CS 457, Fall 2019

Drexel University, Department of Computer Science

Lecture 15

Maximum Subarray Problem

- Can you provide a dynamic programming solution to this problem?
  - How would you break this problem into sub-problems?
  - You want the solutions to the subproblems to help you solve larger subproblems faster

- If you knew the best subarray ending at A$[i]$, could you find the best subarray ending at A$[i+1]$?
  - Let's create a new array B and store the sum of the best subbary ending at $A[i]$ in $B[i]$
  - The best ending at $A[i+1]$ is either the best ending at A$[i]$ plus A$[i+1]$, or just A$[i+1]$
  - This means that $B[i+1] = \max\{B[i] + A[i+1],\ A[i+1]\}$
  - How quickly can you check which one is best?
  - How quickly can you compute the best subarray ending at A$[i]$ for every $i \in \{1, \dots, n\}$?
  - Can you compute the optimal solution using this information?

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
A	13	–3	–25	20	–3	–16	–23	18	20	–7	12	–5	–22	15	–4	7

maximum subarray

Matrix-Chain Multiplication

- Given sequence $\langle A_1, A_2, \ldots, A_n \rangle$ of n matrices, compute the product $A_1 A_2 \cdots A_n$
  - Matrix multiplication is associative so all parenthesizations yield the same product
  - But, do they all take the same amount of time?
  - E.g., say that n=3 and the dimensions are 10x100, 100x5, and 5x50

```
MATRIX-MULTIPLY(A, B)
1  if A.columns ≠ B.rows
2      error "incompatible dimensions"
3  else let C be a new A.rows × B.columns matrix
4      for i = 1 to A.rows
5          for j = 1 to B.columns
6              c_ij = 0
7              for k = 1 to A.columns
8                  c_ij = c_ij + a_ik · b_kj
9      return C
```

If A is p x q and B is q x r, then step 8 is executed pqr times

- Find the optimal multiplication order using dynamic programming

Matrix-Chain Multiplication

- The minimum cost of parenthesizing the product $A_i A_{i+1} \cdots A_j$ is

$$m[i,j] = \begin{cases} 0 & \text{if } i = j\,, \\ \min\limits_{i \le k < j} \{m[i,k] + m[k+1,j] + p_{i-1} p_k p_j\} & \text{if } i < j\,. \end{cases}$$

- Input is $p = \langle p_0, p_1, \ldots, p_n \rangle$ where $p_{i-1} \text{ x } p_i$ are the dimensions of matrix A_i
- Recursive algorithm would require exponential time
- But, how many subproblems have we defined?
  - One for each pair (i,j) so $\Theta(n^2)$ subproblems
  - Each subproblem depends on $\Theta(n)$ subproblems
- Output is a table m and a table s
  - Table m stores cost of each $m[i,j]$
  - Table s records index of k that achieved that cost

