ขั้นตอนวิธีกราฟ
Graph Algorithms

การอบรมคอมพิวเตอร์โอลิมปิก สอวน. ค่ายที่ 2 ปีการศึกษา 2563

อ.ดร.ฐาปนา บุญชู

สาขาวิชาวิทยาการคอมพิวเตอร์ คณะวิทยาศาสตร์และเทคโนโลยี

มหาวิทยาลัยธรรมศาสตร์

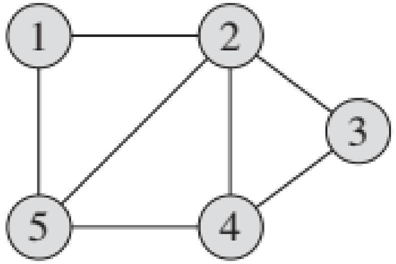
Graph Fundamentals

Representing a Graph

- We can choose between two standard ways to represent a graph $G = (V, E)$ as a collection of adjacency lists or as an adjacency matrix.
- Because the adjacency-list representation provides a compact way to represent sparse graphs—those for which $|E|$ is much less than $|V|^2$—it is usually the method of choice.
- We may prefer an adjacency-matrix representation, however, when the graph is dense—$|E|$ is close to $|V|^2$—or when we need to be able to tell quickly if there is an edge connecting two given vertices.

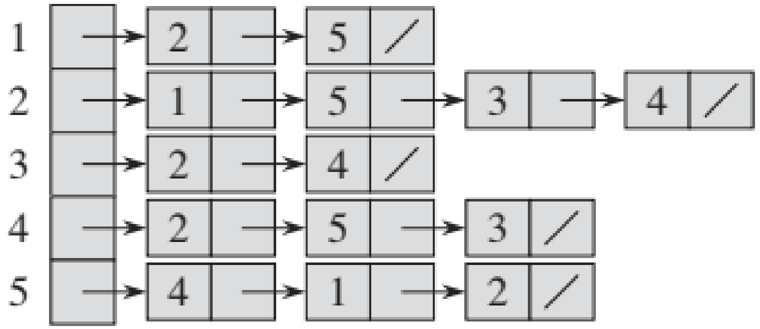
Representing a Graph

Graph



(a)

Represented using Adjacency Lists.



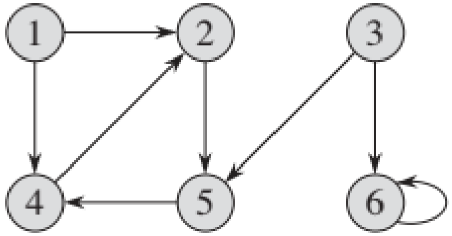
(b)

Represented using Adjacency Matrix.

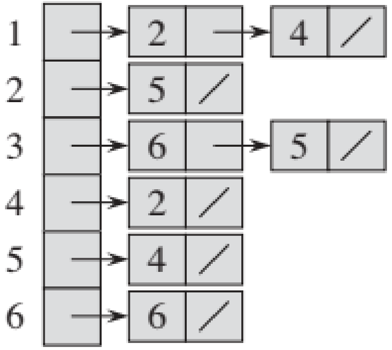
	1	2	3	4	5
1	0	1	0	0	1
2	1	0	1	1	1
3	0	1	0	1	0
4	0	1	1	0	1
5	1	1	0	1	0

(c)

Undirected graph



(a)



(b)

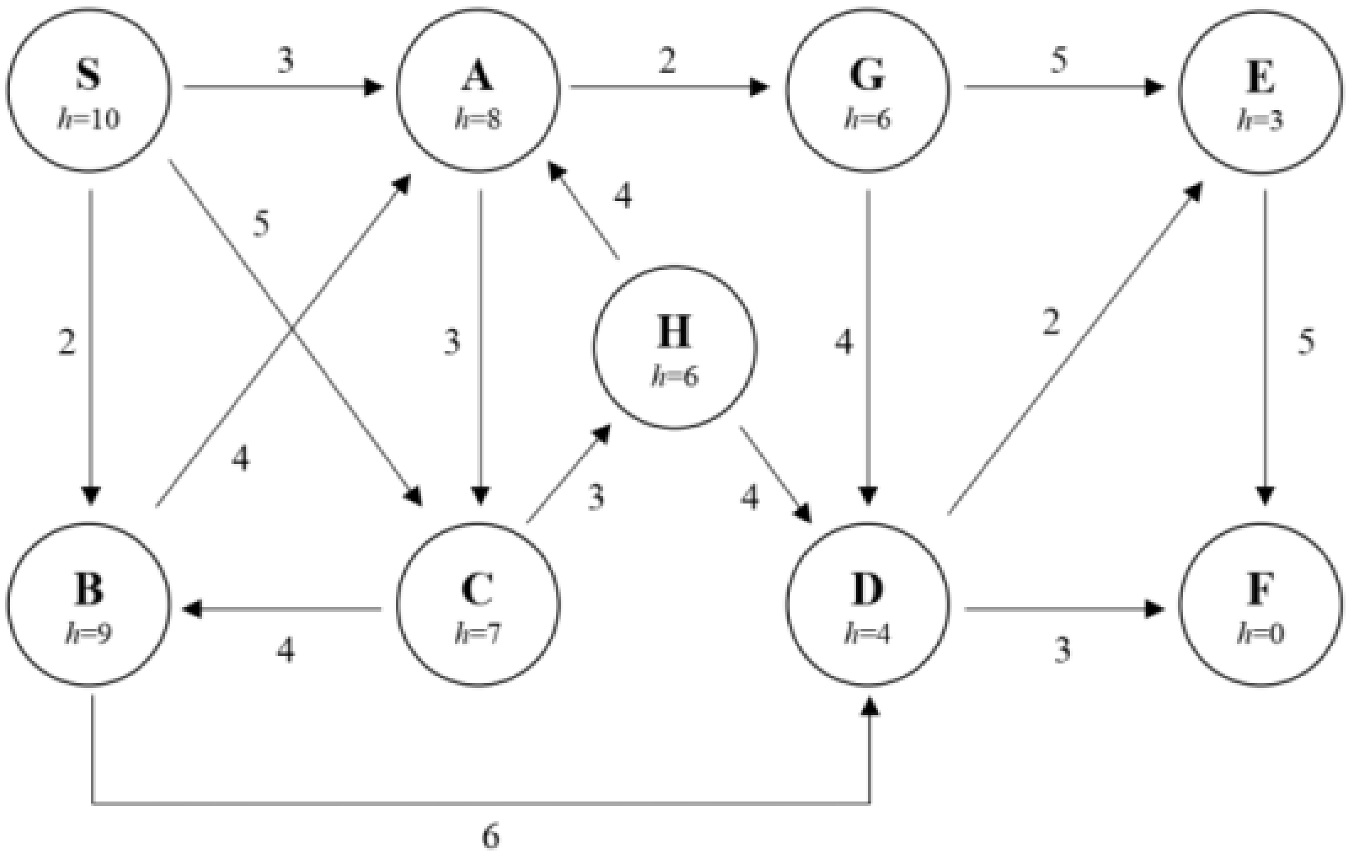
	1	2	3	4	5	6
1	0	1	0	1	0	0
2	0	0	0	0	1	0
3	0	0	0	0	1	1
4	0	1	0	0	0	0
5	0	0	0	1	0	0
6	0	0	0	0	0	1

(c)

Directed graph

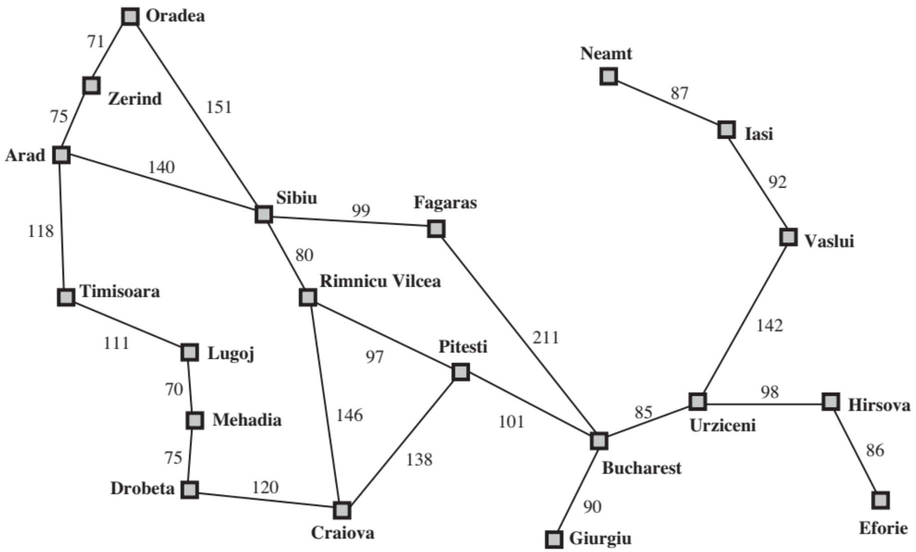
Exercise 1

- ให้นักเรียนเขียนโปรแกรมเพื่อ**แทนค่า**กราฟต่อไปนี้ด้วยวิธี Adjacency Matrix



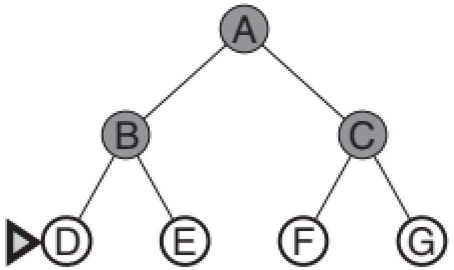
Problem of Graph Searching

- The problem is to find the solution for the **goal** on a given graph.
- A **solution** is an action sequence, so search algorithms work by considering various possible action sequences.
- **Solution quality** is measured by the path cost function, and an **optimal solution** has the lowest path cost among all solutions.



Uninformed Search Strategies
(Blind Search)

Breadth-First Search (BFS)

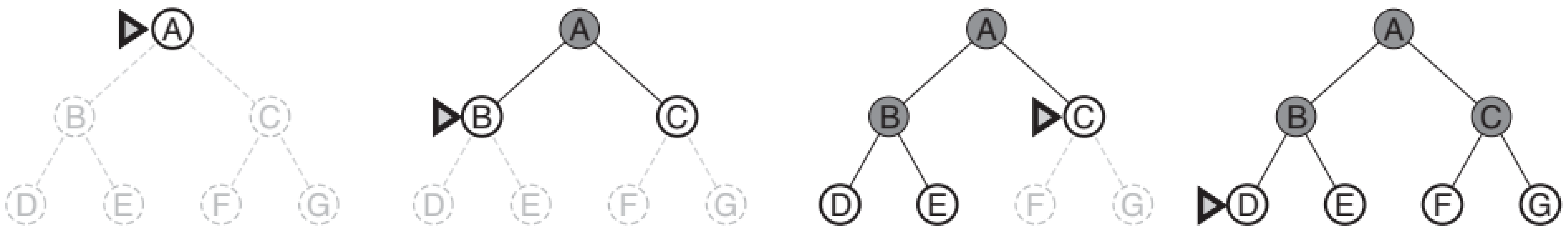


- Root is expanded first.
- Then, all the *successors* of the root node are expanded next, then *their successors*, and so on.
- In general, all the nodes are expanded at a **given depth** in the search tree **before** any nodes at the next level are expanded.
- BFS is achieved very simply by using a **FIFO queue** for the *frontier*.
  - Thus, new nodes (which are always deeper than their parents) go to the back of the queue, and **old nodes**, which are shallower than the new nodes, **get expanded first**.
- The algorithm, following the general template for graph search, discards any new path to a state already in the frontier or explored set; it is easy to see that any such path must be at least as deep as the one already found.
- There is one **slight tweak** on the general graph-search algorithm, which is that **the goal test is applied to each node when it is generated** rather than when it is selected for expansion.
  - Thus, BFS always has the shallowest path to every node on the frontier.

Breadth-First Search (BFS)

- Four criteria:
  - BFS is *complete*.
    - As soon as a goal node is generated, we know it is the shallowest goal node because all shallower nodes must have been generated already and failed the goal test.
  - Now, the shallowest goal node is not *necessarily* the *optimal* one; technically, breadth-first search is ***optimal*** if the path cost is a nondecreasing function of the depth of the node.
  - Complexity, both space and time complexity are $O(b^d)$.
- The memory requirements are a bigger problem for breadth-first search than is the execution time.
  - One might wait 13 days for the solution to an important problem with search depth 12, but no personal computer has the petabyte of memory it would take.
- In general, exponential-complexity search problems cannot be solved by uninformed methods for any but the smallest instances.

BFS



Depth	Nodes	Time	Memory
2	110	.11 milliseconds	107 kilobytes
4	11,110	11 milliseconds	10.6 megabytes
6	10^6	1.1 seconds	1 gigabyte
8	10^8	2 minutes	103 gigabytes
10	10^{10}	3 hours	10 terabytes
12	10^{12}	13 days	1 petabyte
14	10^{14}	3.5 years	99 petabytes
16	10^{16}	350 years	10 exabytes

Time and memory requirements for breadth-first search. The numbers shown assume branching factor $b = 10$; 1 million nodes/second; 1000 bytes/node.

