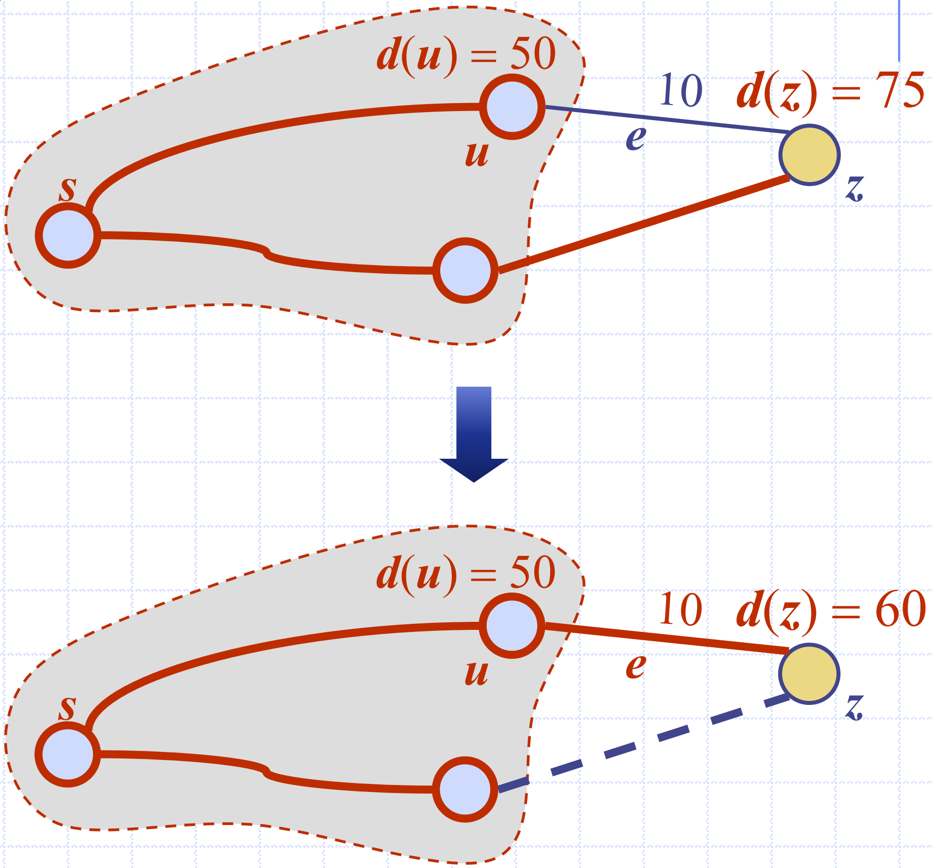
- Distance of a vertex v from a vertex s is the length of the shortest path between s and v
- Dijkstra’s algorithm computes the distances of all the vertices from a given start vertex s
- Assumptions
  - graph is connected
  - edges are undirected
  - edge weights are not negative

Dijkstra's Algorithm

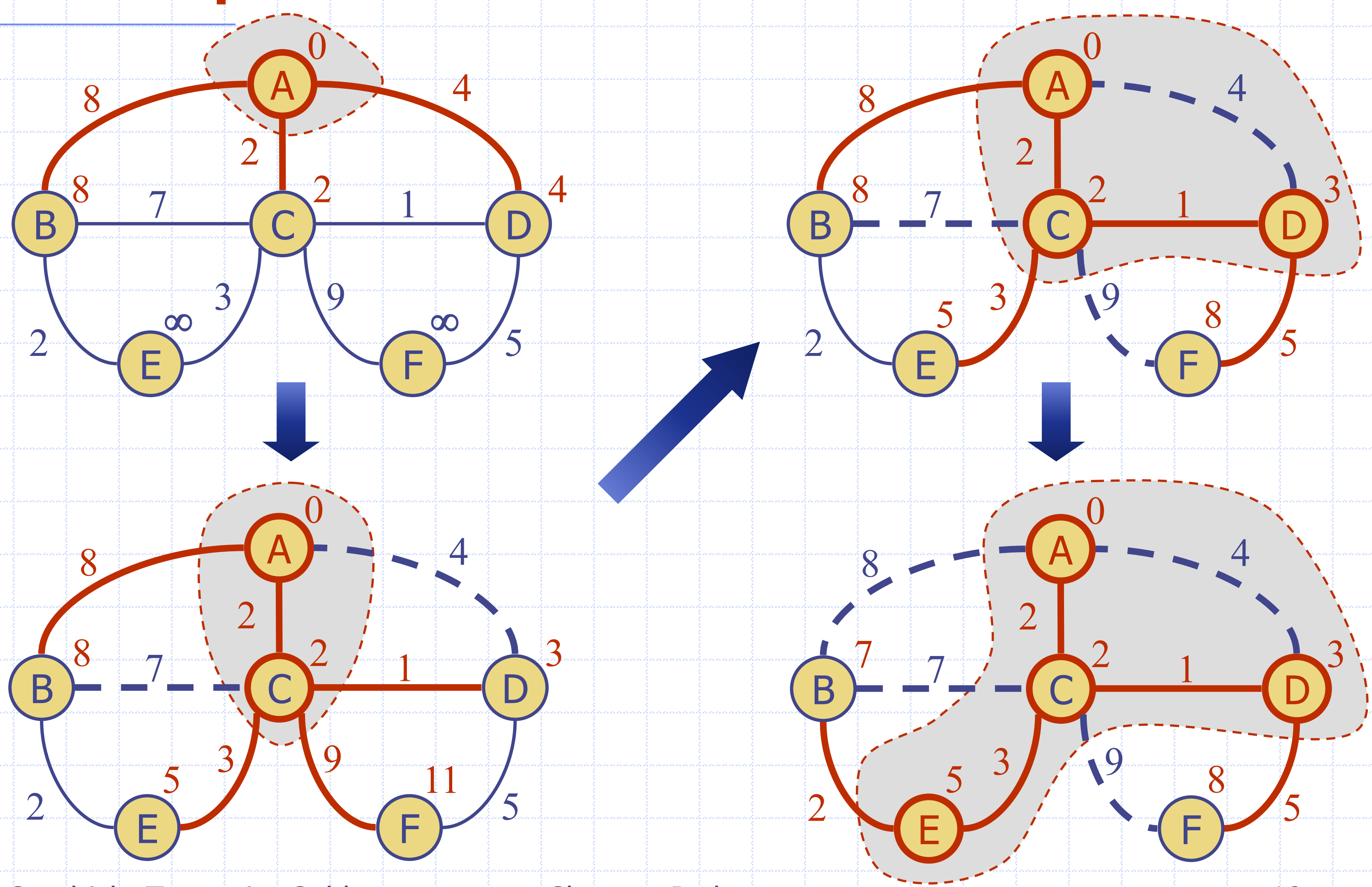
- Grow a "cloud" of vertices, beginning with s and eventually covering all the vertices
- Store a label $d(v)$ at each vertex v
  - representing distance of v from s in the subgraph consisting of the cloud and its adjacent vertices
- At each step
  - add to the cloud the vertex u outside the cloud with the smallest distance label, $d(u)$
  - update the labels of the vertices adjacent to u

Edge Relaxation

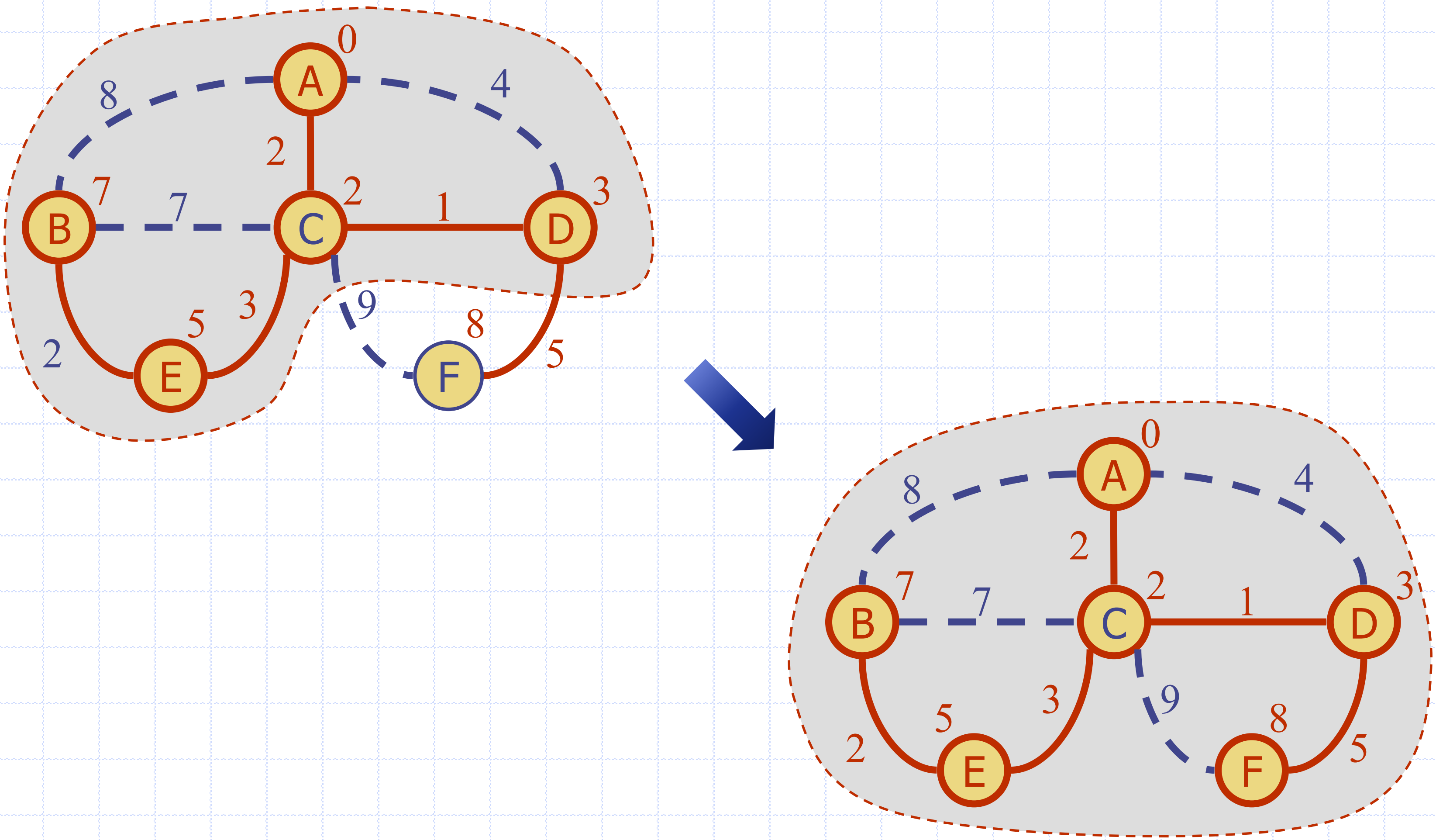
- Consider an edge $e = (u, z)$ where
  - u is the vertex most recently added to the cloud
  - z is not in the cloud
- Relaxation of edge e updates distance $d(z)$
  - $d(z) \leftarrow \min\{d(z),\ d(u) + weight(e)\}$



Example



Example (cont.)