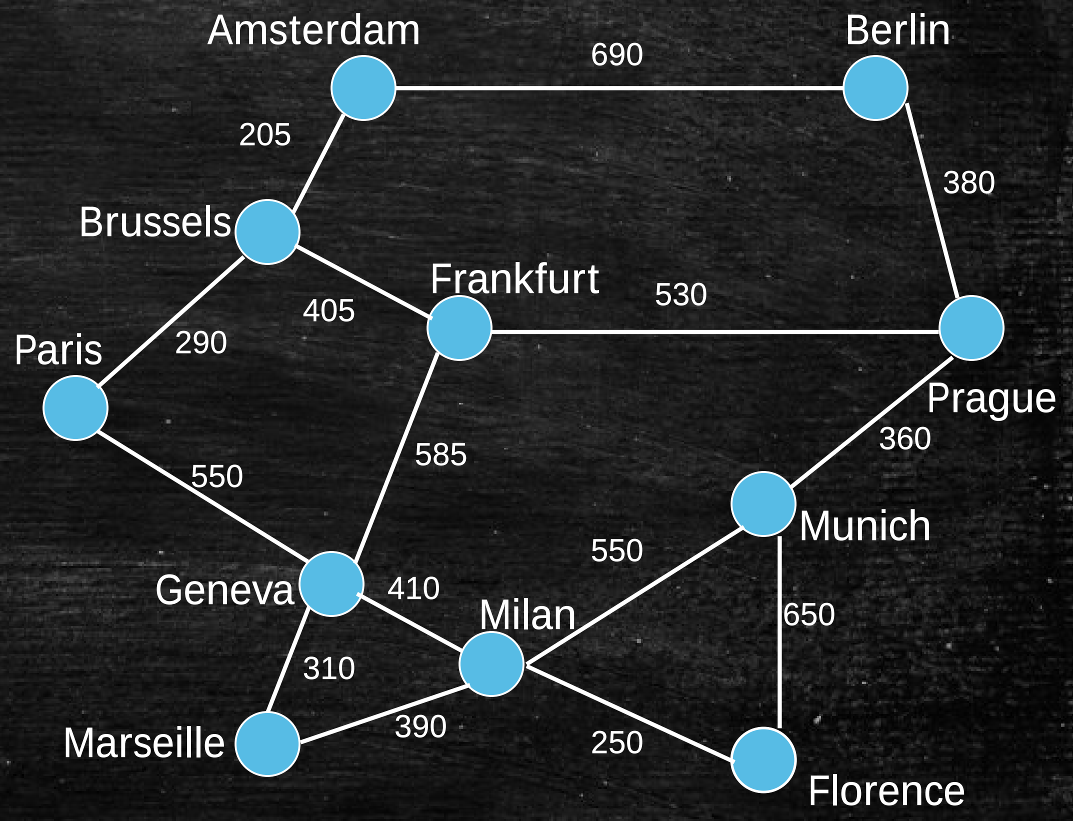
```
MATRIX-CHAIN-ORDER(p)
n = p.length − 1
let m[1..n, 1..n] and s[1..n − 1, 2..n] be new tables
for i = 1 to n
    m[i, i] = 0
for l = 2 to n                // l is the chain length
    for i = 1 to n − l + 1
        j = i + l − 1
        m[i, j] = ∞
        for k = i to j − 1
            q = m[i, k] + m[k + 1, j] + p_{i−1} p_k p_j
            if q < m[i, j]
                m[i, j] = q
                s[i, j] = k
return m and s
```

Today’s Lecture

- We now transition into graph algorithms
  - Minimum Spanning Tree
  - Breadth First Search
  - Depth First Search
  - Topological Sorting
  - Strongly Connected Components

Graphs

- Things to know:
  - **Path**
  - **Cycle**
  - **Sub-graph**
  - **Degree of a vertex**
  - **Maximum and minimum degree**
  - **Maximum number of edges**
  - **Connected components**
  - **Shortest path (weighted & unweighted)**
  - **Distance of two vertices**
  - **Tree (rooted tree)**
  - **Spanning tree of a graph**
  - **Acyclic graph**
  - **Bipartite graph**

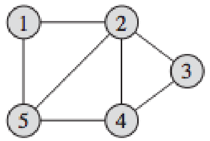


Definitions

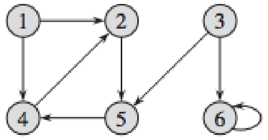
- Given A graph $G=(V,E)$, where
  - V is its vertex set, $|V|=n$,
  - E is its edge set, with $|E|=m=O(n^2)$
- If G is connected then for every pair of vertices u,v in G, there is path connecting them
- In an undirected graph, an edge $(u, v)=(v, u)$.
- In a directed graph, (u, v) is different from (v, u).
- In a weighted graph there are weights associated with edges and/or vertices.
- Running time of graph algorithms are usually expressed in terms of n or m.