Breadth-First Search (BFS)

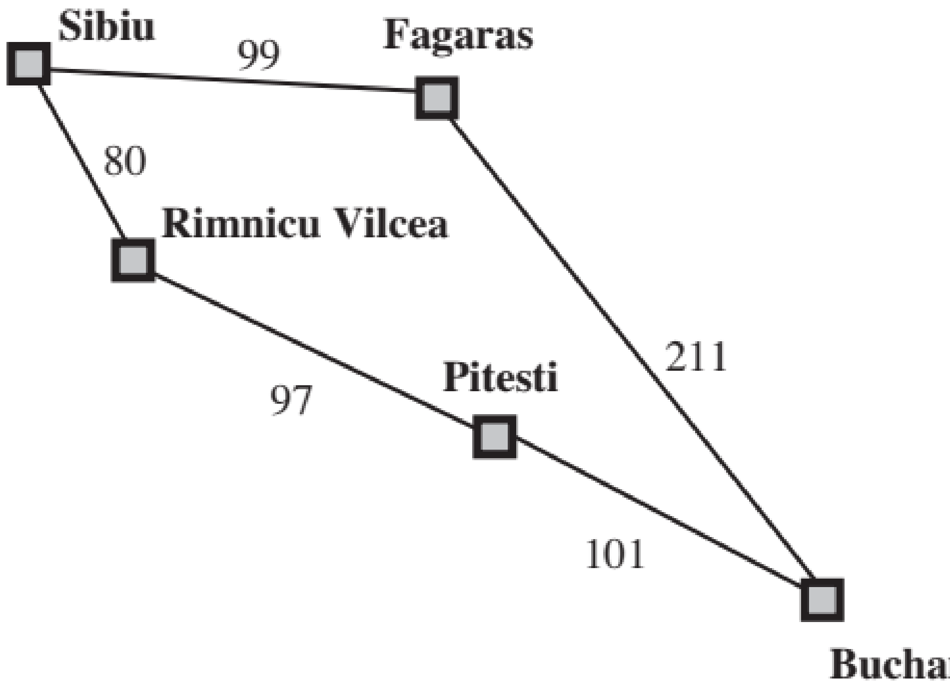
```
function BREADTH-FIRST-SEARCH(problem) returns a solution, or failure
    node ← a node with STATE = problem.INITIAL-STATE, PATH-COST = 0
    if problem.GOAL-TEST(node.STATE) then return SOLUTION(node)
    frontier ← a FIFO queue with node as the only element
    explored ← an empty set
    loop do
        if EMPTY?(frontier) then return failure
        node ← POP(frontier)   /* chooses the shallowest node in frontier */
        add node.STATE to explored
        for each action in problem.ACTIONS(node.STATE) do
            child ← CHILD-NODE(problem, node, action)
            if child.STATE is not in explored or frontier then
                if problem.GOAL-TEST(child.STATE) then return SOLUTION(child)
                frontier ← INSERT(child, frontier)
```

Uniform-cost search

- When **all step costs are equal**, breadth-first search is *optimal* because it always expands the *shallowest unexpanded* node.
- By a simple extension, we can find an algorithm that is optimal with any **step-cost** function.
  - Instead of expanding the shallowest node, uniform-cost search expands the node n with the lowest path cost $g(n)$**.** This is done by storing the frontier as a priority queue ordered by g.
  - Two differences from BFS:
    - The first is that the goal test is applied to a node when it is selected for expansion rather than when it is first generated.
      - The reason is that the first goal node that is generated may be on a suboptimal path.
    - The second difference is that a test is added in case a better path is found to a node currently on the frontier.

Uniform-cost search

```
function UNIFORM-COST-SEARCH(problem) returns a solution, or failure
  node ← a node with STATE = problem.INITIAL-STATE, PATH-COST = 0
  frontier ← a priority queue ordered by PATH-COST, with node as the only elem
  explored ← an empty set
  loop do
     if EMPTY?(frontier) then return failure
     node ← POP(frontier)  /* chooses the lowest-cost node in frontier */
     if problem.GOAL-TEST(node.STATE) then return SOLUTION(node)
     add node.STATE to explored
     for each action in problem.ACTIONS(node.STATE) do
        child ← CHILD-NODE(problem, node, action)
        if child.STATE is not in explored or frontier then
           frontier ← INSERT(child, frontier)
        else if child.STATE is in frontier with higher PATH-COST then
           replace that frontier node with child
```



Uniform-cost search on a graph. The algorithm is identical to the general graph search algorithm, except for the use of a priority queue and the addition of an extra check in case a shorter path to a frontier state is discovered. The data structure for frontier needs to support efficient membership testing, so it should combine the capabilities of a priority queue and a hash table.

Depth-First Search (DFS)

- Depth-first search always expands the **deepest node** in the current frontier of the search tree.
  - The search proceeds immediately to the deepest level of the search tree, where **the nodes have no successors**.
  - Whereas breadth-first-search uses a FIFO queue, depth-first search uses a LIFO queue.
    - A LIFO queue means that the **most recently generated node** is chosen for expansion.
- It is common to implement depth-first search with a **recursive** function that calls itself on each of its children in turn.
- The graph-search version, which avoids repeated states and redundant paths, is **complete** in finite state spaces because it will eventually expand every node.
- The tree-search version, on the other hand, is **not complete**.
  - Consider Arad-Sibiu-Ard-… (Infinite loop)
- Depth-first tree search can be modified at no extra memory cost so that it checks new states against those on the path from the root to the current node; this avoids infinite loops in finite state spaces but does not avoid the proliferation of redundant paths.

Depth-First Search (DFS)

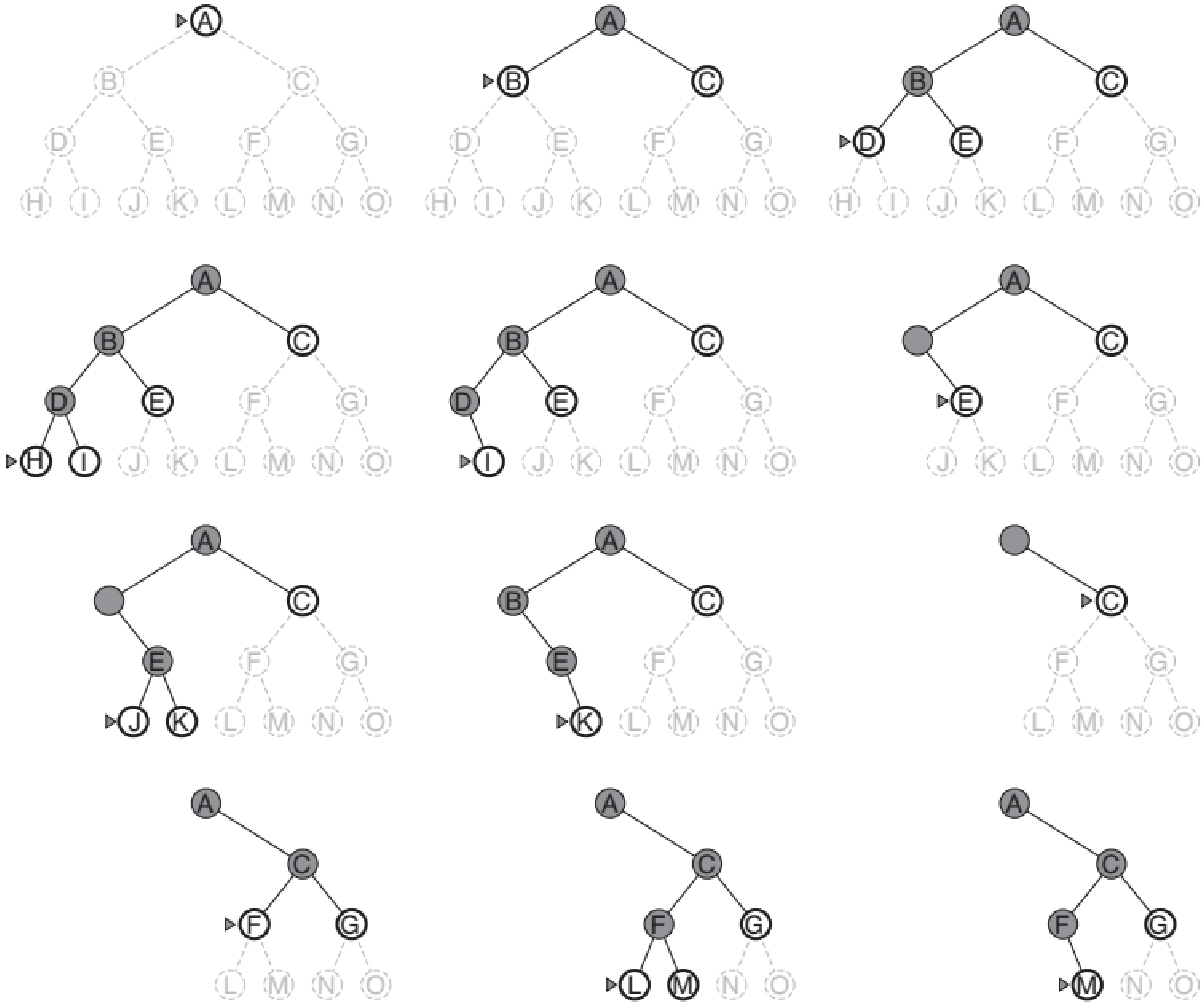
- DFS is not optimal.
  - In the example, it will go explore the entire left subtree even node C is a goal node.
  - If node J were also a goal node, then depth-first search would return it as a solution instead of C, which would be a better solution; hence, depth-first search is not optimal.
- Time Complexity: DFS tree may generate all of the $O(b^m)$ nodes in the tree.
  - m itself can be much larger than d (the depth of the shallowest solution) and is infinite if the tree is unbounded.
- Space Complexity: a depth-first tree search needs to store only a single path from the root to a leaf node, along with the remaining unexpanded sibling nodes for each node on the path.
  - Once a node has been expanded, it can be removed from memory as soon as all its descendants have been fully explored.
  - So, it requires $O(bm)$ nodes.

Depth-First Search (DFS)

```
function DEPTH-LIMITED-SEARCH(problem, limit) returns a solution, or failure/cutoff
   return RECURSIVE-DLS(MAKE-NODE(problem.INITIAL-STATE), problem, limit)

function RECURSIVE-DLS(node, problem, limit) returns a solution, or failure/cutoff
   if problem.GOAL-TEST(node.STATE) then return SOLUTION(node)
   else if limit = 0 then return cutoff
   else
       cutoff_occurred? ← false
       for each action in problem.ACTIONS(node.STATE) do
           child ← CHILD-NODE(problem, node, action)
           result ← RECURSIVE-DLS(child, problem, limit − 1)
           if result = cutoff then cutoff_occurred? ← true
           else if result ≠ failure then return result
       if cutoff_occurred? then return cutoff else return failure
```

Depth-First Search (DFS)



Depth-first search on a binary tree. The unexplored region is shown in light gray. Explored nodes with no descendants in the frontier are removed from memory. Nodes at depth 3 have no successors and M is the only goal node.

Depth-limited search

- The embarrassing failure of depth-first search in infinite state spaces can be **alleviated** by supplying depth-first search with a **predetermined depth limit** l.
  - In *depth-limited search*, nodes at depth l are treated as if they have no successors.
- The **depth limit** solves the infinite-path problem.
- It also introduces an additional source of incompleteness if we choose $l < d$, that is, the shallowest goal is beyond the depth limit.
- Its time complexity is O(b^l) and its space complexity is $O(bl)$.
- Depth-first search can be viewed as a special case of depth-limited search with $l = \infty$.
- Sometimes, depth limits can be based on **knowledge** of the problem.
  - For example, on the map of Romania there are 20 cities. Therefore, we know that if there is a solution, it must be of length 19 at the longest, so l = 19 is a possible choice.
  - Also, we would discover that any city can be reached from any other city in at **most 9 steps**. This number, known as the **diameter** of the state space, gives us a better depth limit, which leads to a more efficient depth-limited search.

Depth-limited search

```
function DEPTH-LIMITED-SEARCH(problem, limit) returns a solution, or failure/cutoff
   return RECURSIVE-DLS(MAKE-NODE(problem.INITIAL-STATE), problem, limit)

function RECURSIVE-DLS(node, problem, limit) returns a solution, or failure/cutoff
   if problem.GOAL-TEST(node.STATE) then return SOLUTION(node)
   else if limit = 0 then return cutoff
   else
       cutoff_occurred? ← false
       for each action in problem.ACTIONS(node.STATE) do
           child ← CHILD-NODE(problem, node, action)
           result ← RECURSIVE-DLS(child, problem, limit − 1)
           if result = cutoff then cutoff_occurred? ← true
           else if result ≠ failure then return result
       if cutoff_occurred? then return cutoff else return failure
```