Dijkstra's Algorithm

- Priority queue stores the vertices outside the cloud
  - Key: distance
  - Element: vertex
- Locator-based methods
  - *insert*(k, e) returns a locator
  - *replaceKey*(l, k) changes the key of an item
- Store two labels with each vertex
  - Distance ($d(v)$ label)
  - locator in priority queue

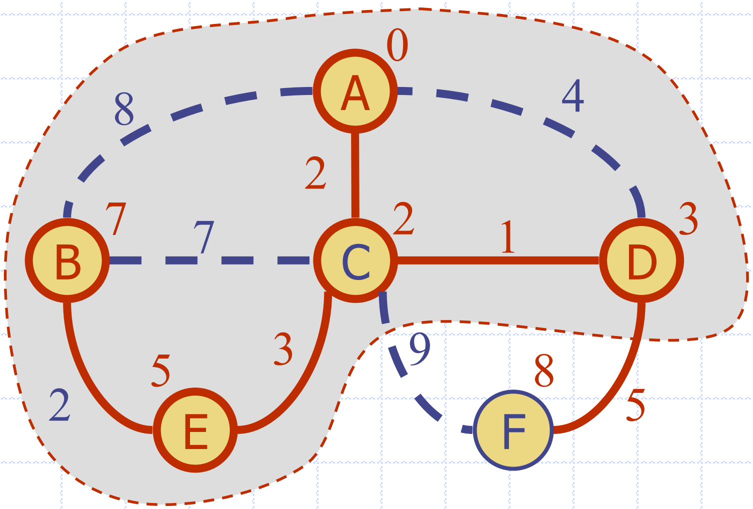
Algorithm *DijkstraDistances*(G, s):
  $PQ \leftarrow$ new heap-based priority queue
  for all $v \in G.vertices()$
    if $v = s$
      $setDistance(v, 0)$
    else
      $setDistance(v, \infty)$
    $PQ.insert(getDistance(v), v)$
  while $\neg PQ.isEmpty()$
    $u \leftarrow PQ.removeMin()$
    for all $e \in G.incidentEdges(u)$
      { relax edge e }
      $z \leftarrow G.opposite(u, e)$
      $r \leftarrow getDistance(u) + weight(e)$
      if $r < getDistance(z)$
        $setDistance(z, r)$
        $PQ.replaceKey(getLocator(z), r)$

Analysis of Dijkstra's Algorithm

- Graph operations
  - find all the incident edges once for each vertex
- Label operations
  - set/get the distance and locator labels of vertex z $\mathbf{O}(\deg(z))$ times
  - setting/getting a label takes $\mathbf{O}(1)$ time
- Priority queue operations
  - each vertex is inserted once into and removed once from the priority queue, where each insertion or removal takes $\mathbf{O}(\log n)$ time
  - key of a vertex in the priority queue is modified at most $\deg(w)$ times, where each key change takes $\mathbf{O}(\log n)$ time
- Dijkstra's algorithm runs in $\mathbf{O}((n + m) \log n)$ time
  - provided the graph is implemented as an adjacency list/map
  - recall that $\Sigma_v \deg(v) = 2m$
  - can also be expressed as $\mathbf{O}(m \log n)$ since the graph is connected

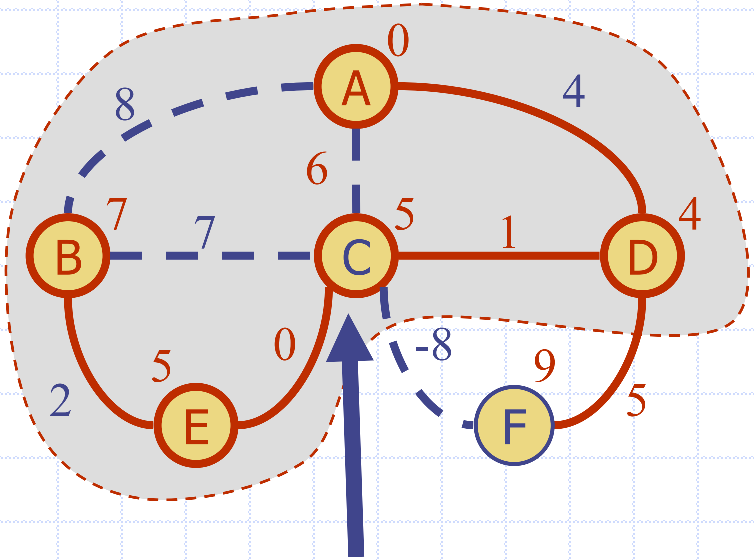
Why Dijkstra's Algorithm Works

- Dijkstra's algorithm is based on the greedy method
  - adds vertices by increasing distance
- Suppose it didn't find all shortest distances. Let F be the first wrong vertex the algorithm processed.
- When previous node, D, on the true shortest path was considered, its distance was correct
- But the edge (D,F) was relaxed at that time!
- Thus, so long as $d(F)>d(D)$, F's distance cannot be wrong. That is, there is no wrong vertex



Why it Doesn't Work for Negative-Weight Edges

- Dijkstra's algorithm is based on the greedy method
  - adds vertices by increasing distance
  - If a node with a negative incident edge were to be added late to the cloud, it could mess up distances for vertices already in the cloud.



C's true distance is **1**, but it is already in the cloud with *d*(*C*)=**5**!

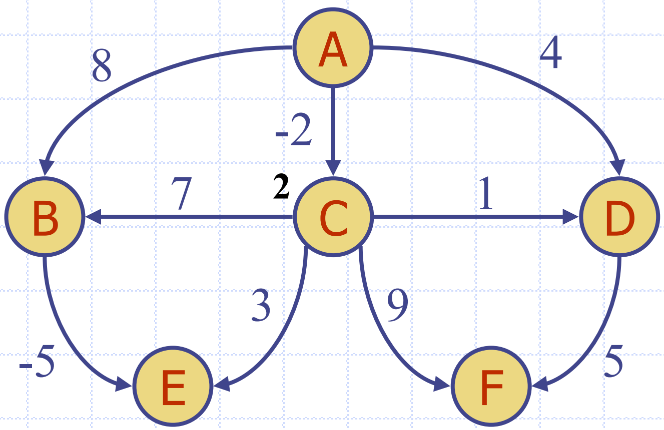
DAG-based Algorithm (not in book)

- Works even with negative-weight edges
- Uses topological order
- Doesn't use other data structures
- Is much faster than Dijkstra's algorithm
- Running time
  - $O(n+m)$

Algorithm *DagDistances*(G, s):
  for all $v \in G.vertices()$
    if $v = s$
      setDistance(v, 0)
    else
      setDistance(v, ∞)
  { Perform a topological sort of the vertices }
  for $u \leftarrow 1$ **to** n **do**   {in topological order}
    for each $e \in G.outEdges(u)$
      { relax edge e }
      $z \leftarrow G.opposite(u, e)$
      $r \leftarrow getDistance(u) + weight(e)$
      if $r < getDistance(z)$
        setDistance(z, r)