Graph Representations

- Adjacency List
  - Good for sparse graphs
- Adjacency Matrix
  - Quick edge existence query
  - Simple



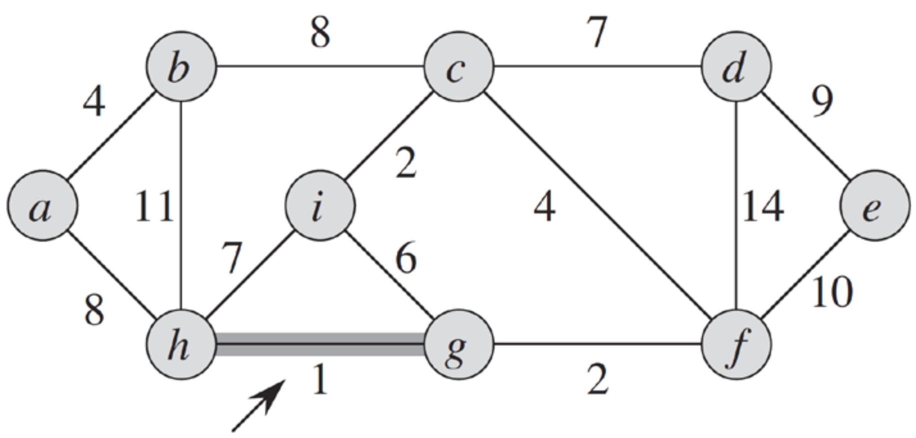
(a)



(a)

Minimum Spanning Tree of a Weighted Graph

- Let G=(V,E) be a graph on n vertices, m edges, and a weight w on edges in E.
- Sub-graph T=(V,E') with $E' \subseteq E$ with no cycles is a spanning tree
- The weight of T is the sum of the weights of its edges: $w(T) = \sum_{(u,v)\in E'} w(u,v)$

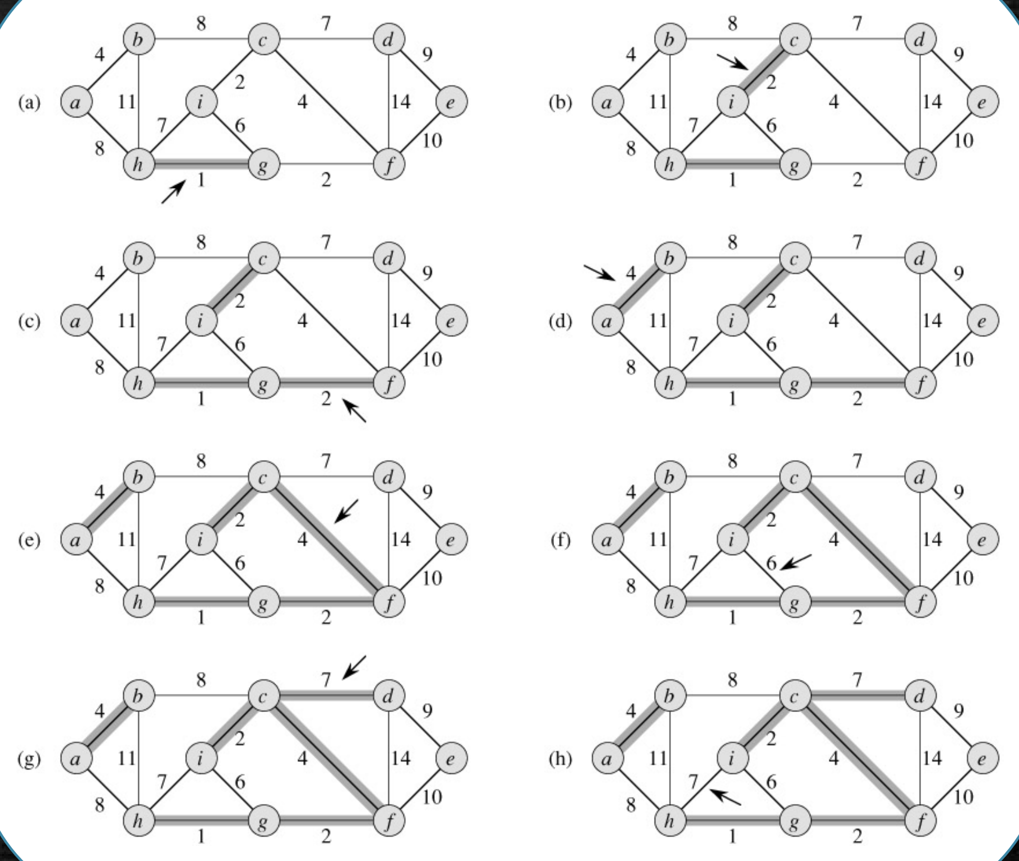


Set Operations

- We will use the following set operations:
  - Make-Set(v): creates a set containing element v, i.e., $\{v\}$
  - Find-Set(v): returns the set to which v belongs to
  - Union(u, v): creates a set which is the union of two sets, the one containing v and the one containing u