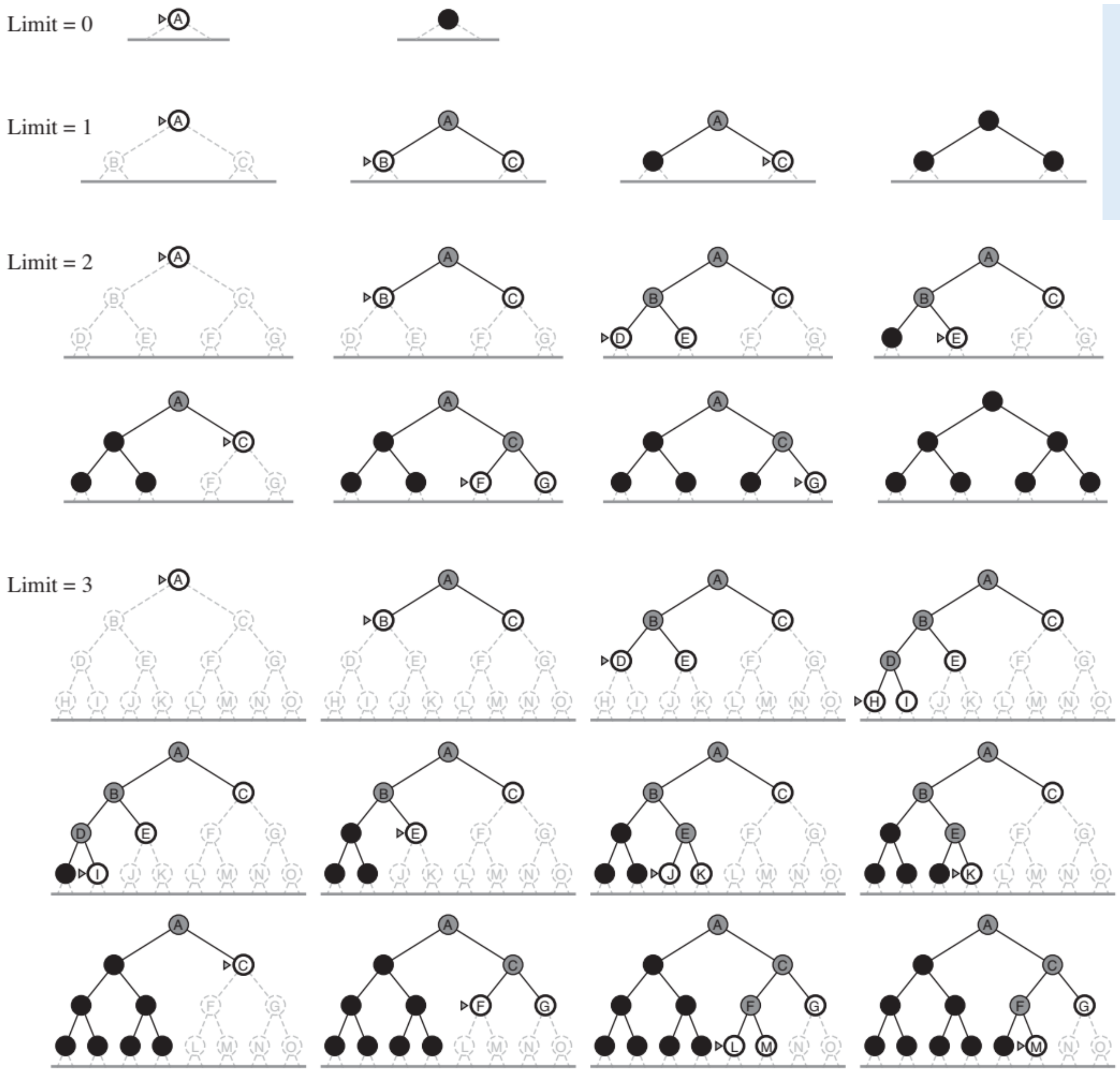
Iterative Deepening Depth-First Search

- Iterative deepening search (or iterative deepening depth-first search) is a general strategy, often used in combination with depth-first tree search, that finds the best depth limit.
  - It does this by gradually increasing the limit—first 0, then 1, then 2, and so on—until a goal is found.
    - This will occur when the depth limit reaches d, the depth of the shallowest goal node.
- Iterative deepening combines the benefits of **depth-first** and **breadth-first** search.
- Space complexity: $O(bd)$.
- Time complexity: Same as BFS, i.e. $O(b^d)$.
- Completeness: like breadth-first search, it is **complete** when the branching factor is finite and **optimal** when the path cost is a nondecreasing function of the depth of the node.
- Iterative deepening search may seem wasteful because states are generated multiple times. It turns out this is not too costly.
  - The reason is that in a search tree with the same (or nearly the same) branching factor at each level, most of the nodes are in the bottom level, so it does not matter much that the upper levels are generated multiple times.

Iterative Deepening Depth-First Search

```
function ITERATIVE-DEEPENING-SEARCH(problem) returns a solution, or failure
   for depth = 0 to ∞ do
      result ← DEPTH-LIMITED-SEARCH(problem, depth)
      if result ≠ cutoff then return result
```

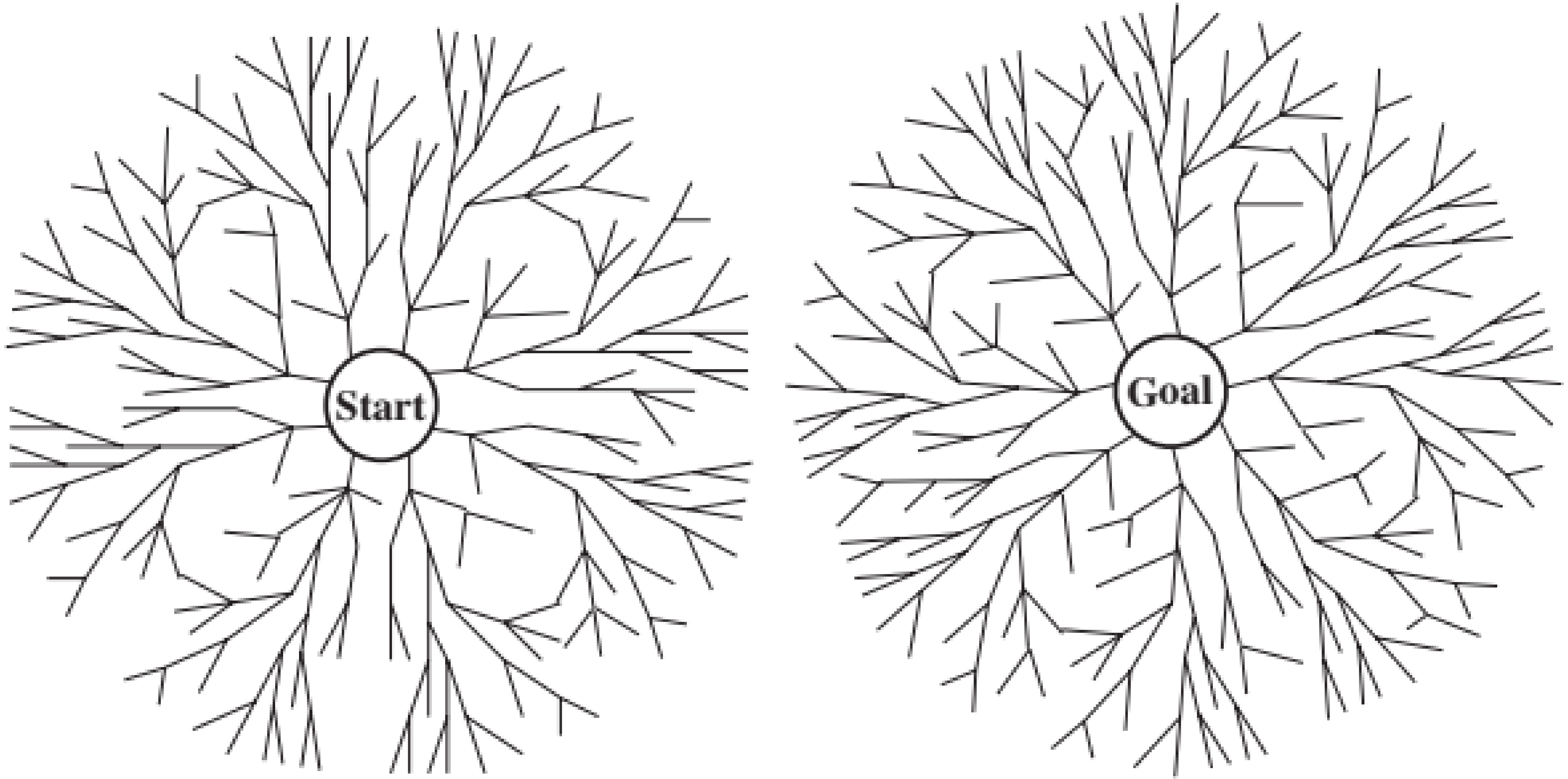
Iterative Deepening Depth-First Search



Bidirectional search

- The idea behind bidirectional search is to run two simultaneous searches—one forward from the initial state and the other backward from the goal—hoping that the two searches meet in the middle.
- Time and complexity: $b^{d/2} + b^{d/2} = O(b^{d/2})$
- Bidirectional search is implemented by replacing the goal test with a check to see whether the frontiers of the two searches intersect; if they do, a solution has been found.
- Consider the question of what we mean by “the goal” in searching “backward from the goal.” For the 8-puzzle and for finding a route in Romania, there is just one goal state, so the backward search is very much like the forward search.
- If there are several explicitly listed goal states—for example, the two dirt-free goal states—then we can construct a new dummy goal state whose immediate predecessors are all the actual goal states.
  - But if the goal is an abstract description, such as the goal that “no queen attacks another queen” in the n-queens problem, then bidirectional search is difficult to use.

Bidirectional search



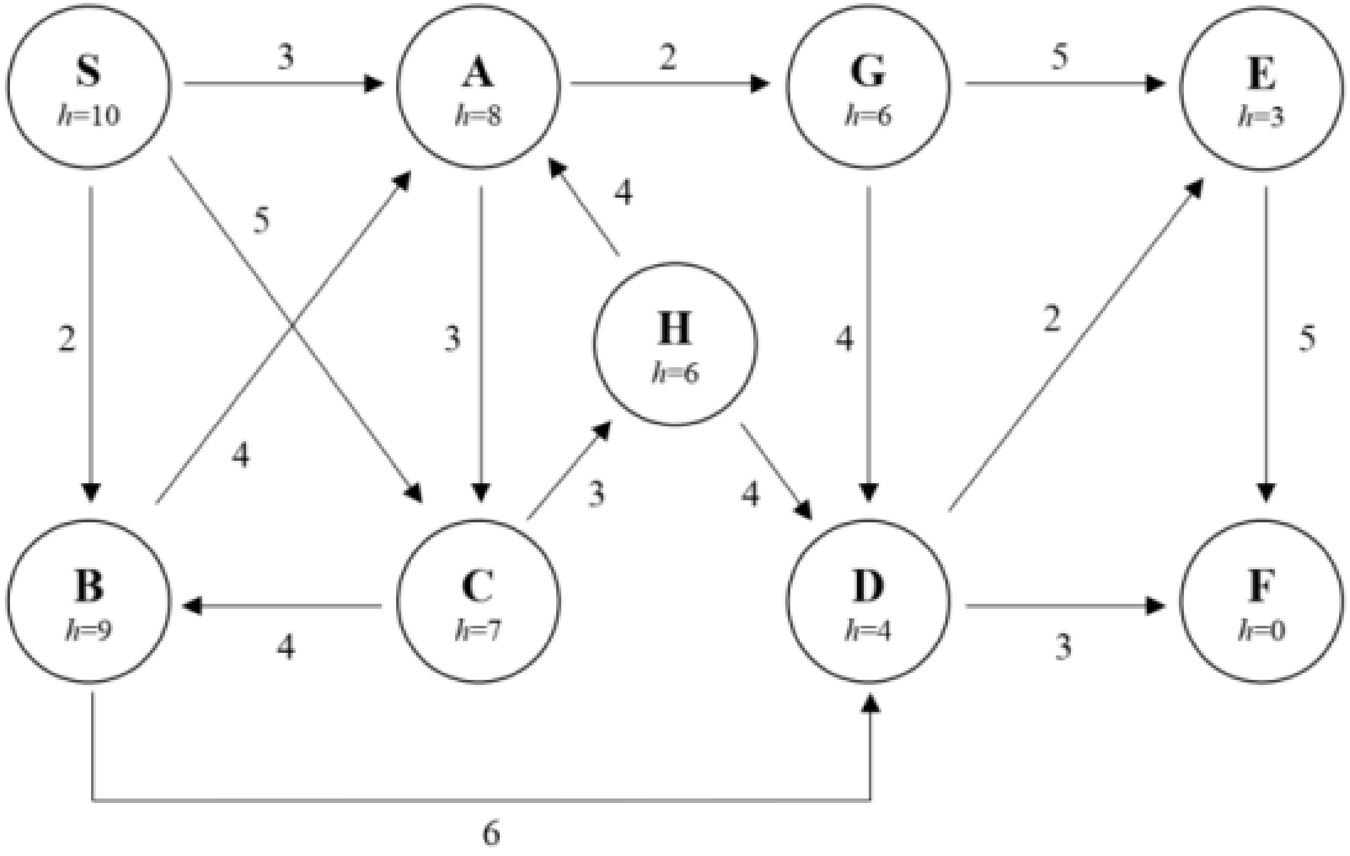
Comparison

Criterion	Breadth-First	Uniform-Cost	Depth-First	Depth-Limited	Iterative Deepening	Bidirectional (if applicable)
Complete?	Yes^{a}	Yes^{a,b}	No	No	Yes^{a}	Yes^{a,d}
Time	$O(b^d)$	$O(b^{1+\lfloor C^*/\epsilon \rfloor})$	$O(b^m)$	$O(b^\ell)$	$O(b^d)$	$O(b^{d/2})$
Space	$O(b^d)$	$O(b^{1+\lfloor C^*/\epsilon \rfloor})$	$O(bm)$	$O(b\ell)$	$O(bd)$	$O(b^{d/2})$
Optimal?	Yes^{c}	Yes	No	No	Yes^{c}	Yes^{c,d}

Evaluation of tree-search strategies. b is the branching factor; d is the depth of the shallowest solution; m is the maximum depth of the search tree; l is the depth limit. Superscript caveats are as follows: [a] complete if b is finite; [b] complete if step costs ≥ for positive ; [c] optimal if step costs are all identical; [d] if both directions use breadth-first search.