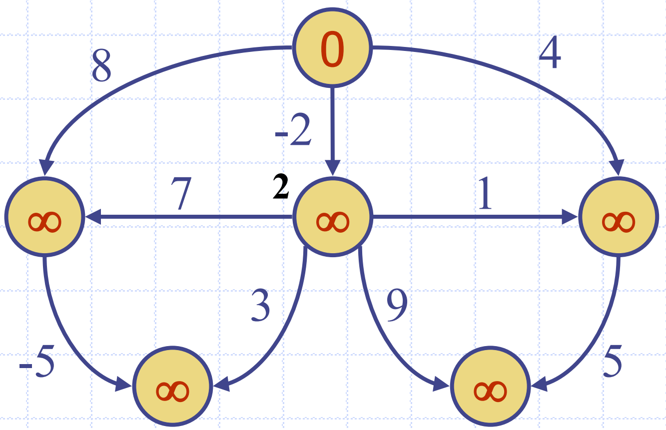
DAG Example

- Label nodes with initial distances
  - $d(v)$ values



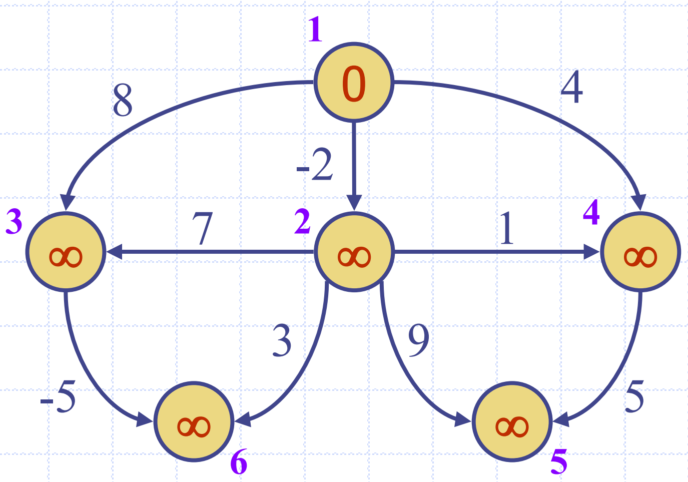
DAG Example

- Label nodes with initial distances
  - $d(v)$ values



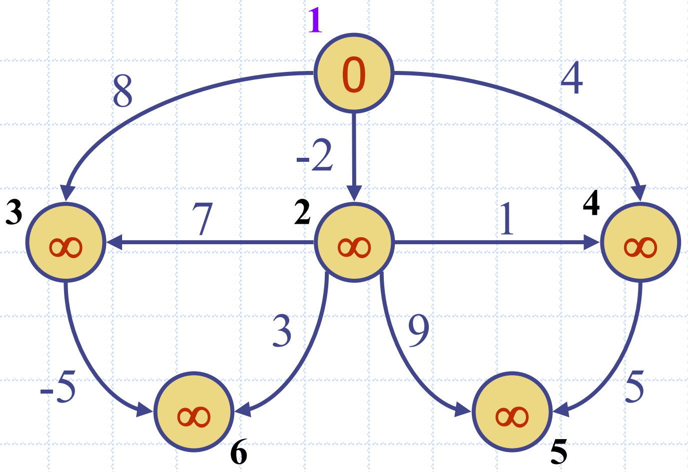
DAG Example

- Perform topological sort



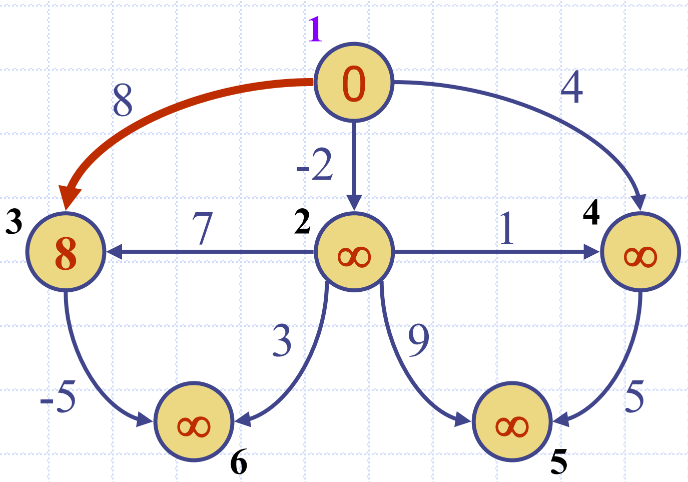
DAG Example

- Visit vertices in topological order
- Relax each edge for each vertex



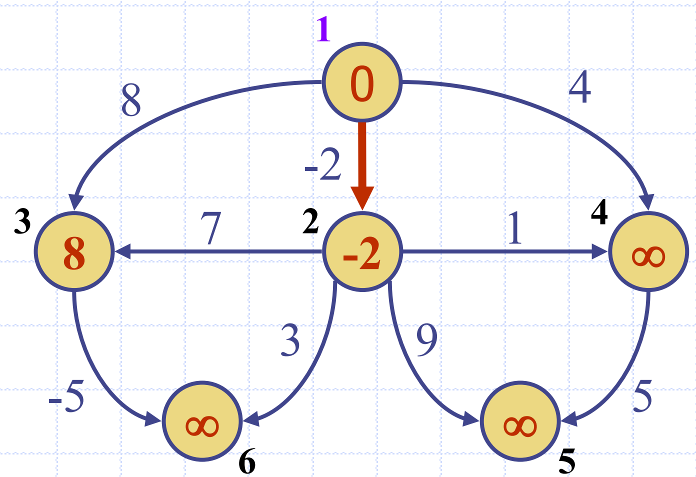
DAG Example

- Visit vertices in topological order
- Relax each edge for each vertex



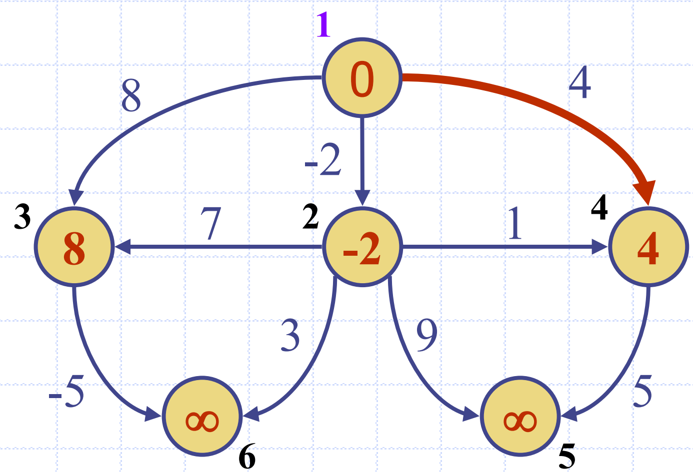
DAG Example

- Visit vertices in topological order
- Relax each edge for each vertex



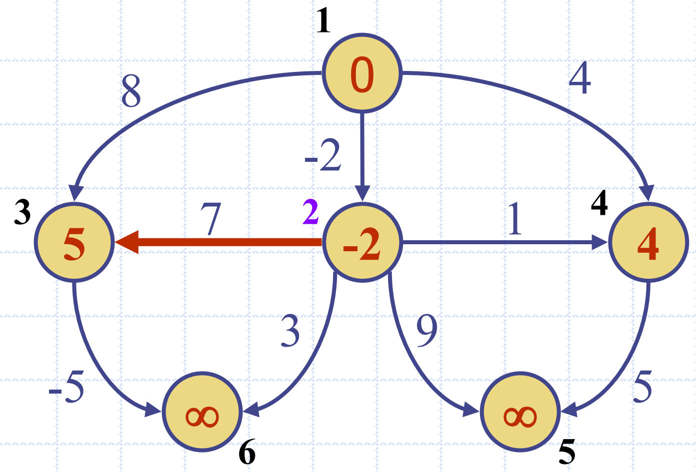
DAG Example

- Visit vertices in topological order
- Relax each edge for each vertex



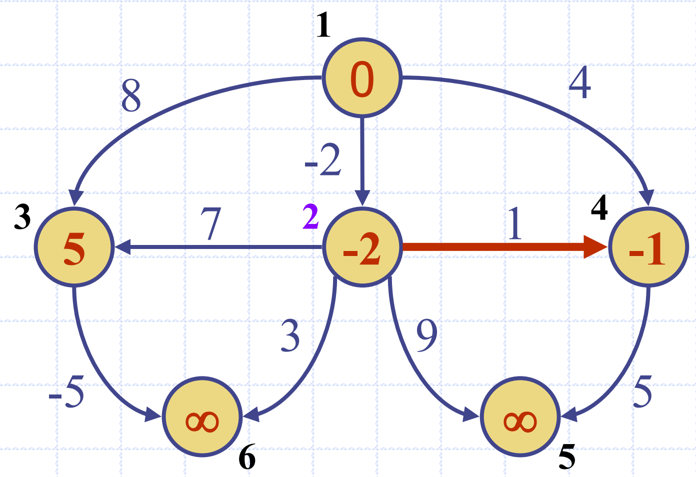
DAG Example

- Visit vertices in topological order
- Relax each edge for each vertex



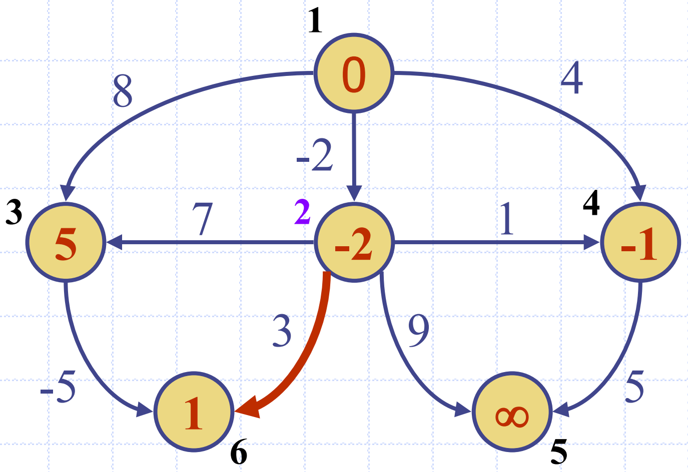
DAG Example

- Visit vertices in topological order
- Relax each edge for each vertex



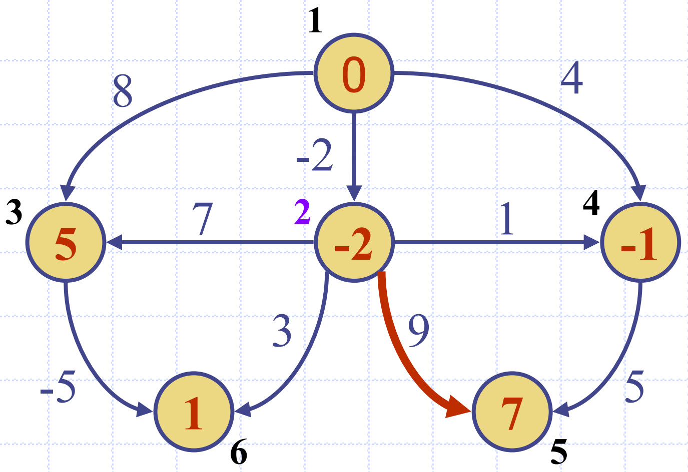
DAG Example

- Visit vertices in topological order
- Relax each edge for each vertex



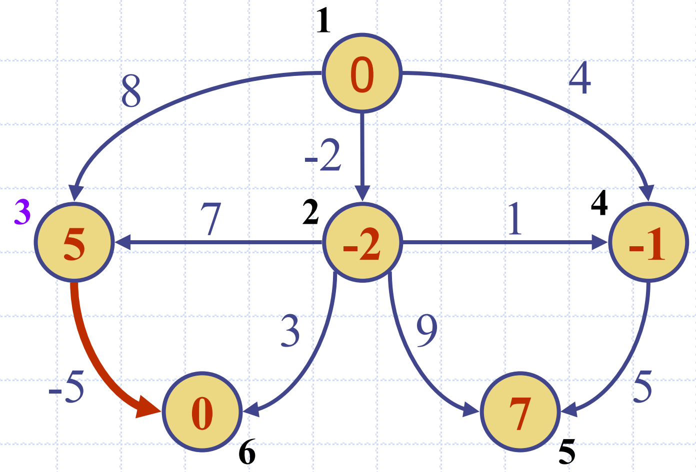
DAG Example

- Visit vertices in topological order
- Relax each edge for each vertex



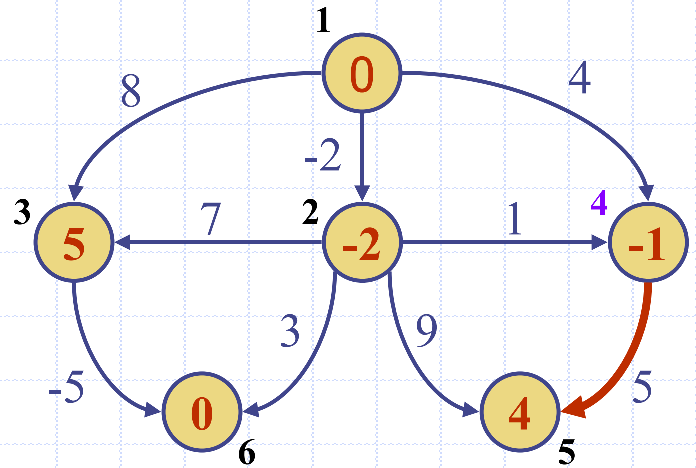
DAG Example

- Visit vertices in topological order
- Relax each edge for each vertex



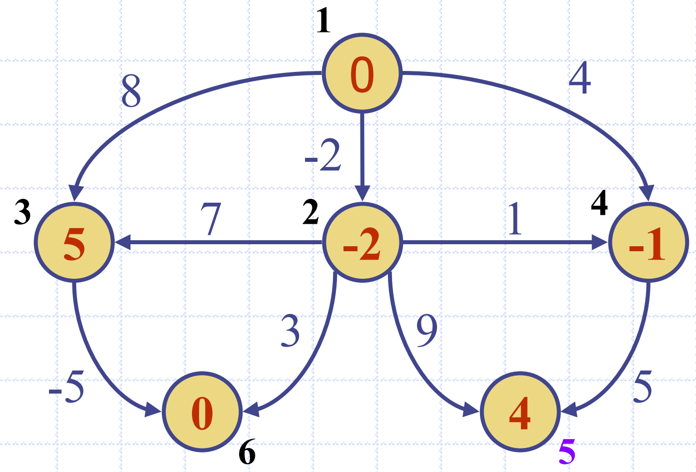
DAG Example

- Visit vertices in topological order
- Relax each edge for each vertex



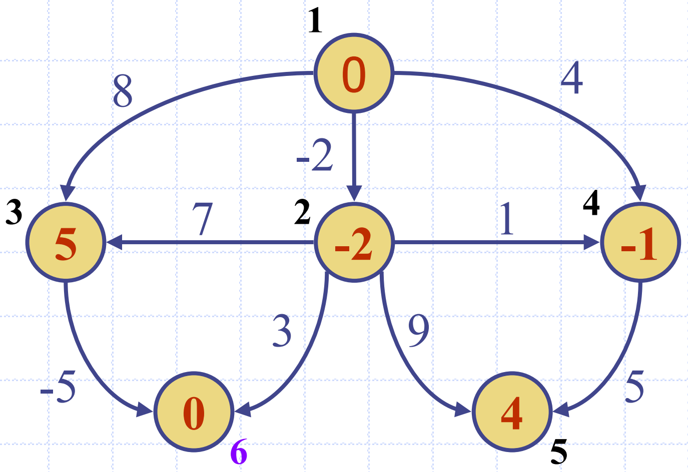
DAG Example

- Visit vertices in topological order
- Relax each edge for each vertex



DAG Example

- Visit vertices in topological order
- Relax each edge for each vertex



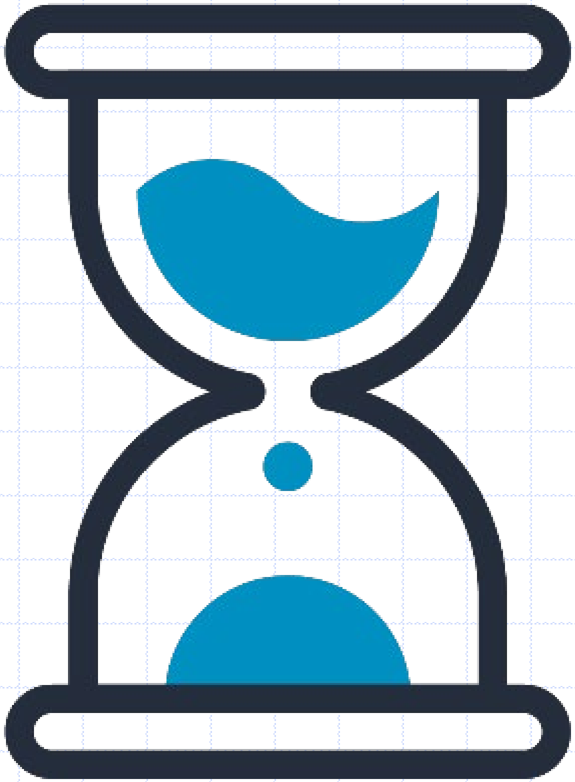
Java Implementation

```
/** Computes shortest-path distances from src vertex to all reachable vertices of g. */
public static <V> Map<Vertex<V>, Integer>
shortestPathLengths(Graph<V,Integer> g, Vertex<V> src) {
  // d.get(v) is upper bound on distance from src to v
  Map<Vertex<V>, Integer> d = new ProbeHashMap<>();
  // map reachable v to its d value
  Map<Vertex<V>, Integer> cloud = new ProbeHashMap<>();