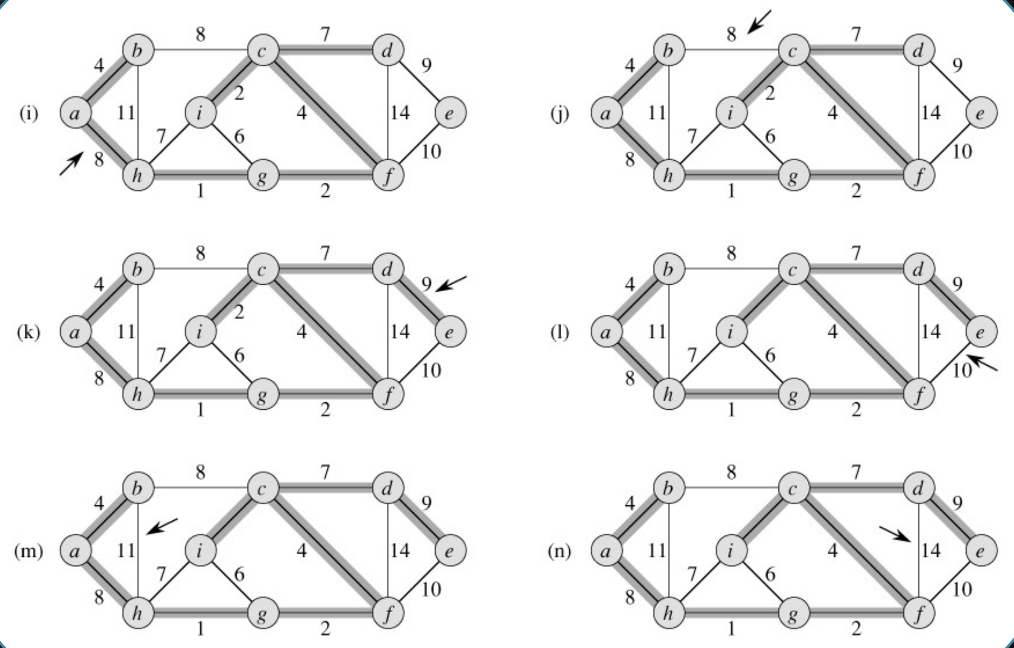
Kruskal’s Algorithm

```
MST-KRUSKAL(G, w)
A = ∅
for each vertex v ∈ G.V
    MAKE-SET(v)
sort the edges of G.E into nondecreasing order by weight w
for each edge (u, v) ∈ G.E, taken in nondecreasing order by weight
    if FIND-SET(u) ≠ FIND-SET(v)
        A = A ∪ {(u, v)}
        UNION(u, v)
return A
```