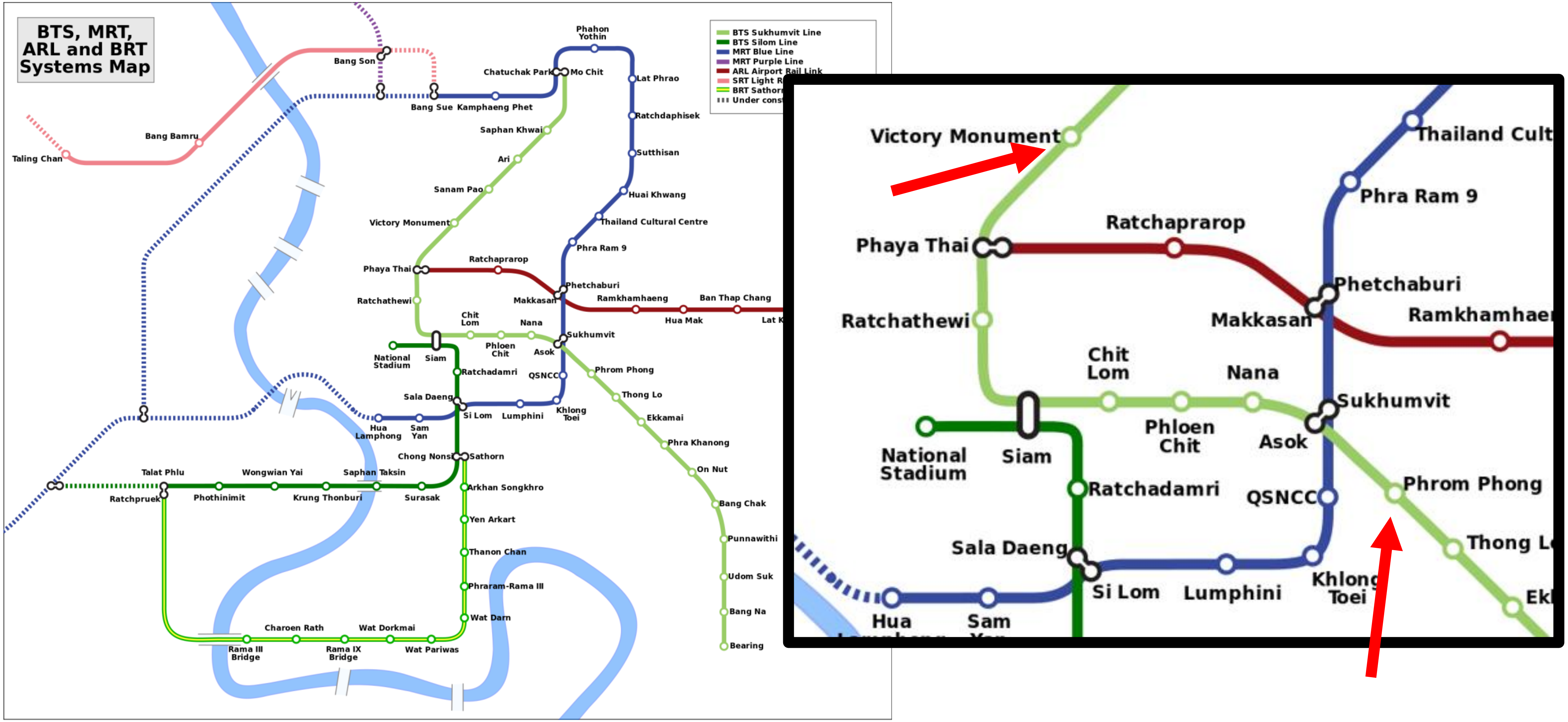
Exercise

Problem: ให้นักเรียนเขียนลำดับการ visit nodes ตามความเข้าใจ เมื่อทำเสร็จแล้ว
ให้เช็คคำตอบกับผลลัพธ์จากการรันโปรแกรมของตนเอง



Single-Source Shortest Paths

Single-Source Shortest Paths



Shortest-paths problem

Note: Edge weights can represent metrics other than distances, such as time, cost, penalties, loss, or any other quantity that accumulates linearly along a path and that we would want to minimize

- In a shortest-paths problem, we are given a weighted, directed graph $G(V,E)$, with weight function $w: E \to R$ mapping edges to real-valued weights.
- The weight $w(p)$ of path $p = \langle v_0, v_1, \dots, v_k \rangle$ is the sum of the weights of its constituent edges: $w(p) = \sum_{i=1}^{k} w(v_{i-1}, v_i)$
- We define the shortest-path weight $\delta(u,v)$ from u to v by

$$\delta(u,v) = \begin{cases} \min\{w(p) : u \overset{p}{\leadsto} v\} & \text{if there is a path from } u \text{ to } v\text{ ,} \\ \infty & \text{otherwise .} \end{cases}$$

- A shortest path from vertex u to vertex v is then defined as any path p with weight $w(p) = \delta(u,v)$.

Optimal substructure of a shortest path

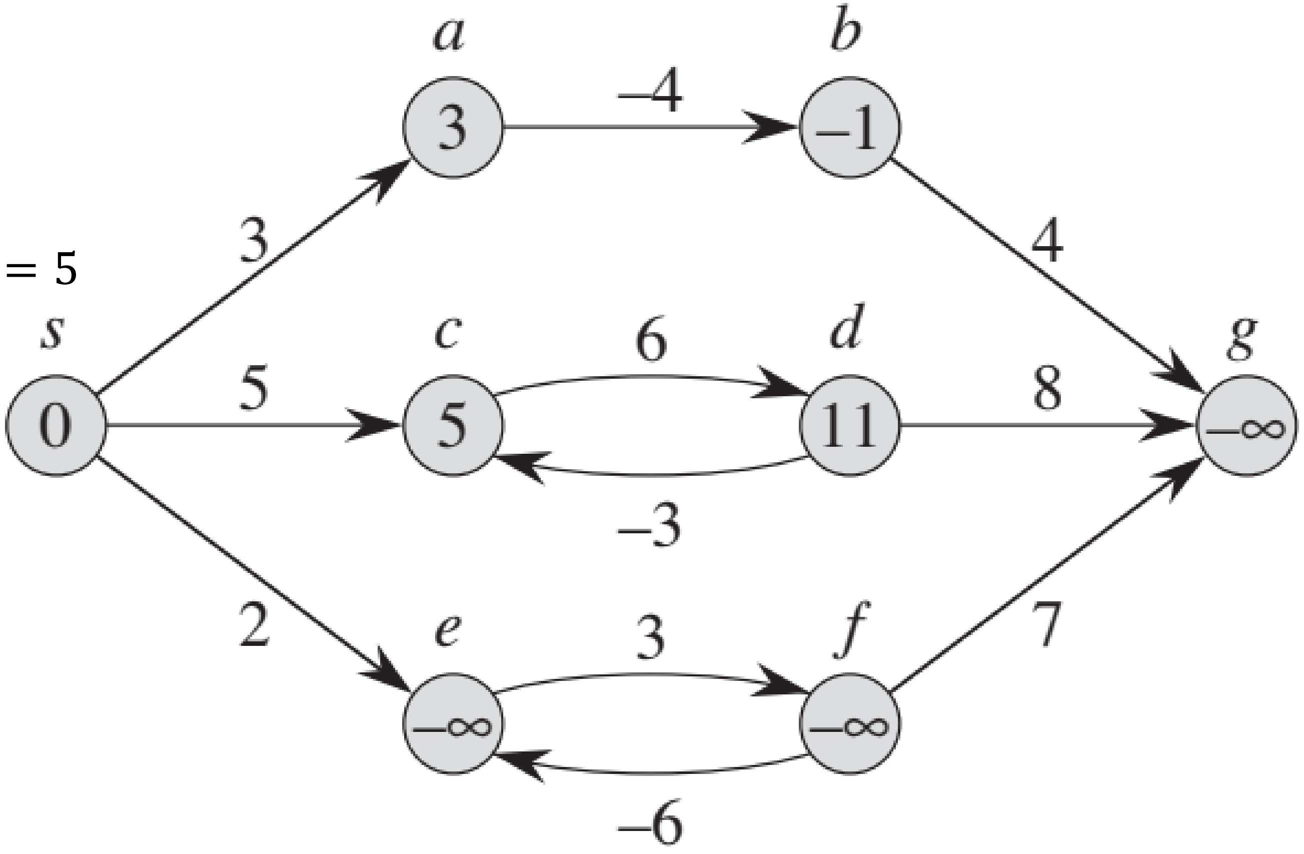
- Shortest-paths algorithms typically rely on the property that a shortest path between two vertices contains other shortest paths within it.
- Recall that optimal substructure is one of the key indicators that dynamic programming and the greedy method might apply.
- Dijkstra's algorithm is a greedy algorithm.

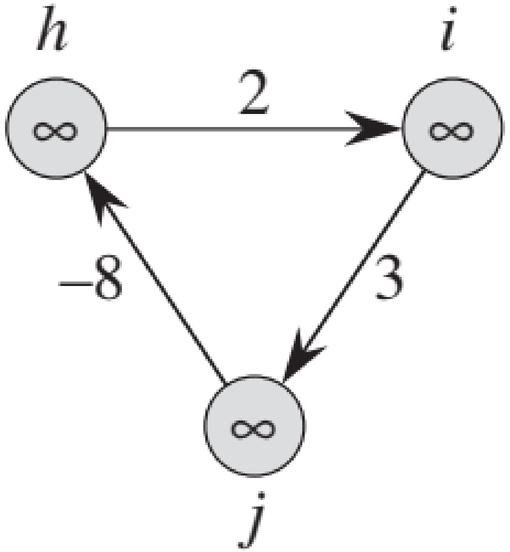
Negative-weight edges

- Some instances of the single-source shortest-paths problem may include edges whose weights are negative.
- If the graph $G(V, E)$ contains no negative weight cycles reachable from the source s, then for all $v \in V$, the shortest-path weight $\delta(s, v)$ remains well defined, even if it has a negative value.
- If the graph contains a negative-weight cycle reachable from s, however, shortest-path weights are not well defined.

Negative-weight edges

$\delta(s, c) = 5$





$\delta(s, e) = -\infty$

Cycles

- Can a shortest path contain a cycle? As we have just seen, it cannot contain a **negative-weight cycle**.
- How about **positive-weight cycle?**
  - Nor can it contain a positive-weight cycle, since removing the cycle from the path produces a path with the same source and destination vertices and a lower path weight.
- Therefore, without loss of generality we can assume that when we are finding shortest paths, they have no cycles, i.e., they are simple paths.

Relaxation

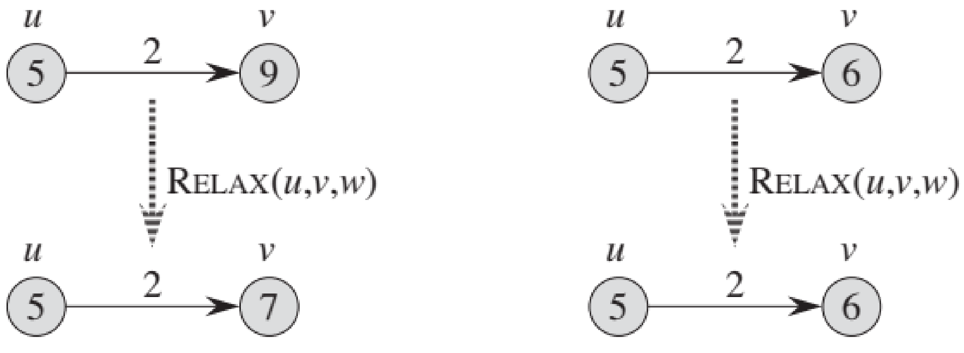
- For each vertex $v \in V$, we maintain an attribute $v.d$, which is an upper bound on the weight of a shortest path from source s to v.
  - We call $v.d$ a shortest-path estimate.
- We initialize the shortest-path estimates and predecessors by the following $\Theta(V)$-time procedure:

```
INITIALIZE-SINGLE-SOURCE(G, s)
for each vertex v ∈ G.V
    v.d = ∞
    v.π = NIL
s.d = 0
```

Relaxation

- The process of relaxing an edge (u, v) consists of testing whether we can improve the shortest path to v found so far by going through u and, if so, updating $v.d$ and $v.\pi$.
- The following code performs a relaxation step on edge (u, v) in $O(1)$ time:

```
RELAX(u, v, w)
if v.d > u.d + w(u, v)
      v.d = u.d + w(u, v)
      v.π = u
```



(a) (b)

The Bellman-Ford algorithm

The Bellman-Ford algorithm

- The **Bellman-Ford algorithm** solves the single-source shortest-paths problem in the general case in which edge weights may be negative.
- Given a weighted, directed graph $G = (V, E)$ with source s and weight function $w: E \rightarrow R$, the Bellman-Ford algorithm returns a Boolean value indicating whether or not there is a negative-weight cycle that is reachable from the source.
  - If there is such a cycle, the algorithm indicates that no solution exists.
  - If there is no such cycle, the algorithm produces the shortest paths and their weights.
- The Bellman-Ford algorithm relaxes each edge $|V| - 1$ times.
- The algorithm relaxes edges, progressively decreasing an estimate $v.d$ on the weight of a shortest path from the source s to each vertex $\delta(s, v)$ until it achieves the actual shortest-path weight