  // pq will have vertices as elements, with d.get(v) as key
  AdaptablePriorityQueue<Integer, Vertex<V>> pq;
  pq = new HeapAdaptablePriorityQueue<>();
  // maps from vertex to its pq locator
  Map<Vertex<V>, Entry<Integer,Vertex<V>>> pqTokens;
  pqTokens = new ProbeHashMap<>();

  // for each vertex v of the graph, add an entry to the priority queue, with
  // the source having distance 0 and all others having infinite distance
  for (Vertex<V> v : g.vertices()) {
    if (v == src)
      d.put(v,0);
    else
      d.put(v, Integer.MAX_VALUE);
    pqTokens.put(v, pq.insert(d.get(v), v));        // save entry for future updates
  }
```

Java Implementation, 2

```
  // now begin adding reachable vertices to the cloud
  while (!pq.isEmpty()) {
    Entry<Integer, Vertex<V>> entry = pq.removeMin();
    int key = entry.getKey();
    Vertex<V> u = entry.getValue();
    cloud.put(u, key);                              // this is actual distance to u
    pqTokens.remove(u);                             // u is no longer in pq
    for (Edge<Integer> e : g.outgoingEdges(u)) {
      Vertex<V> v = g.opposite(u,e);
      if (cloud.get(v) == null) {
        // perform relaxation step on edge (u,v)
        int wgt = e.getElement();
        if (d.get(u) + wgt < d.get(v)) {            // better path to v?
          d.put(v, d.get(u) + wgt);                 // update the distance
          pq.replaceKey(pqTokens.get(v), d.get(v)); // update the pq entry
        }
      }
    }
  }
  return cloud;          // this only includes reachable vertices
}
```

10 minute break

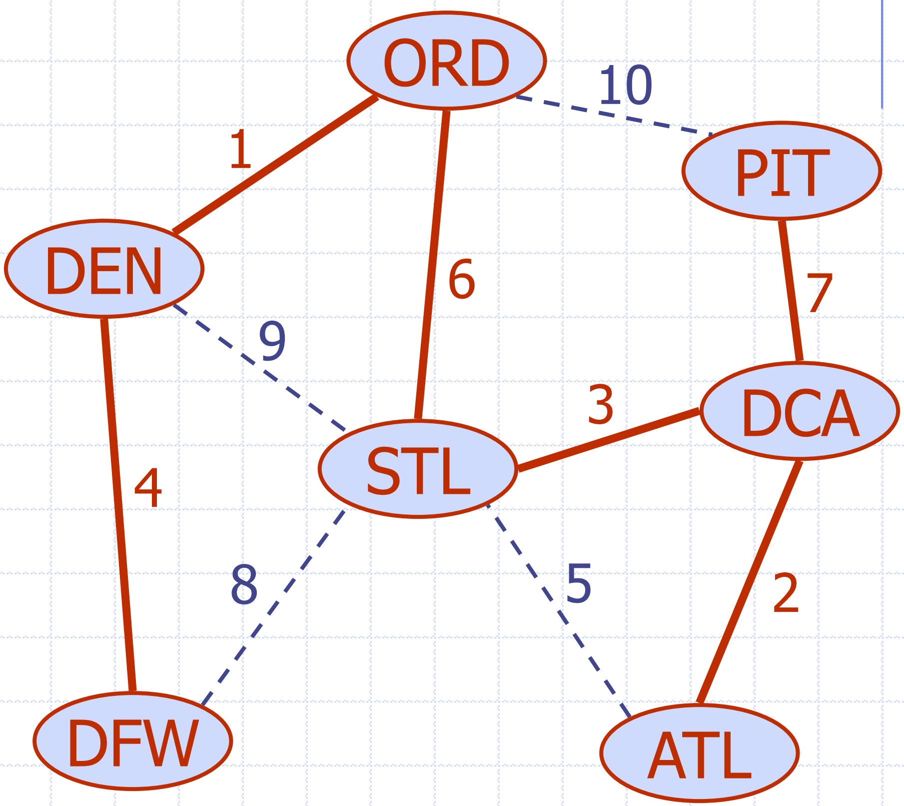


Week 10 – Graphs & Selection

1. Shortest path algorithms
2. Minimum spanning trees
3. Selection algorithms

Minimum Spanning Trees

- Spanning Subgraph
  - subgraph of a graph ***G*** containing all vertices of ***G***
- Spanning Tree
  - spanning subgraph that is itself a (free) tree
- Minimum Spanning Tree (MST)
  - spanning tree of a weighted graph with minimum total edge weight

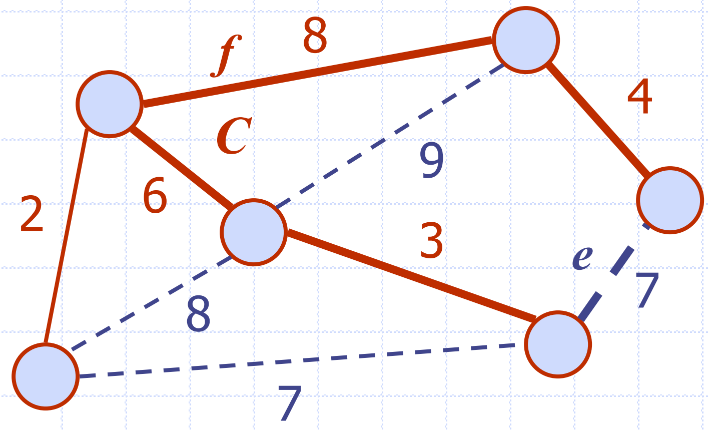


Cycle Property

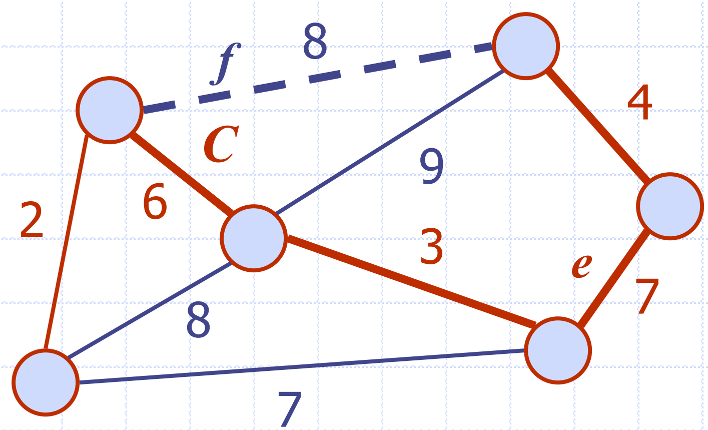
- Let T be a MST of a weighted graph G
- Let e be an edge of G that is not in T
- Let C be the cycle formed by e with T
- For every edge f of C, $weight(f) \leq weight(e)$