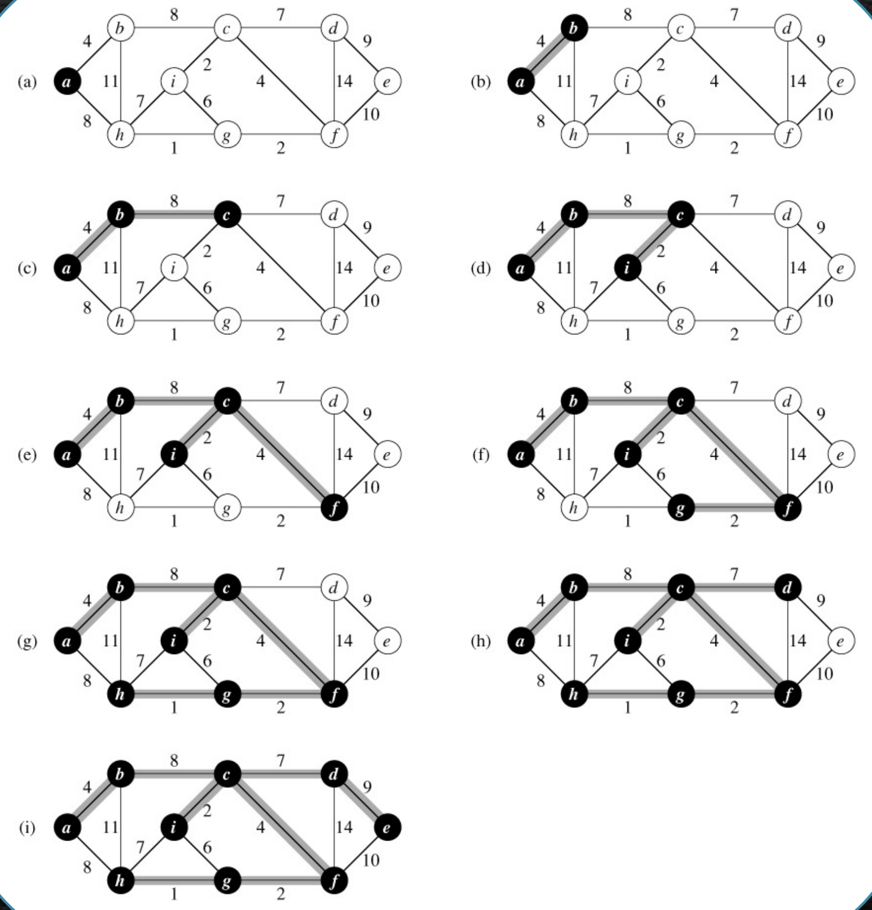
Kruskal's Algorithm

```
MST-KRUSKAL(G, w)
  A = ∅
  for each vertex v ∈ G.V
      MAKE-SET(v)
  sort the edges of G.E into nondecreasing order by weight w
  for each edge (u, v) ∈ G.E, taken in nondecreasing order by weight
      if FIND-SET(u) ≠ FIND-SET(v)
          A = A ∪ {(u, v)}
          UNION(u, v)
  return A
```



Prim’s Algorithm

```
MST-PRIM(G, w, r)
for each u ∈ G.V
    u.key = ∞
    u.π = NIL
r.key = 0
Q = G.V
while Q ≠ ∅
    u = EXTRACT-MIN(Q)
    for each v ∈ G.Adj[u]
        if v ∈ Q and w(u, v) < v.key
            v.π = u
            v.key = w(u, v)
```



Practice Problem

- Is the path between two vertices in an MST necessarily a shortest path between the two vertices in the full graph? Give a proof or a counterexample.
  - Answer: No it is not. For example, consider a graph that forms a single n-vertex cycle. The minimum spanning tree will remove just one edge (u, v), significantly increasing the distance between u and v

Breadth First Search (BFS)

- Given a graph $G = (V,E)$, BFS starts at some source vertex s and discovers which vertices are reachable from s
- The distance between a vertex v and s is the minimum number of edges on a path from s to v (the shortest path length)
- BFS discovers vertices in increasing order of distance, and hence can be used as an algorithm for computing shortest paths from s
- At any given time there is a frontier of vertices that have been discovered, but not yet processed.
- BFS first visits all vertices across the breadth of this frontier (hence the name)