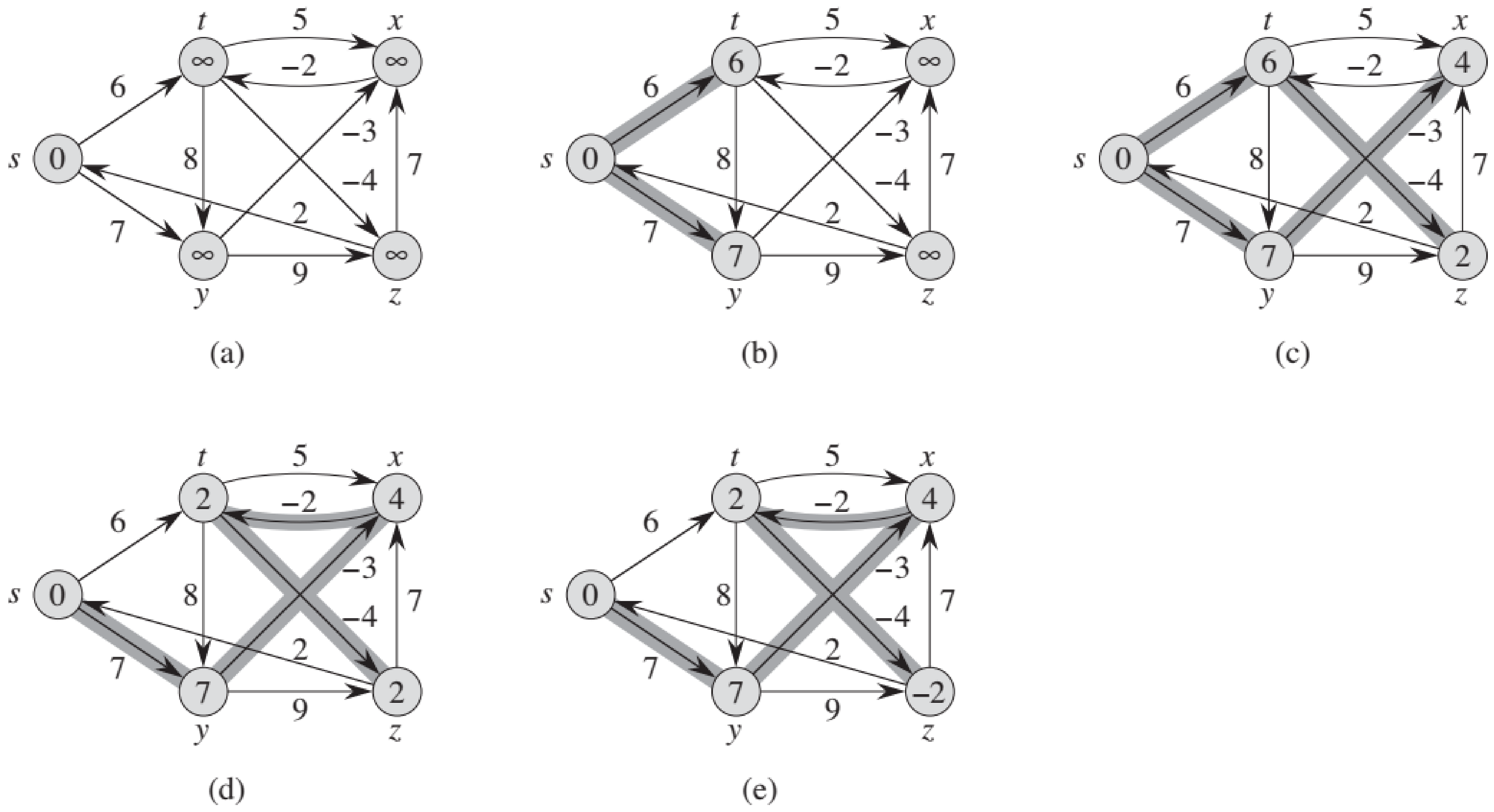
The Bellman-Ford algorithm

```
BELLMAN-FORD(G, w, s)
    INITIALIZE-SINGLE-SOURCE(G, s)
    for i = 1 to |G.V| − 1
        for each edge (u, v) ∈ G.E
            RELAX(u, v, w)
    for each edge (u, v) ∈ G.E
        if v.d > u.d + w(u, v)
            return FALSE
    return TRUE
```

```
INITIALIZE-SINGLE-SOURCE(G, s)
    for each vertex v ∈ G.V
        v.d = ∞
        v.π = NIL
    s.d = 0
```

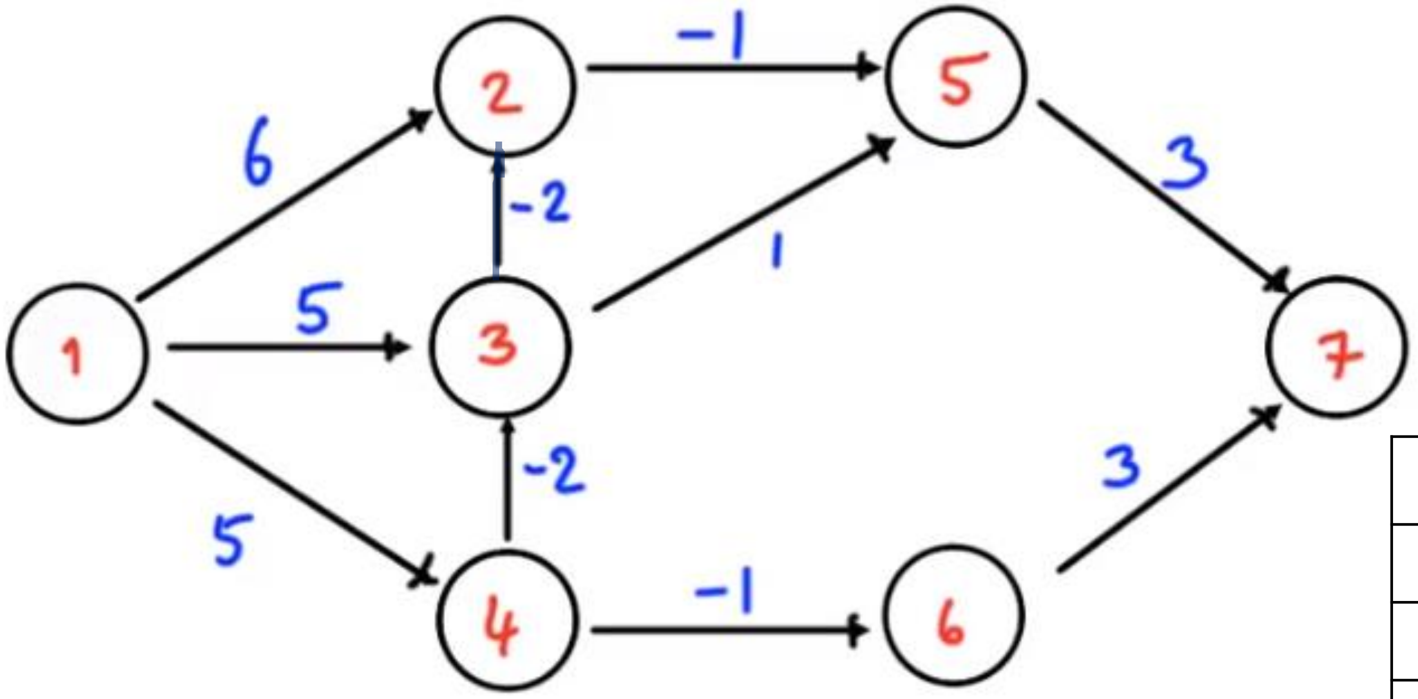
```
RELAX(u, v, w)
    if v.d > u.d + w(u, v)
        v.d = u.d + w(u, v)
        v.π = u
```

The Bellman-Ford algorithm



Quiz#08

Source: node 1

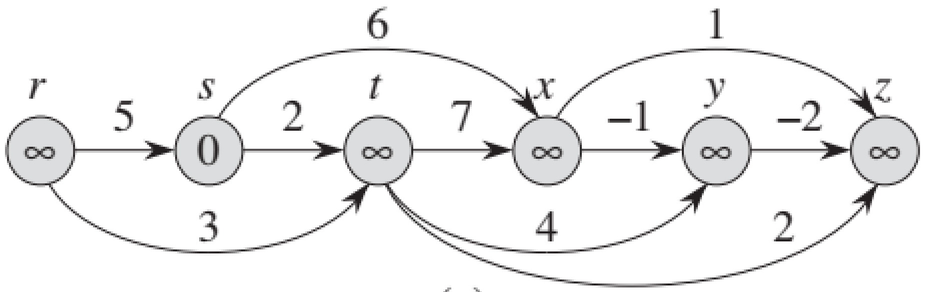


Complexity Analysis

- The Bellman-Ford algorithm runs in time $O(VE)$, since the initialization takes $\Theta(V)$ time, each of the $|V| - 1$ passes over the edges takes $\mathrm{O}(E)$ time.

Single-source shortest paths in directed acyclic graphs

- By relaxing the edges of a weighted dag (directed acyclic graph) $G = (V, E)$ according to a topological sort of its vertices, we can compute shortest paths from a single source in $\Theta(V + E)$ time.
- Shortest paths are always well defined in a dag, since even if there are negative-weight edges, no negative-weight cycles can exist.
- The algorithm starts by topologically sorting the dag to impose a linear ordering on the vertices.



(a)

```
DAG-SHORTEST-PATHS(G, w, s)
topologically sort the vertices of G
INITIALIZE-SINGLE-SOURCE(G, s)
for each vertex u, taken in topologically sorted order
    for each vertex v ∈ G.Adj[u]
        RELAX(u, v, w)
```

Dijkstra’s algorithm

Dijkstra's algorithm

- Dijkstra's algorithm solves the single-source shortest-paths problem on a weighted, directed graph $G = (V, E)$ for the case in which all edge weights are nonnegative.
- As we shall see, with a good implementation, the running time of Dijkstra's algorithm is lower than that of the Bellman-Ford algorithm.
- Dijkstra's algorithm maintains a set S of vertices whose final shortest-path weights from the source s have already been determined.
- The algorithm repeatedly selects the vertex $u \in V - S$ with the minimum shortest-path estimate, adds u to S, and relaxes all edges leaving u.

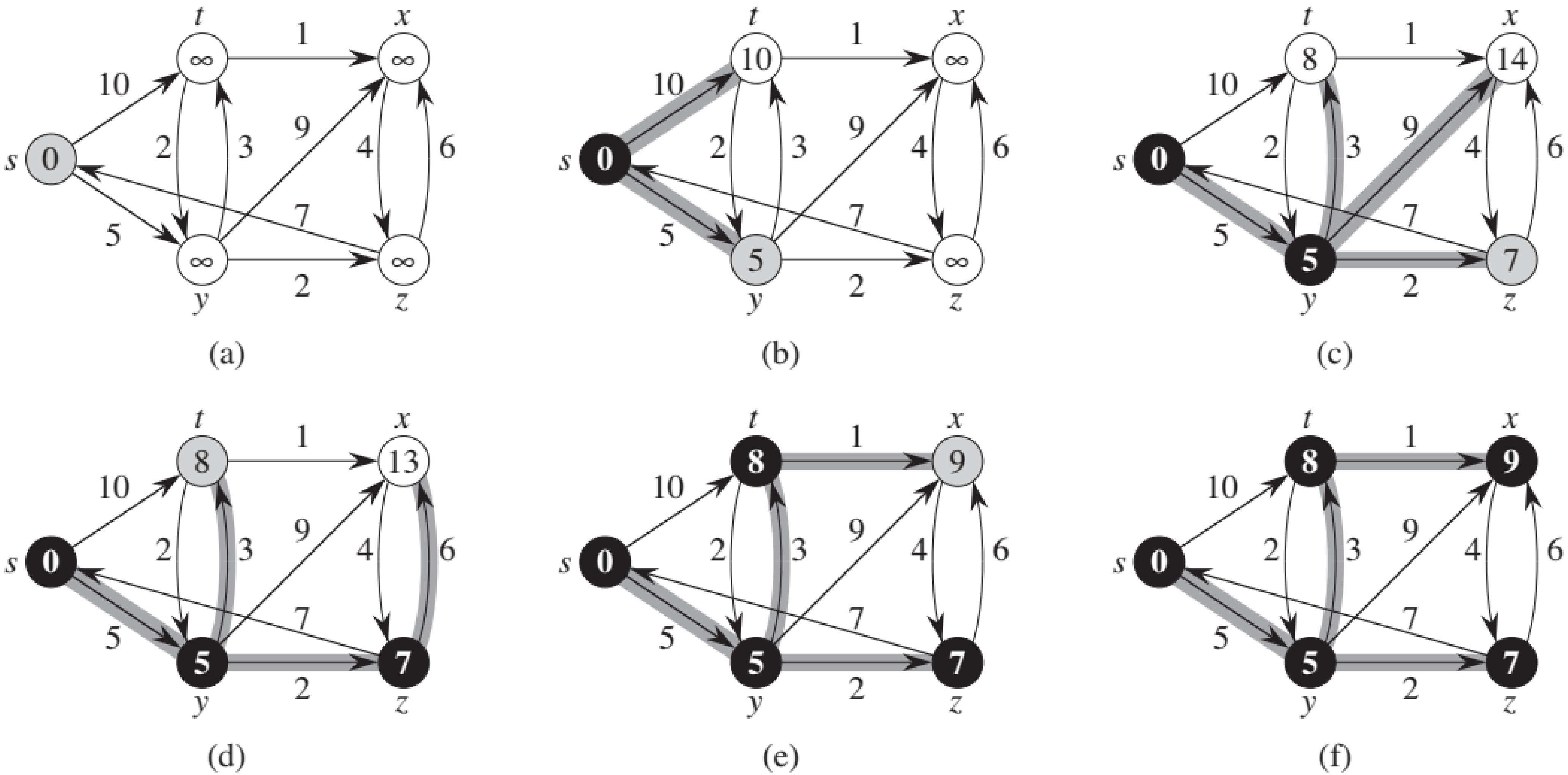
Dijkstra's algorithm

```
DIJKSTRA(G, w, s)
INITIALIZE-SINGLE-SOURCE(G, s)
S = Ø
Q = G.V
while Q ≠ Ø
    u = EXTRACT-MIN(Q)
    S = S ∪ {u}
    for each vertex v ∈ G.Adj[u]
        RELAX(u, v, w)
```