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

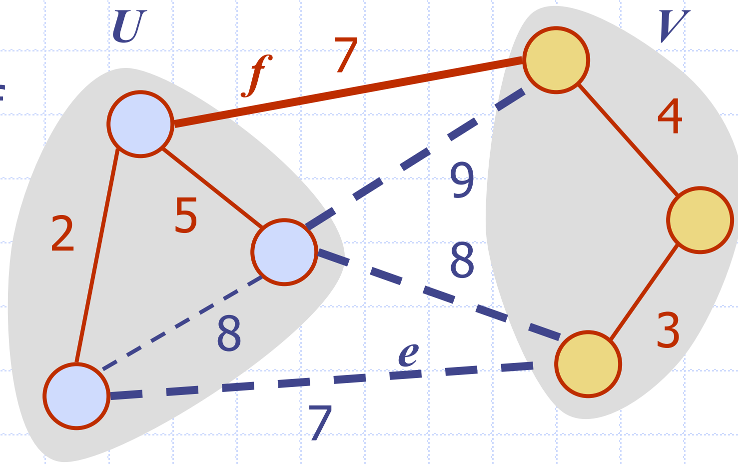


Partition Property

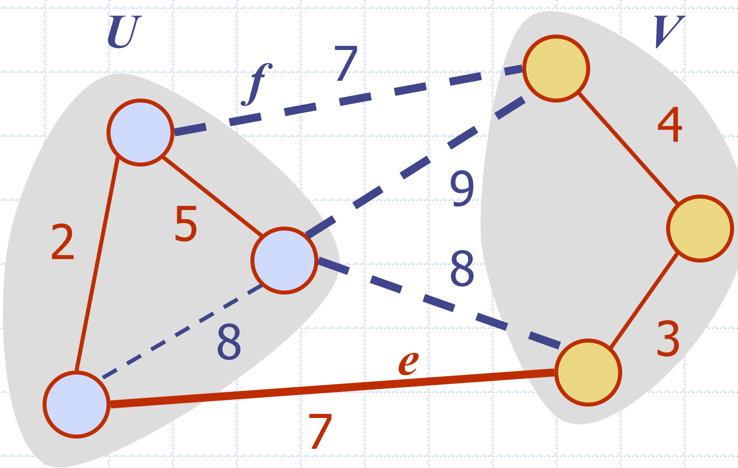
- Consider a partition of the vertices of G into subsets U and V
- Let e be an edge of minimum weight across the partition
- There is a minimum spanning tree of G containing edge e

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



Replacing f with e yields another MST



Prim-Jarnik's Algorithm

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

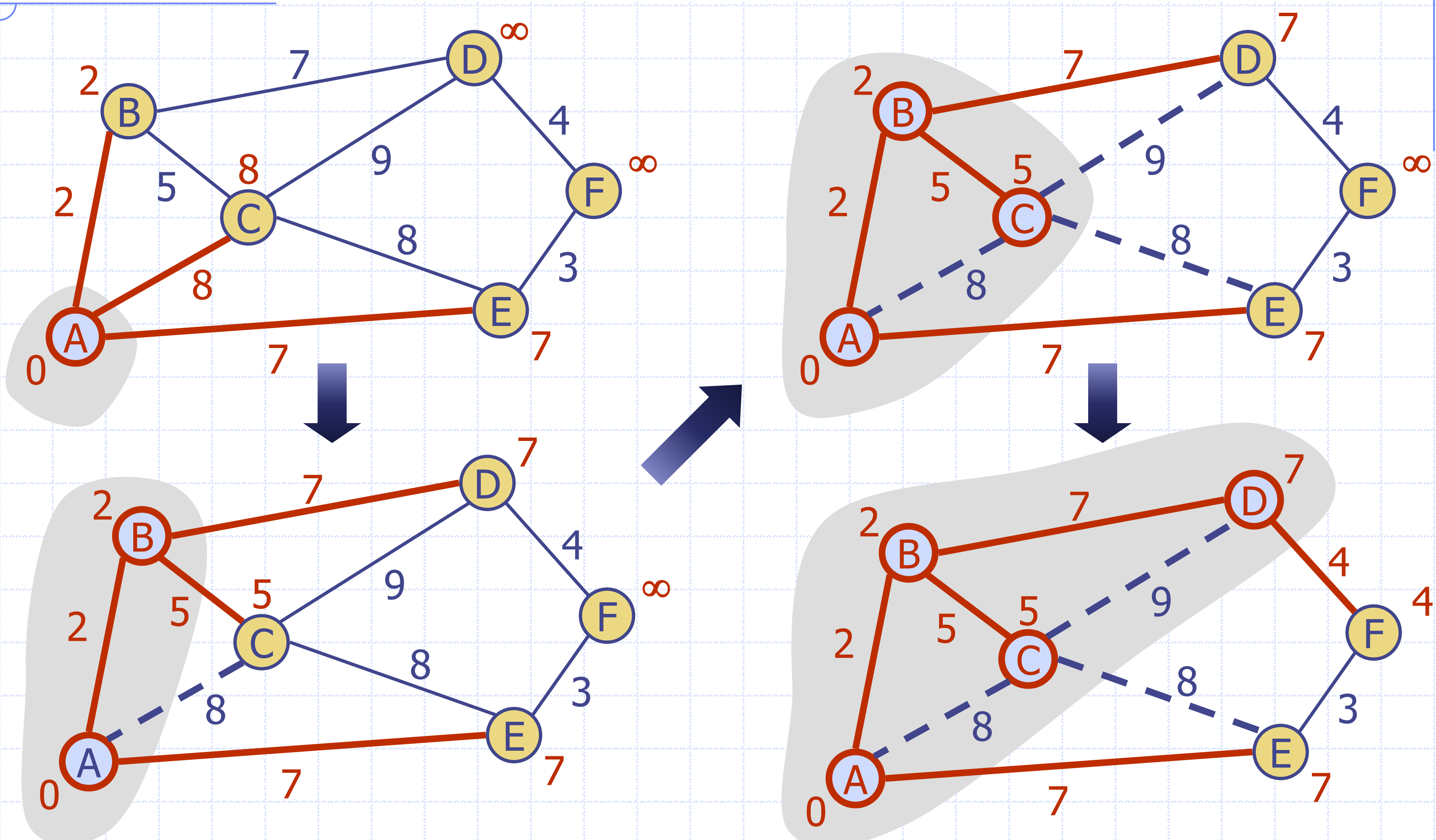
Prim-Jarnik Algorithm

Algorithm PrimJarnik(G):

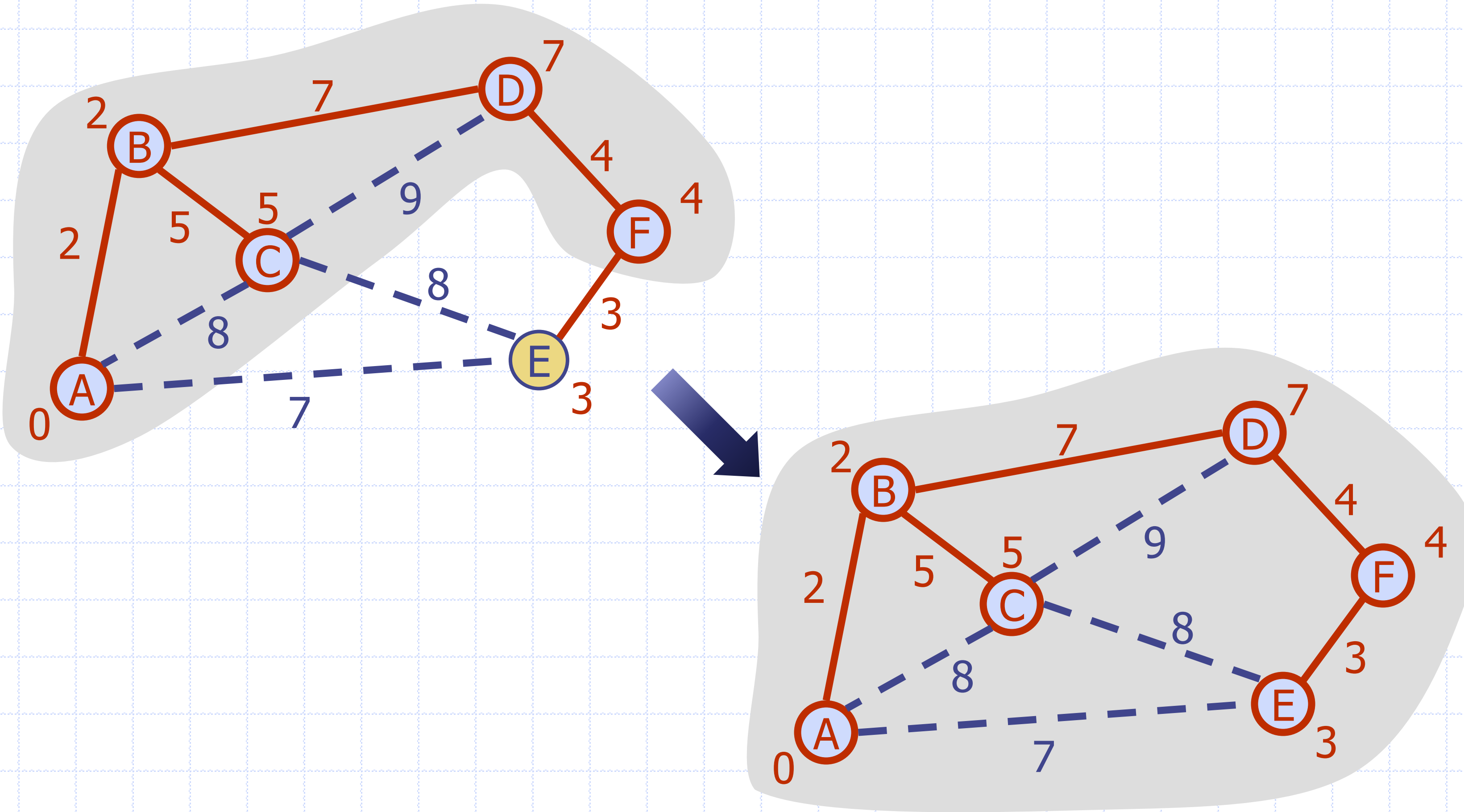
Input: An undirected, weighted, connected graph G with n vertices and m edges
Output: A minimum spanning tree T for G

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

Example



Example (contd.)