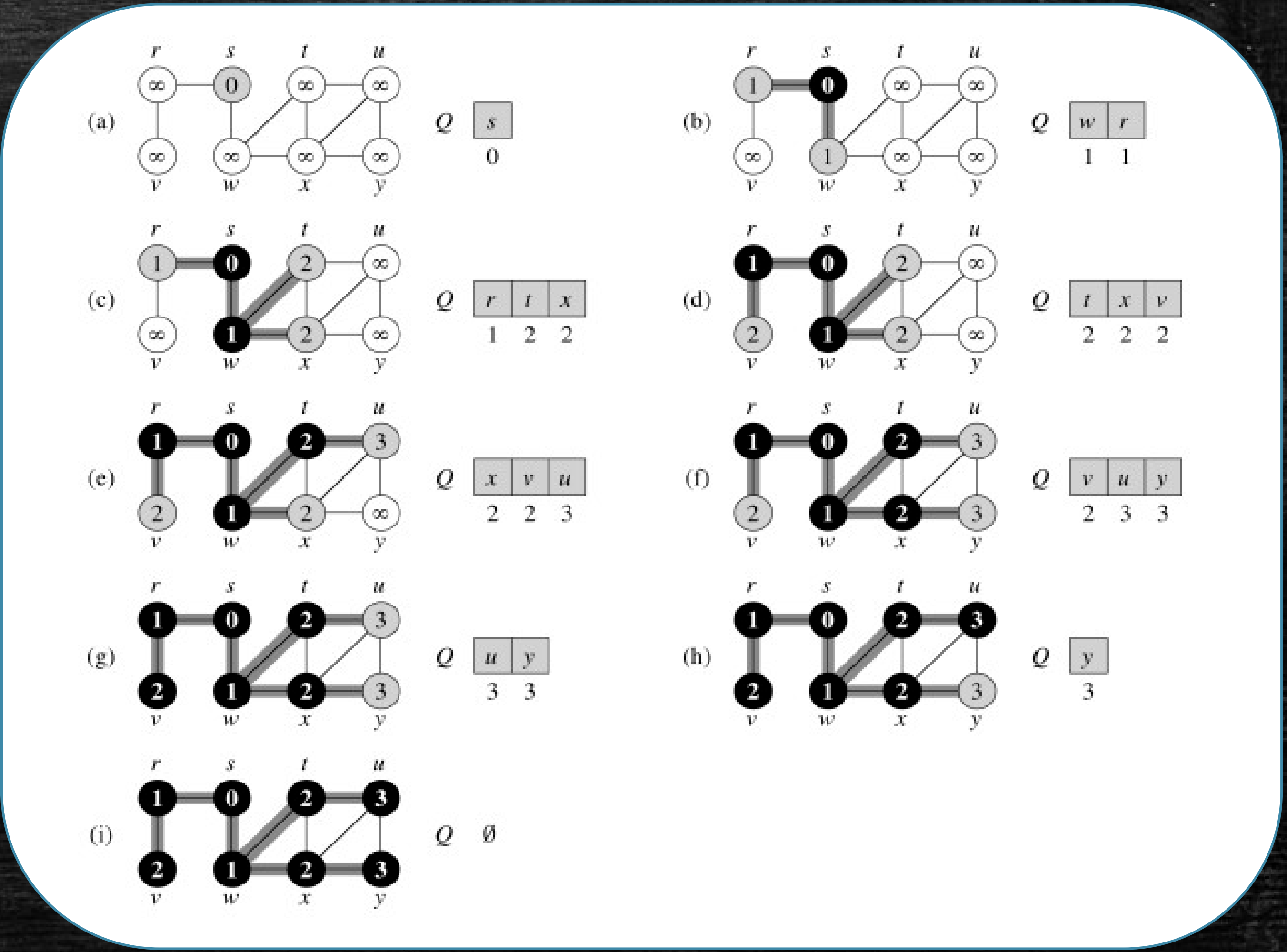
BFS: coloring

- We use a coloring procedure to show the status of BFS at each time:
  - Initially all vertices (except the source) are white: they are undiscovered
  - When a vertex is first discovered, it becomes gray (and is part of the frontier)
  - When a gray vertex is processed, it becomes black

BFS Algorithm

BFS (G, s)

```
for each u ∈ G.V − {s}
        u.color = WHITE
        u.d = ∞
        u.π = NIL
s.color = GRAY
s.d = 0
s.π = NIL
Q = ∅
ENQUEUE(Q, s)
while Q ≠ ∅
        u = DEQUEUE(Q)
        for each v ∈ G.Adj[u]
                if v.color == WHITE
                        v.color = GRAY
                        v.d = u.d + 1
                        v.π = u
                        ENQUEUE(Q, v)
        u.color = BLACK
```