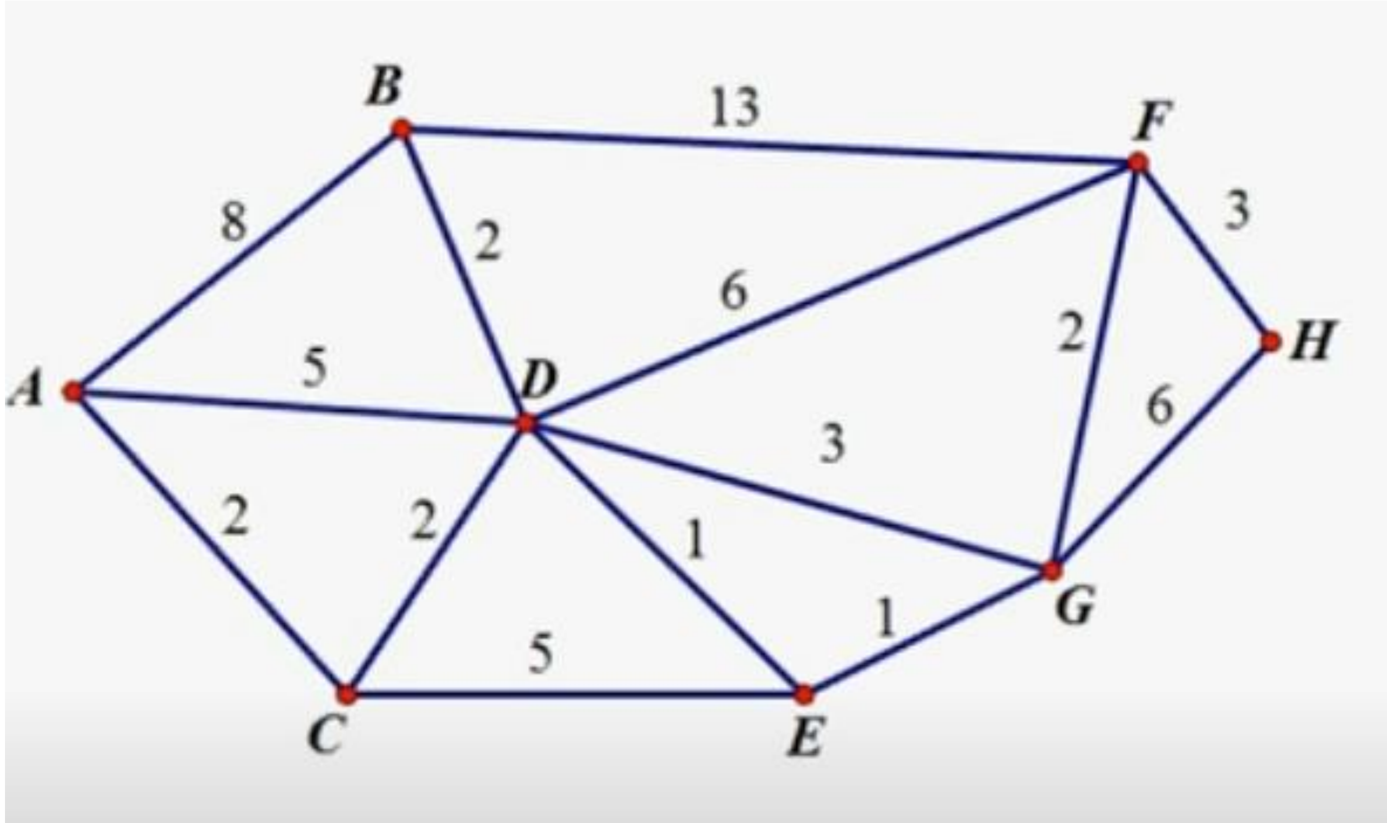
```
INITIALIZE-SINGLE-SOURCE(G, s)
for each vertex v ∈ G.V
    v.d = ∞
    v.π = NIL
s.d = 0
```

```
RELAX(u, v, w)
if v.d > u.d + w(u, v)
    v.d = u.d + w(u, v)
    v.π = u
```

Dijkstra's algorithm



Exercise



โครงสร้างข้อมูลแบบเซตไม่มีส่วนร่วม
Disjoint Sets

Disjoint-set

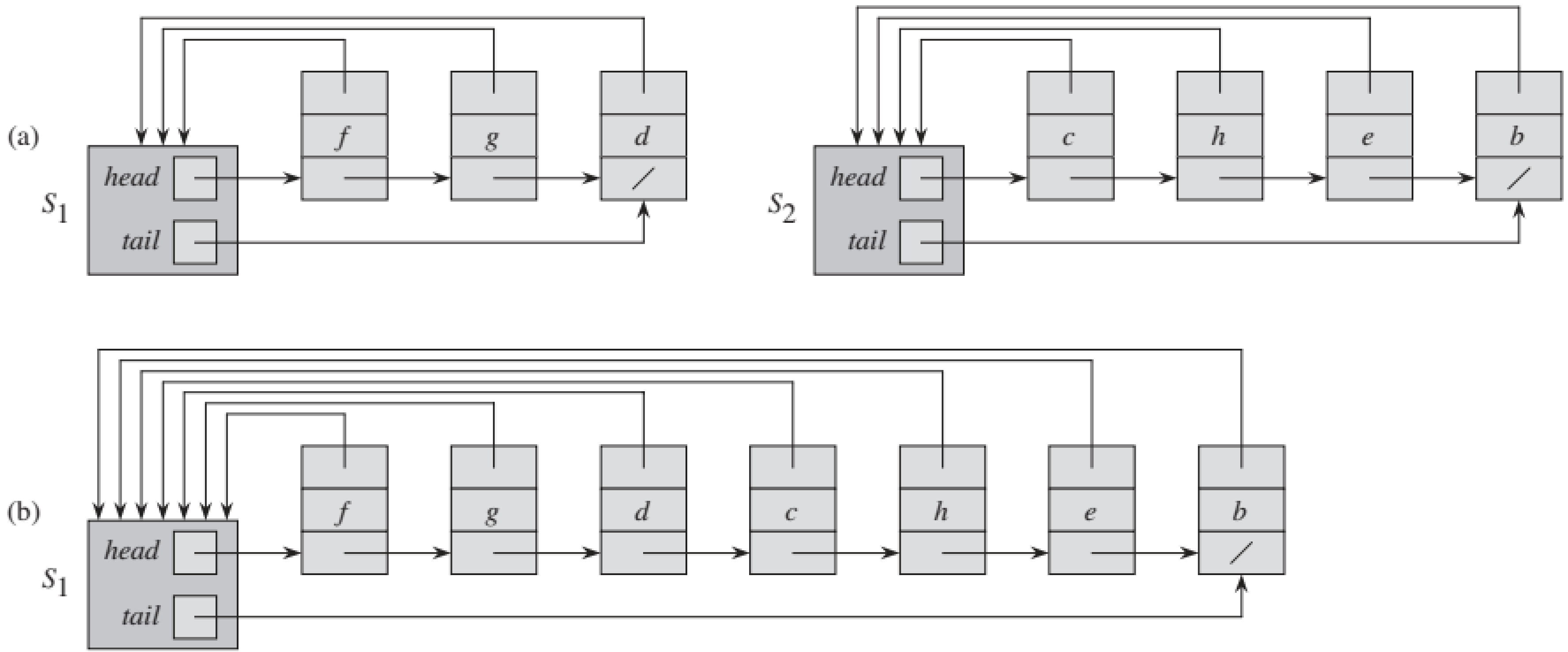
- A disjoint-set data structure maintains a collection $S = \{S_1, \ldots, S_k\}$ of disjoint dynamic sets.
- We identify each set by a representative, which is some member of the set.

Disjoint-set Operations

- As in the other dynamic-set implementations we have studied, we represent each element of a set by an object. Letting x denote an object, we wish to support the following operations:
  - **MAKE-SET(x)** creates a new set whose only member (and thus representative) is x. Since the sets are disjoint, we require that x not already be in some other set.
  - **UNION(x,y)** unites the dynamic sets that contain x and y, say Sx and Sy, into a new set that is the union of these two sets.
  - **FIND-SET(x)** returns a pointer to the representative of the (unique) set containing x.

Implementation

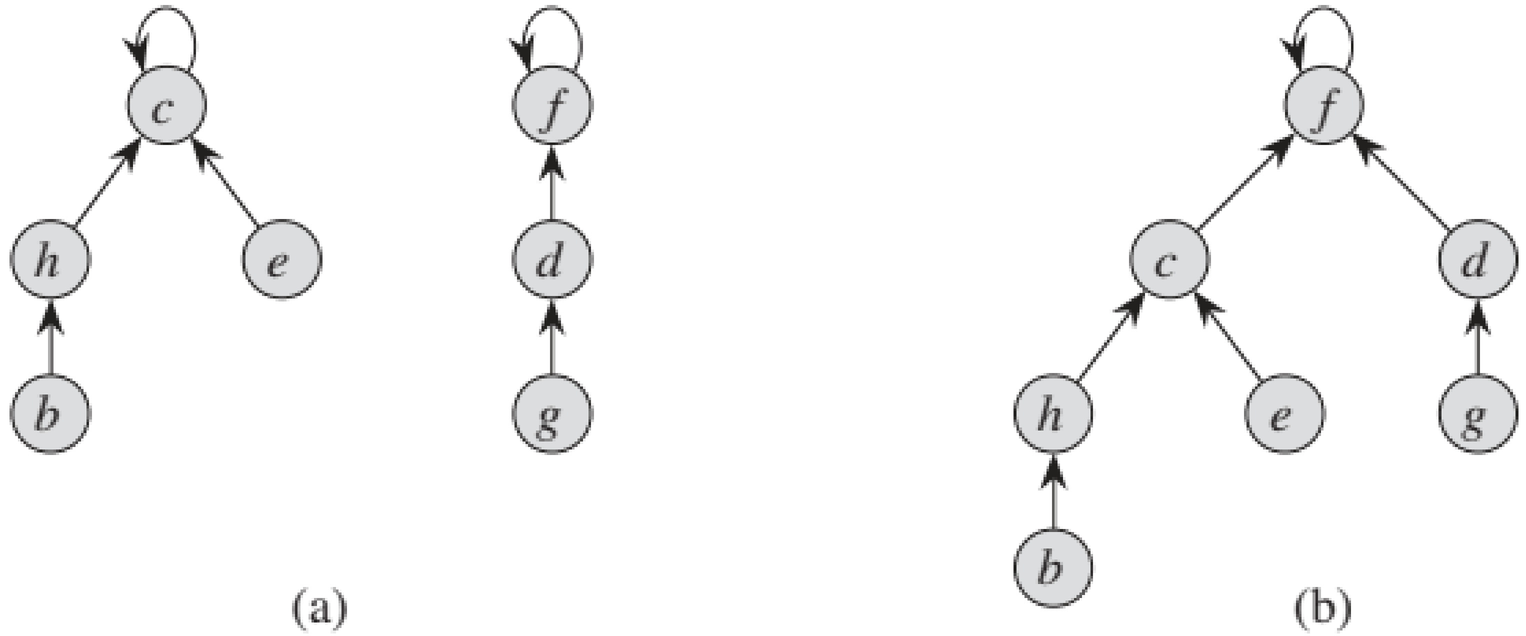
- Linked List representation



Implementation

- Forest Representation.

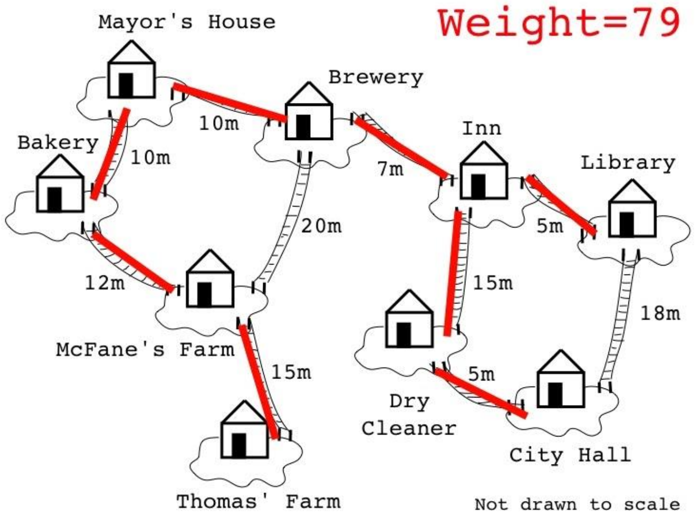
MAKE-SET(x) create a new tree with one node in the forest.
UNION(x,y) following parent pointers until we find the root of the tree.
FIND-SET(x) causes the root of one tree to point to the root of the other.



Minimum Spanning Trees

Minimum Spanning Trees

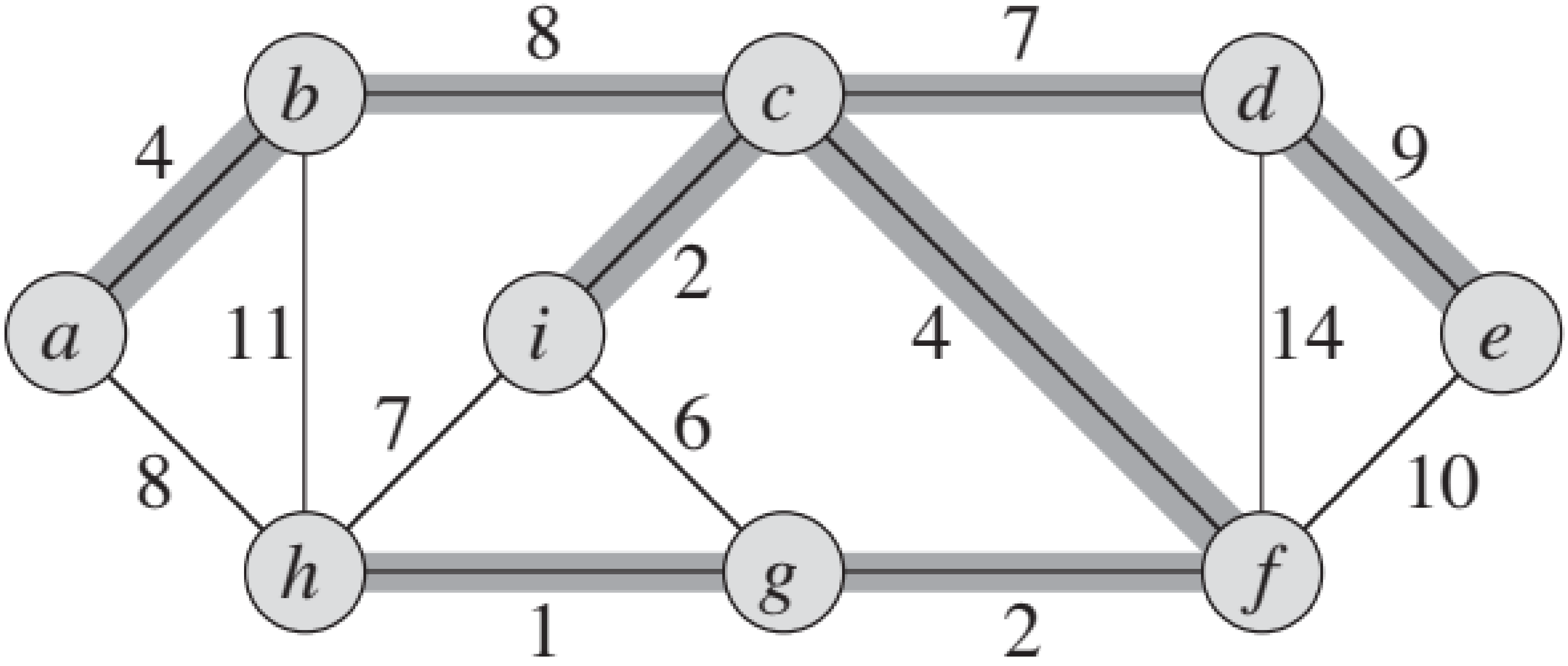
- Electronic circuit designs often need to make the pins of several components electrically equivalent by wiring them together.
- To interconnect a set of n pins, we can use an arrangement of $n-1$ wires, each connecting two pins.
- Of all such arrangements, the one that uses the least amount of wire is usually the most desirable.