Analysis

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

Kruskal's Approach

- Maintain a partition of the vertices into clusters
  - initially, single-vertex clusters
  - keep an MST for each cluster
  - merge "closest" clusters and their MSTs
- Priority queue stores the edges outside clusters
  - key: weight
  - element: edge
- At the end of the algorithm
  - one cluster and one MST

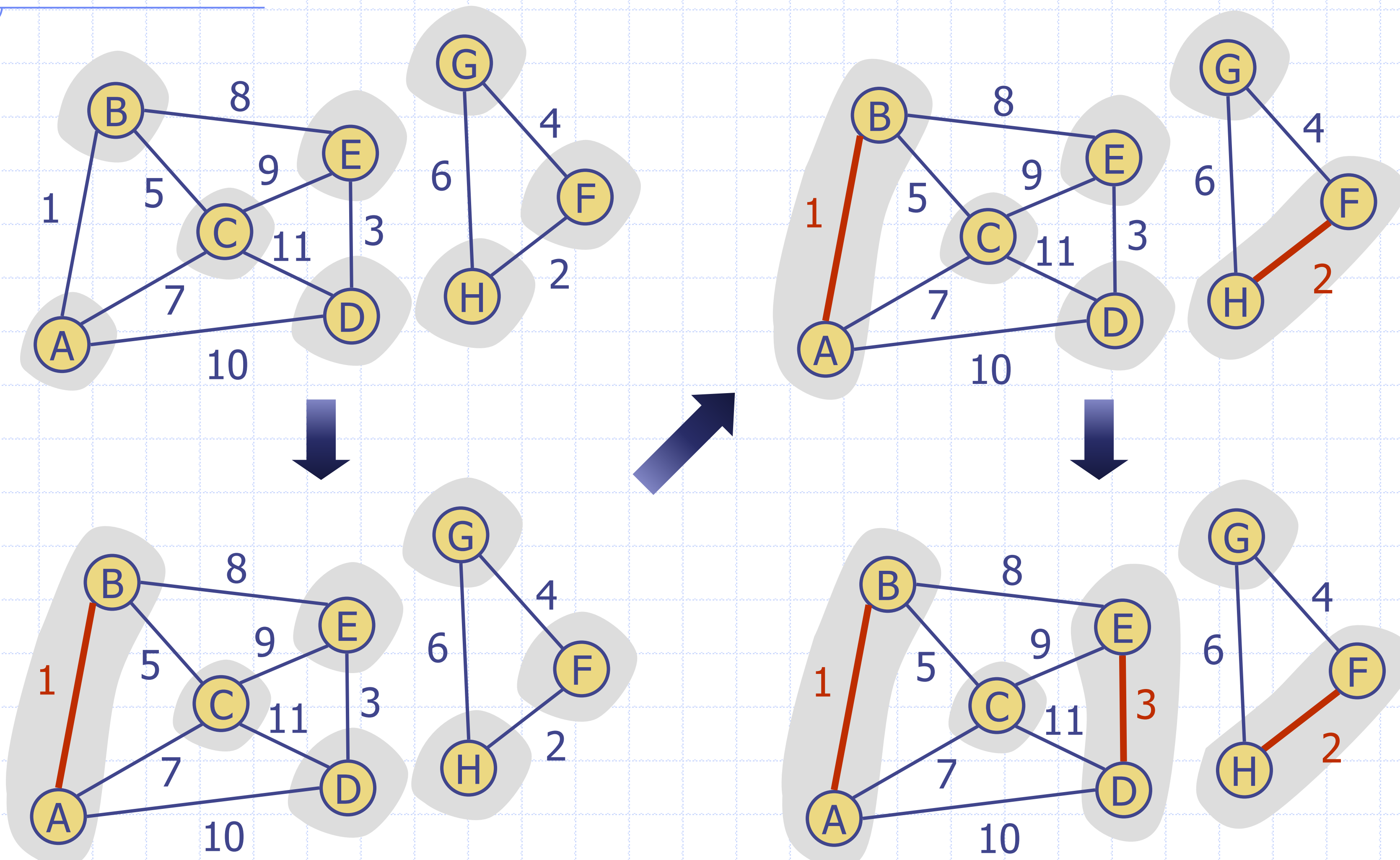
Kruskal's Algorithm

Algorithm Kruskal(G):

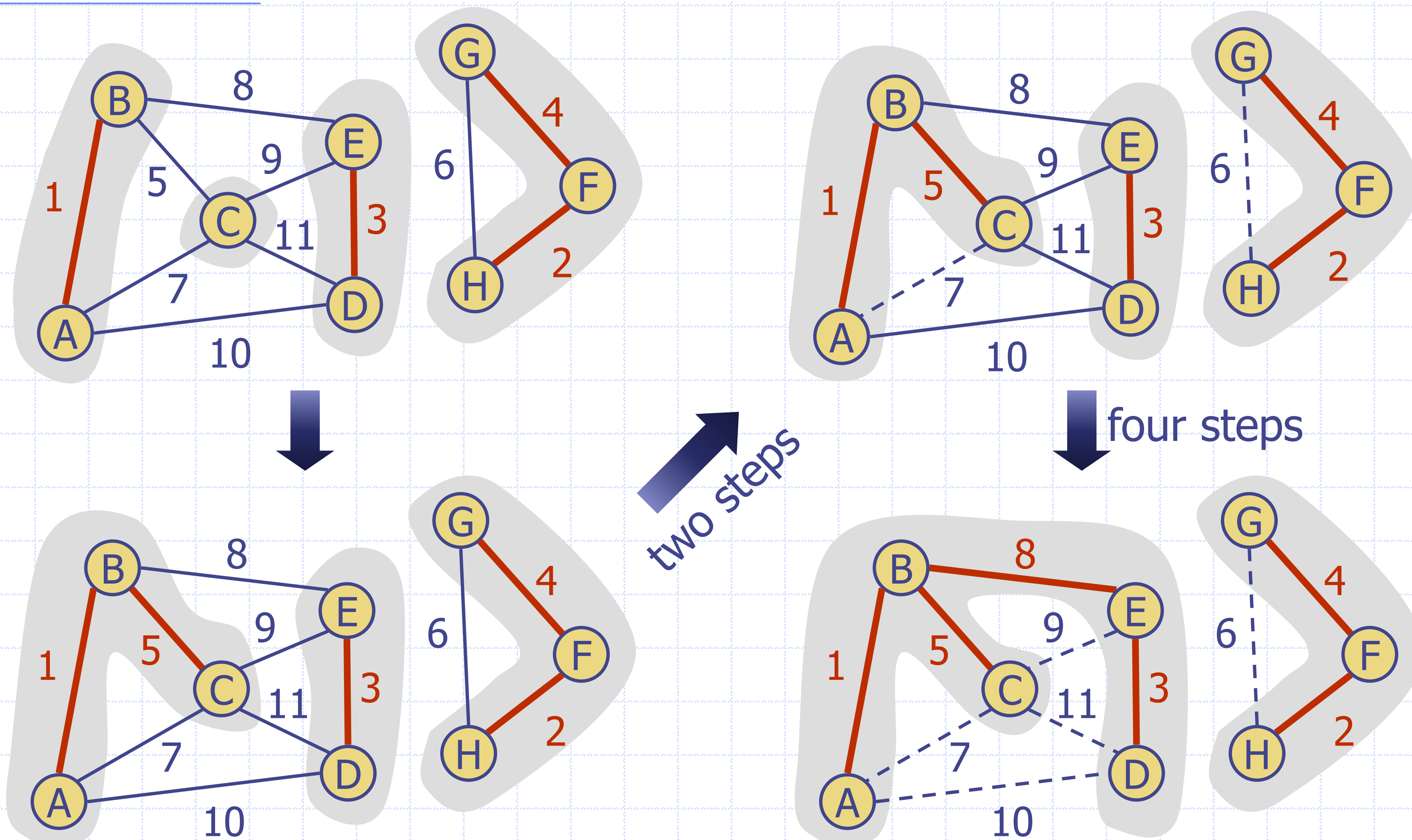
  Input: A simple connected weighted graph G with n vertices and m edges
  Output: A minimum spanning tree T for G

  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

Example



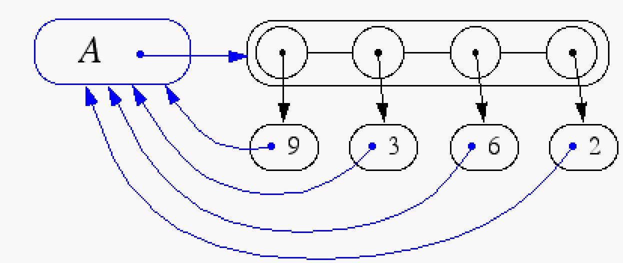
Example (contd.)



Data Structure for Kruskal's Algorithm

- Algorithm maintains a forest of trees
- Priority queue extracts the edges by increasing weight
- An edge is accepted if it connects distinct trees
- 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

List-based Partition



- Each set is stored in a sequence
- Each element has a reference back to the set
  - find(u) takes $\mathbf{O(1)}$ time, and returns the set of which u is a member
  - union(A, B) moves the elements of the smaller set to the sequence of the larger set and updates their references
    - takes min($|A|$, $|B|$) time
- Whenever an element is processed, it goes into a set of size at least double, hence each element is processed at most $\log n$ times

Partition-Based Implementation

- Partition-based version of Kruskal's Algorithm
  - cluster merges as unions
  - cluster locations as finds
- Running time $O((n + m) \log n)$
  - priority queue operations: $O(m \log n)$
  - union-find operations: $O(n \log n)$

Baruvka's Algorithm

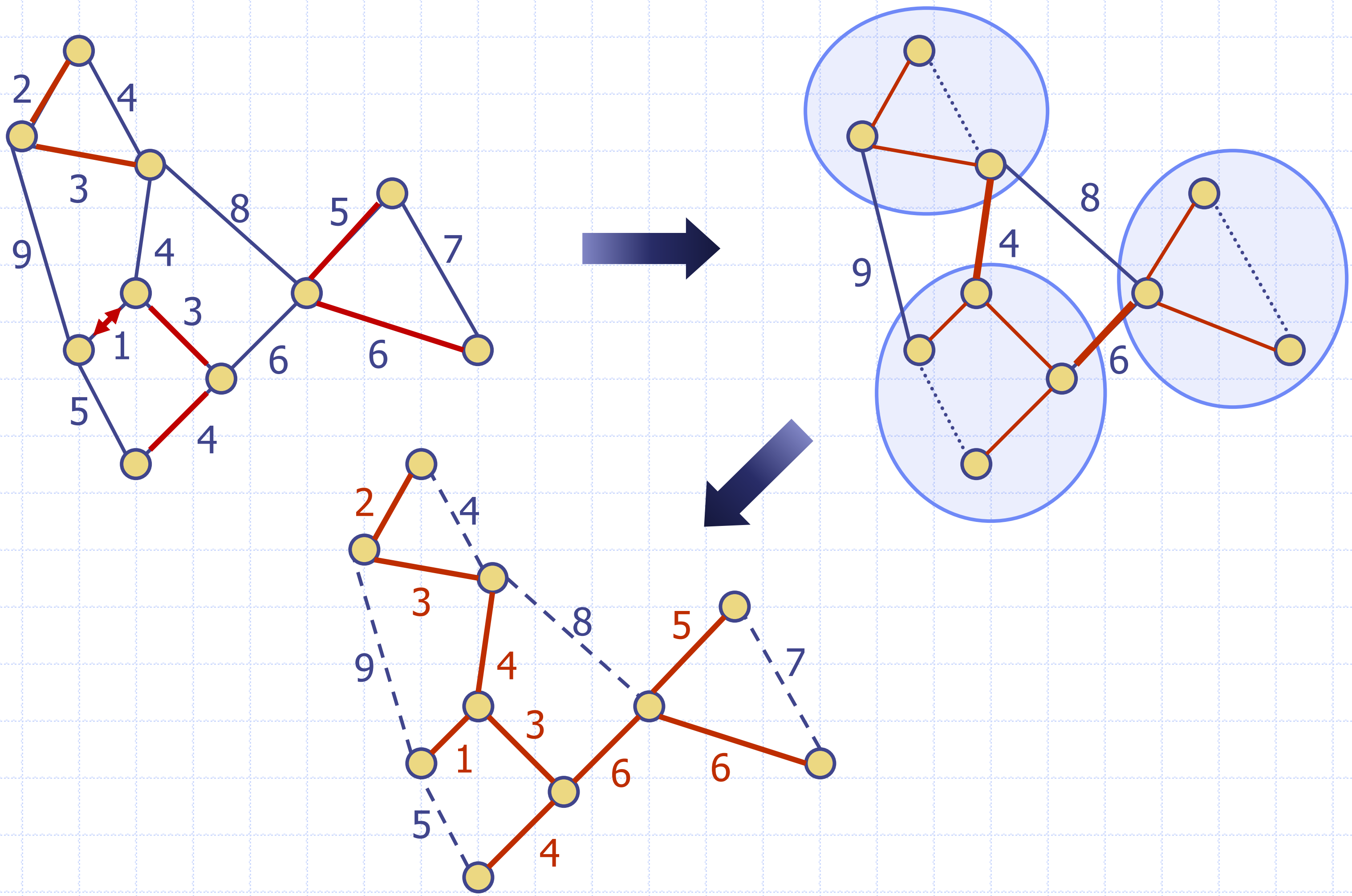
- Like Kruskal's Algorithm, Baruvka's algorithm grows many clusters at once and maintains a forest T
- Each iteration of the while loop halves the number of connected components in forest T
- Running time is $O(m \log n)$

Algorithm *BaruvkaMST*(G):

    $T \leftarrow V$ {just the vertices of G}
    while T has fewer than $n - 1$ edges **do**
        for each connected component C in T **do**
            Let edge e be the smallest-weight edge from C to another component in T
            if e is not already in T **then**
                Add edge e to T
    return T

Example of Baruvka's Algorithm

Slide by Matt Stallmann included with permission.



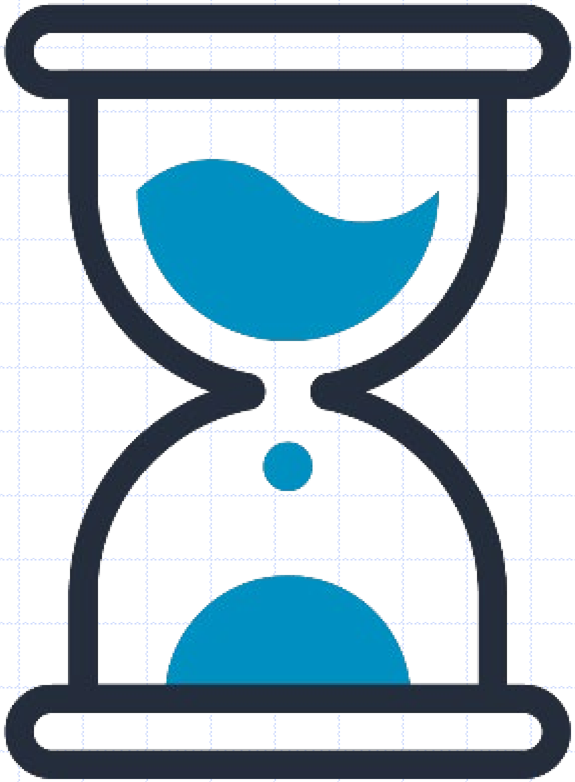
Java Implementation

```
/** Computes a minimum spanning tree of graph g using Kruskal's algorithm. */
public static <V> PositionalList<Edge<Integer>> MST(Graph<V,Integer> g) {
  // tree is where we will store result as it is computed
  PositionalList<Edge<Integer>> tree = new LinkedPositionalList<>();
  // pq entries are edges of graph, with weights as keys
  PriorityQueue<Integer, Edge<Integer>> pq = new HeapPriorityQueue<>();
  // union-find forest of components of the graph
  Partition<Vertex<V>> forest = new Partition<>();
  // map each vertex to the forest position
  Map<Vertex<V>,Position<Vertex<V>>> positions = new ProbeHashMap<>();

  for (Vertex<V> v : g.vertices())
    positions.put(v, forest.makeGroup(v));

  for (Edge<Integer> e : g.edges())
    pq.insert(e.getElement(), e);

```

Java Implementation, 2

```
    int size = g.numVertices( );
    // while tree not spanning and unprocessed edges remain...
    while (tree.size( ) != size − 1 && !pq.isEmpty( )) {
      Entry<Integer, Edge<Integer>> entry = pq.removeMin( );
      Edge<Integer> edge = entry.getValue( );
      Vertex<V>[ ] endpoints = g.endVertices(edge);
      Position<Vertex<V>> a = forest.find(positions.get(endpoints[0]));
      Position<Vertex<V>> b = forest.find(positions.get(endpoints[1]));
      if (a != b) {
        tree.addLast(edge);
        forest.union(a,b);
      }
    }

    return tree;
  }
```

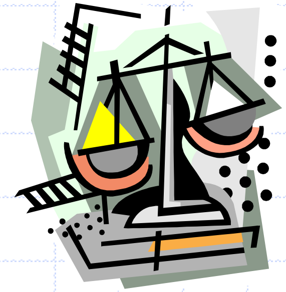
10 minute break



Week 10 – Graphs & Selection

1. Shortest path algorithms
2. Minimum spanning trees
3. Selection algorithms

Selection Problem



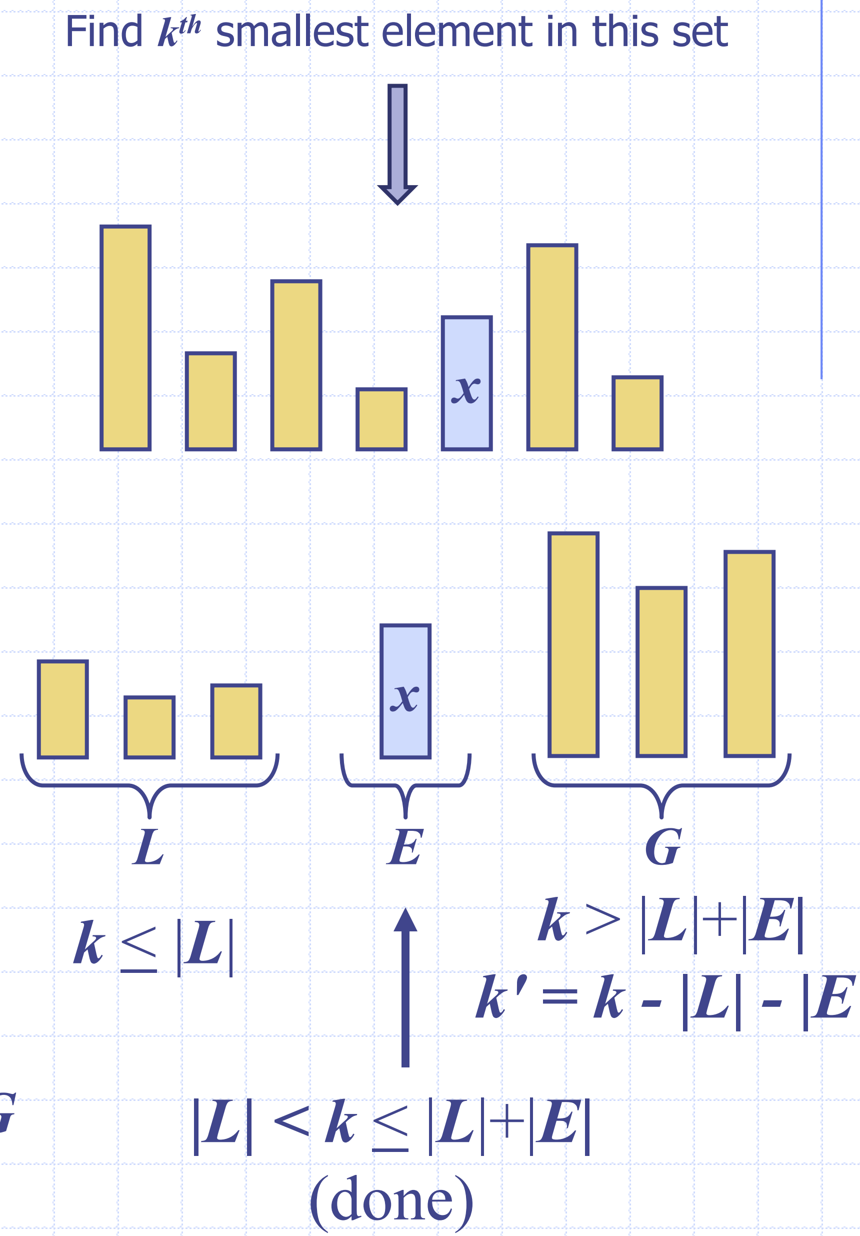
- Given an integer k and n elements $x_1, x_2, \ldots, x_n$, taken from a total order
  - find the k^{th} smallest element
- Of course, we can sort the set in $\mathbf{O}(n \log n)$ time and then index the k^{th} element

k=3 $\boxed{7\ 4\ 9\ 6\ 2 \rightarrow 2\ 4\ \underline{6}\ 7\ 9}$

- Can we solve the selection problem faster?