Minimum Spanning Trees

- We can model this wiring problem with a connected, undirected graph $G = (V, E)$, where V is the set of pins. E is the set of possible interconnections between pairs of pins, and for each edge $(u, v) \in E$, we have a weight $w(u, v)$ specifying the cost (amount of wire needed) to connect u and v.
- In other words, We then wish to find an acyclic subset $T \subseteq E$, that connects all of the vertices and whose total weight $w(T) = \sum_{(u,v)} T$
- Since T is acyclic and connects all of the vertices, it must form a tree, which we call a spanning tree since it “spans” the graph G.
- We call the problem of determining the tree T the minimum-spanning-tree problem.

Minimum Spanning Trees



Minimum Spanning Trees

- In this lecture, we shall examine two algorithms for solving the minimum spanning-tree problem: Kruskal's algorithm and Prim's algorithm.
- We can easily make each of them run in time $O(E \lg V)$ using ordinary binary heaps.

Growing a minimum spanning tree

Growing a minimum spanning tree

- Assume that we have a *connected, undirected* graph $G = (V, E)$ with a weight function $w: E \rightarrow R$, and we wish to find a minimum spanning tree for G.
- Prior to each iteration, A is a subset of some minimum spanning tree.
- At each step, we determine an edge (u, v) that we can add to A without violating this invariant, so $A \cup \{(u, v)\}$ is also a subset of a minimum spanning tree.
  - We call such an edge a safe edge for A, since we can add it safely to A while maintaining the invariant.

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

We use the loop invariant as follows:

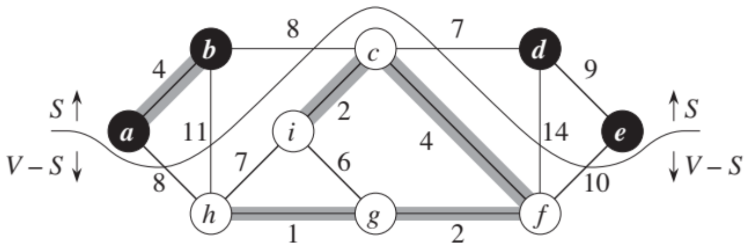
Initialization: After line 1, the set A trivially satisfies the loop invariant.

Maintenance: The loop in lines 2–4 maintains the invariant by adding only safe edges.

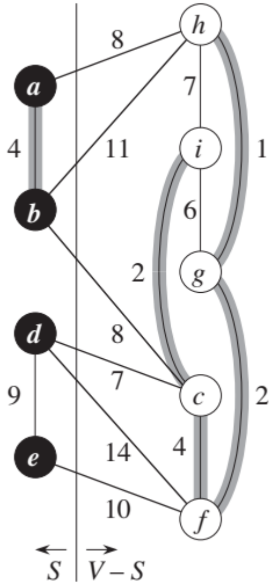
Termination: All edges added to A are in a minimum spanning tree, and so the set A returned in line 5 must be a minimum spanning tree.

Growing a minimum spanning tree

- Let us consider some definitions:
  - A cut $(S, V - S)$ of an undirected graph $G = (V, E)$ is a partition of V.
  - We say that an edge $(u, v) \in E$ crosses the cut if one of its end points is in S and the other is in $V - S$
  - We say that a cut respects a set A of edges if no edge in A crosses the cut.
  - An edge is a light edge crossing a cut if its weight is the minimum of any edge crossing the cut. (able to have more than one light edge)
  - The following theorem are the rule for recognizing safe edges.



(a)



(b)

Let $G = (V, E)$ be a connected, undirected graph with a real-valued weight function w defined on E. Let A be a subset of E that is included in some minimum spanning tree for G, let $(S, V - S)$ be any cut of G that respects A, and let (u, v) be a light edge crossing $(S, V - S)$. Then, edge (u, v) is safe for A.

Complexity Analysis

- The loop in lines 2-4 of GST-MST executes $|V| - 1$ times because it finds one of the $|V| - 1$ edges of a minimum spanning tree in each iteration.
- Initially, when $A = \emptyset$ there are $|V|$ trees in G_A and each iteration reduces that number by 1.

Corollary

> Let $G = (V, E)$ be a connected, undirected graph with a real-valued weight function w defined on E. Let A be a subset of E that is included in some minimum spanning tree for G, and let $C = (V_C, E_C)$ be a connected component (tree) in the forest $G_A = (V, A)$. If (u, v) is a light edge connecting C to some other component in G_A, then (u, v) is safe for A.

Kruskal’s algorithm

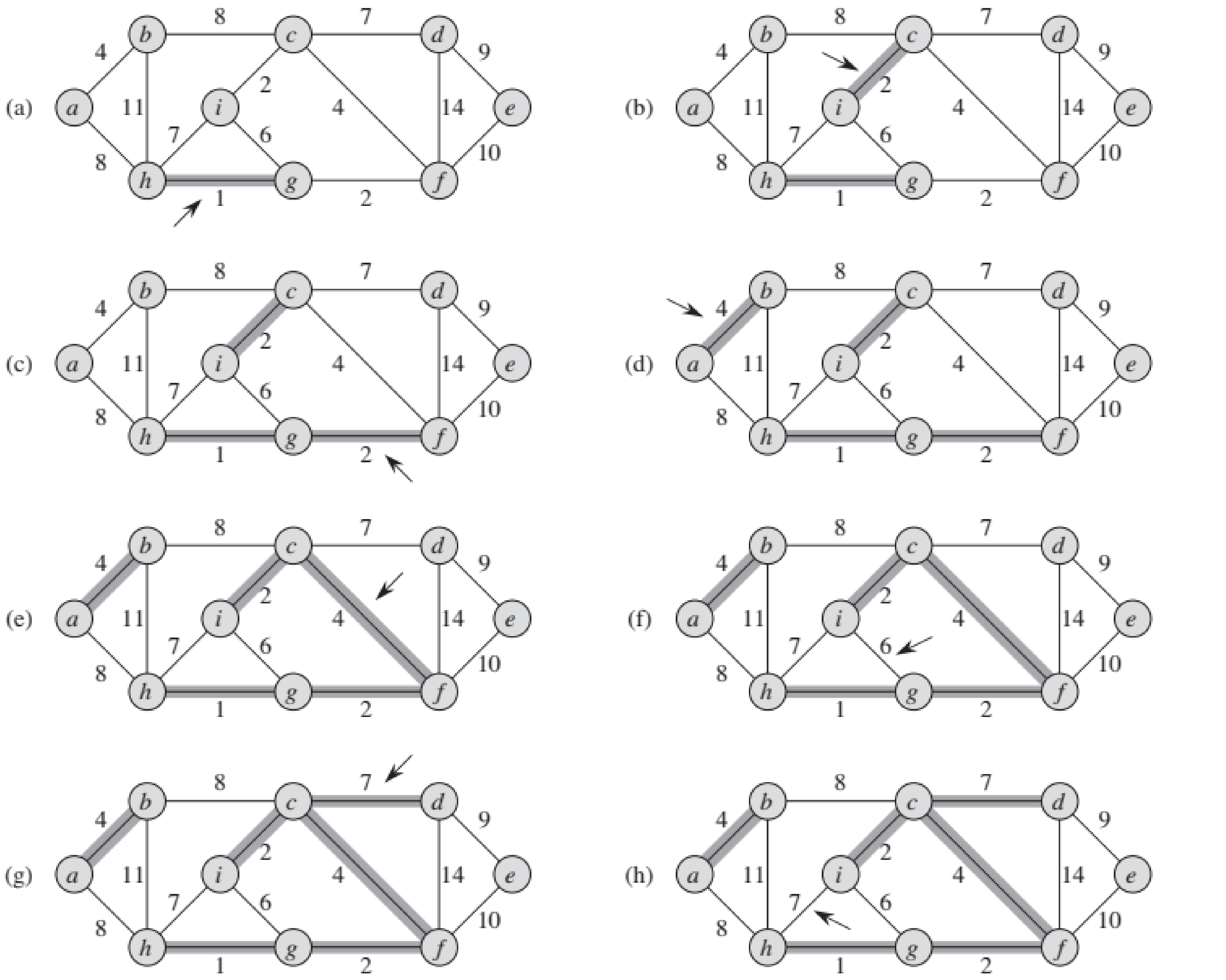
Kruskal's algorithm

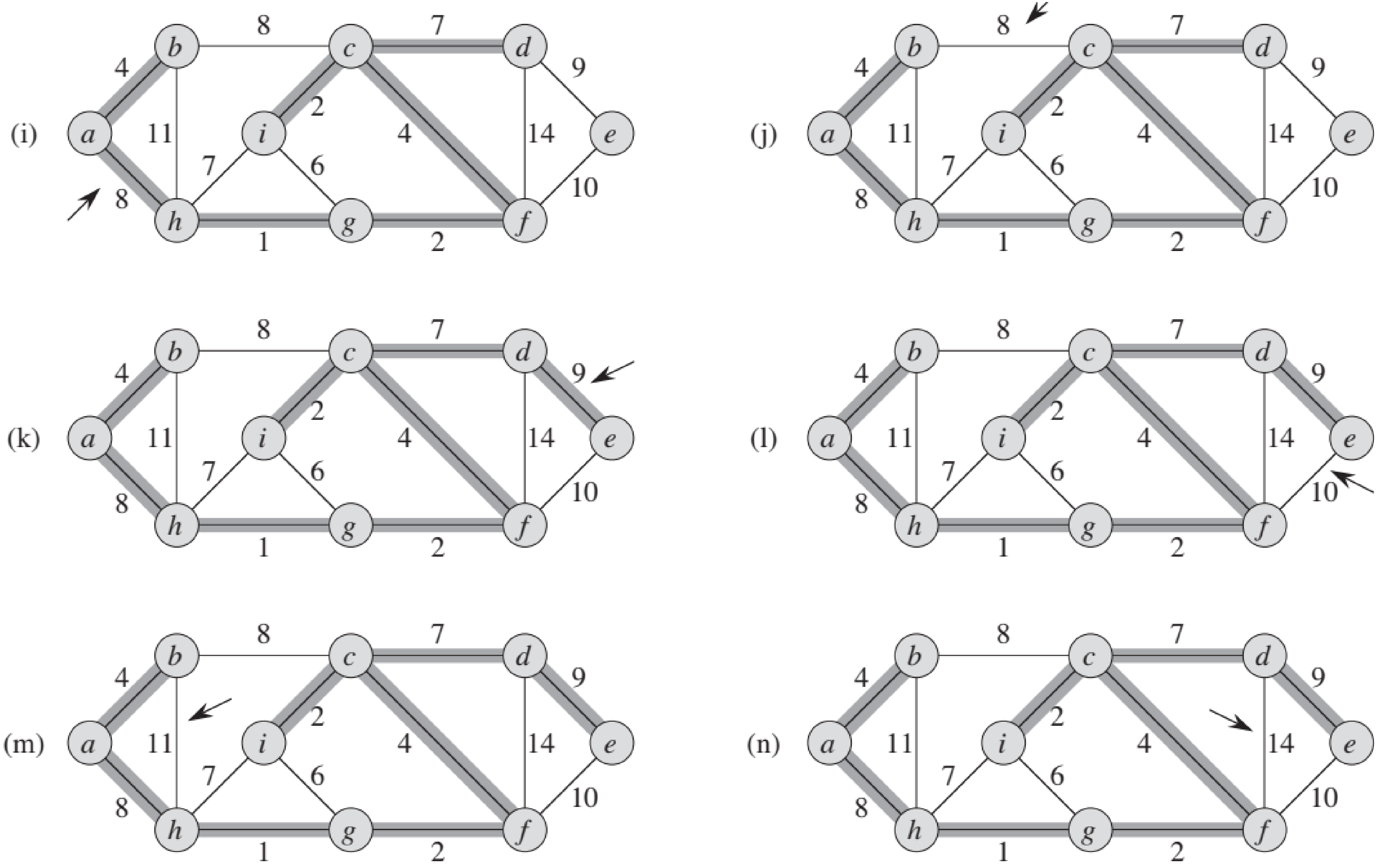
- Kruskal's algorithm finds a safe edge to add to the growing forest by finding, of all the edges that connect any two trees in the forest, an edge (u, v) of least weight.
- Let C_1 and C_2 denote the two trees that are connected by (u, v).
- Since (u, v) must be a light edge connecting C_1 to some other tree, the corollary introduced previously (u, v) is a safe edge for C_1
- Kruskal's algorithm qualifies as a greedy algorithm because at each step it adds to the forest an edge of least possible weight.