Quick Select

- Randomised selection algorithm based on the prune-and-search paradigm
  - Prune: pick a random element x (called pivot) and partition S into
    - L: elements less than x
    - E: elements equal x
    - G: elements greater than x
  - Search: depending on k, either answer is in E, or we need to recur in either L or G



Partition

- Partition input sequence as in quick sort
  - remove, in turn, each element y from S, and
  - insert y into L, E or G, depending on the result of the comparison with the pivot x
- Each insertion and removal is at the beginning or end of a sequence
  - hence takes $O(1)$ time
- Thus, partition step takes $O(n)$ time

```
Algorithm partition(S, p)
    Input sequence S, position p of pivot
    Output subsequences L, E, G of the
        elements of S less than, equal to,
        or greater than the pivot.

    L, E, G ← empty sequences
    x ← S.remove(p)
    while ¬S.isEmpty()
        y ← S.remove(S.first())
        if y < x
            L.addLast(y)
        else if y = x
            E.addLast(y)
        else { y > x }
            G.addLast(y)
    return L, E, G
```

Quick-Select Visualisation

- Execution of quick-select can be visualised by a recursion path
  - Each node represents a recursive call of quick-select, and stores k and the remaining sequence

k=5, S=(7 4 9 3 2 6 5 1 8)

Quick-Select Visualisation

- Execution of quick-select can be visualised by a recursion path
  - Each node represents a recursive call of quick-select, and stores k and the remaining sequence

k=5, S=(7 4 9 <u>3</u> 2 6 5 1 8) → (2 1 <u>3</u> 7 4 9 6 5 8)

Quick-Select Visualisation

- Execution of quick-select can be visualised by a recursion path
  - Each node represents a recursive call of quick-select, and stores k and the remaining sequence

k=5, S=(7 4 9 <u>3</u> 2 6 5 1 8) → (2 1 <u>3</u> 7 4 9 6 5 8)

k=2, S=(7 4 9 6 5 8)

k index value has changed, now that the Less and Equal partitions, which had three elements, have been removed

Quick-Select Visualisation

- Execution of quick-select can be visualised by a recursion path
  - Each node represents a recursive call of quick-select, and stores *k* and the remaining sequence

k=5, S=(7 4 9 3 2 6 5 1 8) → (2 1 3 7 4 9 6 5 8)

k=2, S=(7 4 9 6 5 8) → (7 4 6 5 8 9)

Quick-Select Visualisation

- Execution of quick-select can be visualised by a recursion path
  - Each node represents a recursive call of quick-select, and stores k and the remaining sequence

k=5, S=(7 4 9 3 2 6 5 1 8) → (2 1 3 7 4 9 6 5 8)

k=2, S=(7 4 9 6 5 8) → (7 4 6 5 8 9)

k=2, S=(7 4 6 5)

k index value does not change when the Greater partition is removed

Quick-Select Visualisation

- Execution of quick-select can be visualised by a recursion path
  - Each node represents a recursive call of quick-select, and stores k and the remaining sequence

$$k=5,\ S=(7\ 4\ 9\ \underline{3}\ 2\ 6\ 5\ 1\ 8) \rightarrow (2\ 1\ \underline{3}\ 7\ 4\ 9\ 6\ 5\ 8)$$

$$k=2,\ S=(7\ 4\ 9\ 6\ 5\ \underline{8}) \rightarrow (7\ 4\ 6\ 5\ \underline{8}\ 6)$$

$$k=2,\ S=(7\ \underline{4}\ 6\ 5) \rightarrow (\underline{4}\ 7\ 6\ 5)$$

Quick-Select Visualisation

- Execution of quick-select can be visualised by a recursion path
  - Each node represents a recursive call of quick-select, and stores *k* and the remaining sequence

k=5, S=(7 4 9 $\underline{3}$ 2 6 5 1 8) → (2 1 $\underline{3}$ 7 4 9 6 5 8)

k=2, S=(7 4 9 6 5 $\underline{8}$) → (7 4 6 5 $\underline{8}$ 6)

k=2, S=(7 $\underline{4}$ 6 5) → ($\underline{4}$ 7 6 5)

k=1, S=(7 6 5)

Quick-Select Visualisation

- Execution of quick-select can be visualised by a recursion path
  - Each node represents a recursive call of quick-select, and stores k and the remaining sequence

k=5, S=(7 4 9 3 2 6 5 1 8) → (2 1 3 7 4 9 6 5 8)

k=2, S=(7 4 9 6 5 8) → (7 4 6 5 8 6)

k=2, S=(7 4 6 5) → (4 7 6 5)

k=1, S=(7 6 5) → (5 7 6)

Quick-Select Visualisation

- Execution of quick-select can be visualised by a recursion path
  - Each node represents a recursive call of quick-select, and stores k and the remaining sequence

k=5, S=(7 4 9 3 2 6 5 1 8) → (2 1 3 7 4 9 6 5 8)

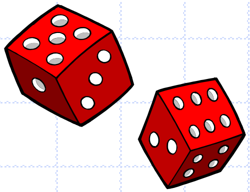
k=2, S=(7 4 9 6 5 8) → (7 4 6 5 8 6)

k=2, S=(7 4 6 5) → (4 7 6 5)

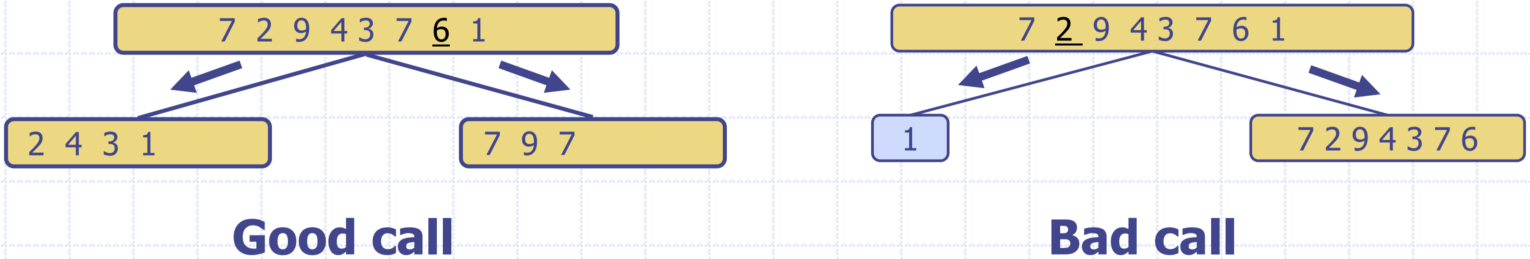
k=1, S=(7 6 5) → (5 7 6)

5

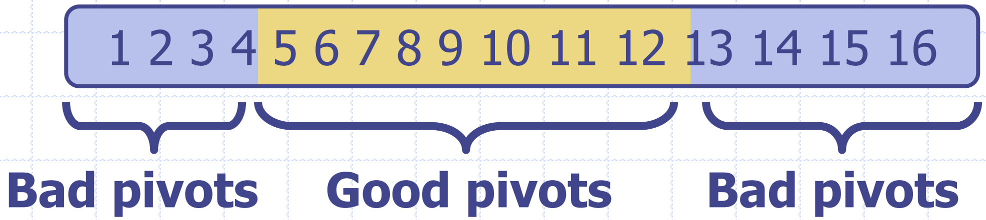
Expected Running Time



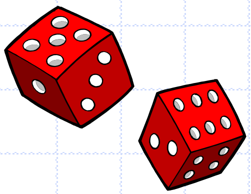
- Consider a recursive call of quick sort on a sequence of size s
  - **Good call:** the sizes of L and G are each less than $3s \div 4$
  - **Bad call:** one of L and G has size greater than $3s \div 4$



- Good calls have a probability of $1/2$
  - ½ of the possible pivots cause good calls

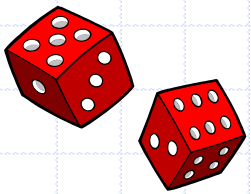


Expected Running Time, Part 2



- Probabilistic Fact #1: Expected number of coin tosses required in order to get one head is two
- Probabilistic Fact #2: Expectation is a linear function
  - $E(X+Y) = E(X) + E(Y)$
  - $E(cX) = cE(X)$

Expected Running Time, Part 2



- Let $T(n)$ denote the expected running time of quick select
- By Fact #2
  - $T(n) \leq T(3n/4) + bn\cdot$(expected # of calls before a good call)
- By Fact #1
  - $T(n) \leq T(3n/4) + 2bn$
- That is, $T(n)$ is a geometric series
  - $T(n) \leq 2bn + 2b(3/4)n + 2b(3/4)^2 n + 2b(3/4)^3 n + \ldots$
- So $T(n)$ is $\mathbf{O}(n)$
  - can solve selection problem in $\mathbf{O}(n)$ expected time

Worst Case Running Time?

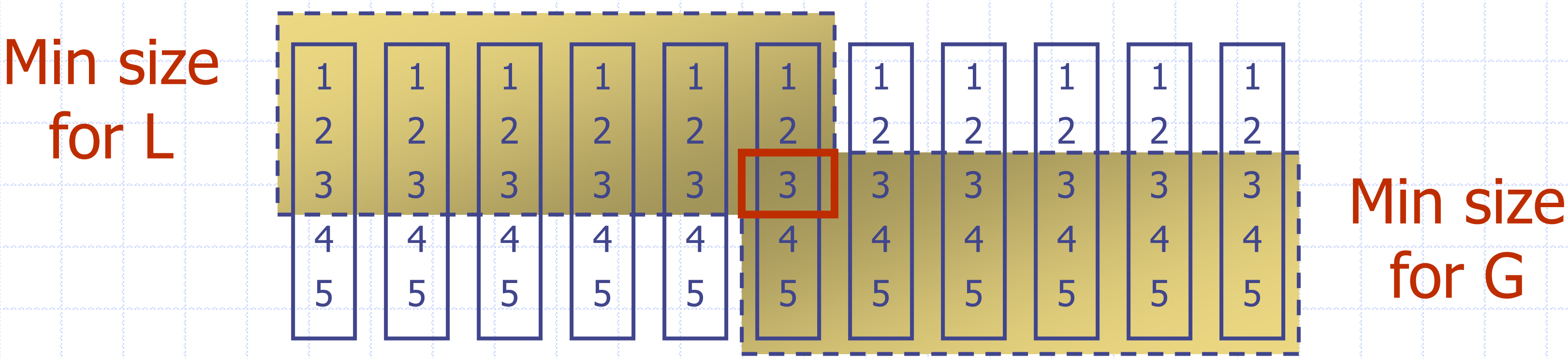
- Remember quick sort
  - $\mathbf{O}(n^2)$ if pivot is always a bad choice

Deterministic Selection

- Recursively use the selection algorithm to find a good pivot for quick select
  1. divide S into $n \div 5$ sets of 5 each
     - last set may have less than 5 elements
  2. find a median in each set
     - use insertion sort on small sets of 5 and index into median
  3. recursively use selection algorithm to find median of the $n \div 5$ medians ("baby" medians)
  4. partition around this median of medians
     - guaranteed to be a good pivot
  5. k^{th} item found or recurse on lower or greater partition

Deterministic Selection

- Steps 1, 2 & 4 take $\mathbf{O}(n)$ time
  - step 2 calls insertion sort $\mathbf{O}(n)$ times on sets of size $\mathbf{O}(1)$
- Step 3 takes $T(n \div 5)$ time
- Step 5 is a reduction of n, by greater than 2
- Thus, $\mathbf{O}(n)$ worst-case time



Further Reading

- Data Structures and Algorithms in Java
  - Chapter 12.5
  - Chapter 14.6, 14.7
- Introduction to Algorithms
  - Chapter 9.2, 9.3
  - Chapter 23
  - Chapter 24

Reminders

- Homework task 4 due at 5:00pm tomorrow!
  - The final one!
- Midsemester break next week
  - No lectures or tutorials
- Upcoming lecture content

Week	Lecture Content
Week 11	COMP7505 Presentations
Week 12	COMP7505 Presentations
Week 13	COMP7505 Presentations/Exam Q&A

COMP7505 Presentations

- COMP7505 presentations will be presented during the week 11 to 13 lecture timeslot.
- It is expected that anyone presenting is present for the full timeslot.
- Attendance is not mandatory for COMP3506 students, however:
  - These presentations may not be recorded
  - The content presented will benefit you greatly for any technical interviews you may complete in the future