Kruskal’s algorithm

MST-KRUSKAL(G, w)

$A = \emptyset$
for each vertex $v \in G.V$
    MAKE-SET(v)
sort the edges of $G.E$ into nondecreasing order by weight w
for each edge $(u, v) \in G.E$, taken in nondecreasing order by weight
    if FIND-SET$(u) \neq$ FIND-SET(v)
        $A = A \cup \{(u, v)\}$
        UNION(u, v)
return A

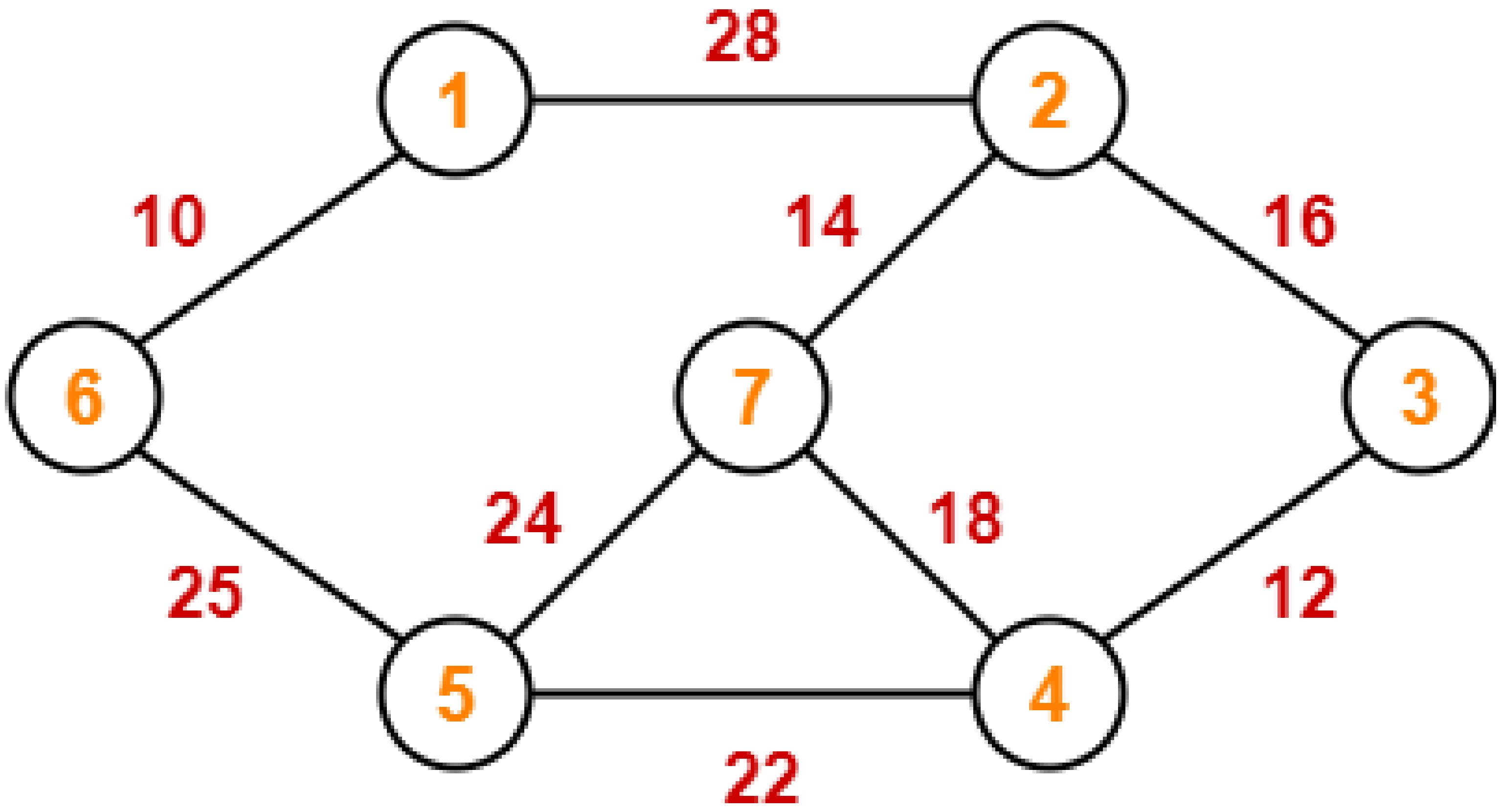

(a)
(b)
(c)
(d)
(e)
(f)
(g)
(h)
a
b
c
d
e
f
g
h
i
4
8
7
9
11
2
4
14
7
6
10
8
1
2


(i)
(j)
(k)
(l)
(m)
(n)

Complexity Analysis

- The running time of Kruskal's algorithm is $O(E \lg V)$.

Quiz#9



Using the Kruskal's algorithm to find The minimum spanning tree.

Prim’s algorithm

Prim's algorithm

- Like Kruskal's algorithm, Prim's algorithm is a special case of the generic minimum-spanning-tree method from Section the previous slide.
- Prim's algorithm operates much like Dijkstra's algorithm for finding shortest paths in a graph.
- Prim's algorithm has the property that the edges in the set A always form a single tree.
- The tree starts from an arbitrary root vertex r and grows until the tree spans all the vertices in V.

Prim's algorithm

```
MST-PRIM(G, w, r)
1  for each u ∈ G.V
2      u.key = ∞
3      u.π = NIL
4  r.key = 0
5  Q = G.V
6  while Q ≠ ∅
7      u = EXTRACT-MIN(Q)
8      for each v ∈ G.Adj[u]
9          if v ∈ Q and w(u, v) < v.key
10             v.π = u
11             v.key = w(u, v)
```

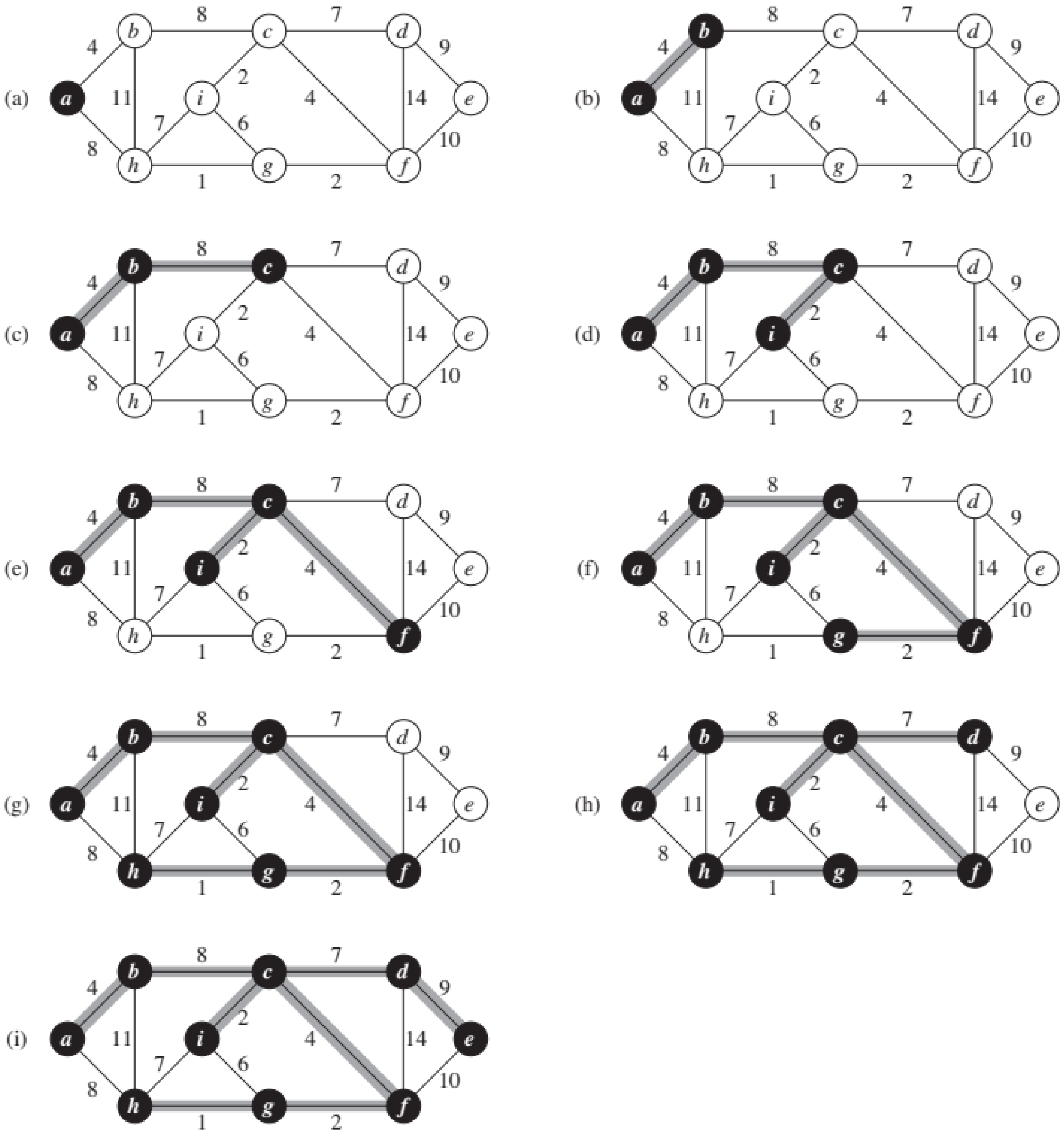
Prior to each iteration of the **while** loop of lines 6–11,

1. $A = \{(v, v.\pi) : v \in V - \{r\} - Q\}$.
2. The vertices already placed into the minimum spanning tree are those in $V - Q$.
3. For all vertices $v \in Q$, if $v.\pi \neq$ NIL, then $v.key < \infty$ and $v.key$ is the weight of a light edge $(v, v.\pi)$ connecting v to some vertex already placed into the minimum spanning tree.

When the algorithm terminates, the min-priority queue Q is empty; the minimum spanning tree A for G is thus

$$A = \{(v, v.\pi) : v \in V - \{r\}\}$$

Example

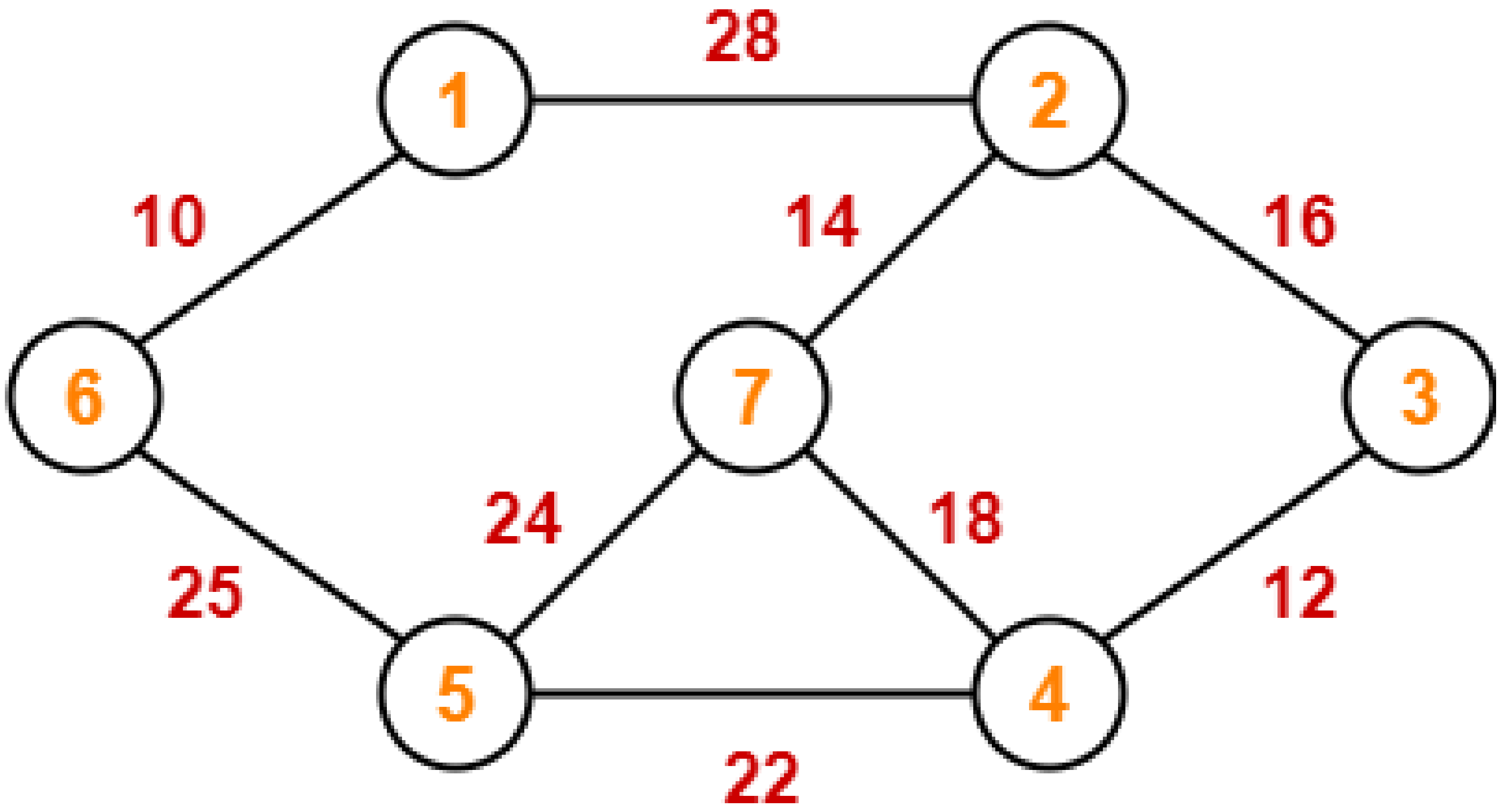


Th

Complexity Analysis

- The running time of Prim's algorithm is $O(E \lg V)$. (Same as Kruskal's algorithm)

Quiz#9



Using the Prim's algorithm to find
The minimum spanning tree.

References