BFS predecessor subgraph of G

For a graph $G = (V, E)$ with source s the predecessor subgraph of G is:

- $G_\pi = (V_\pi, E_\pi)$, where
- $V_\pi = \{v \in V : v.\pi \neq \mathrm{NIL}\} \cup \{s\}$
- $E_\pi = \{(v.\pi, v) : v \in V_\pi - \{s\}\}$

DFS Algorithm

DFS (G)

1. **for** each vertex $u \in G.V$
2. u.color = WHITE
3. u.π = NIL
4. time = 0
5. **for** each vertex $u \in G.V$
6. **if** u.color = WHITE
7. DFS-Visit(G,u)

DFS-Visit(G,u)

1. time = time + 1
2. u.d = time
3. u.color = GRAY
4. **for** each vertex $v \in G.Adj[u]$
5. **if** v.color == WHITE
6. v.π = u
7. DFS-Visit(G,v)
8. u.color = BLACK
9. time = time + 1
10. u.f = time

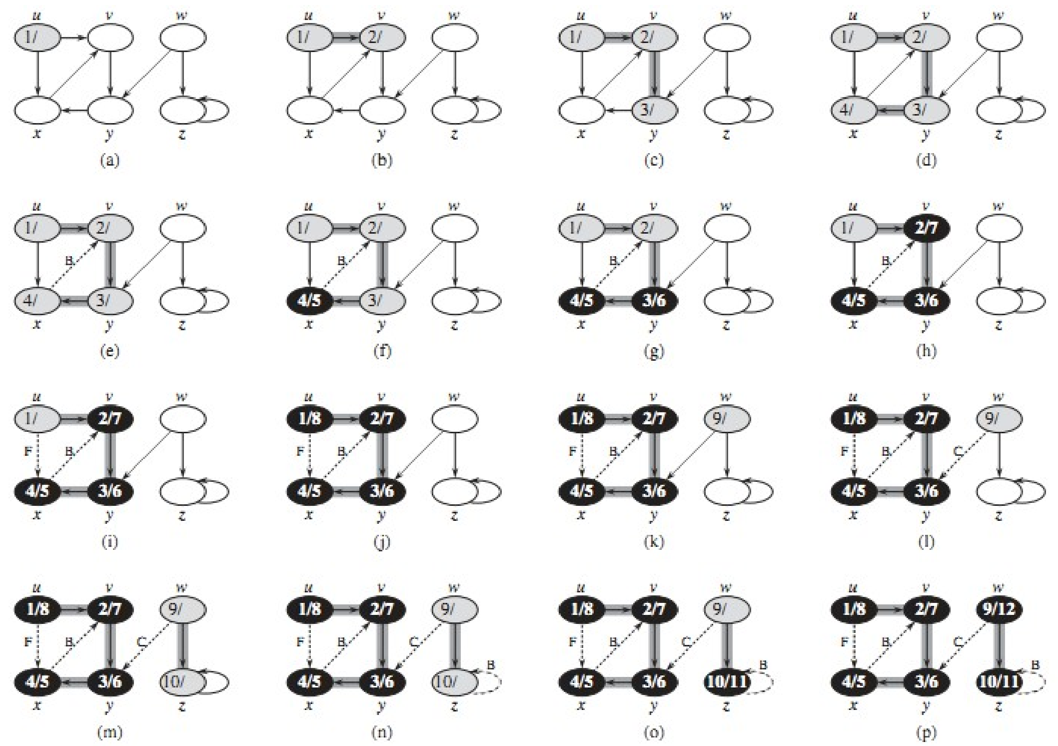


Figure 22.4 The progress of the depth-first-search algorithm DFS on a directed graph. As edges are explored by the algorithm, they are shown as either shaded (if they are tree edges) or dashed (otherwise). Nontree edges are labeled B, C, or F according to whether they are back, cross, or forward edges. Timestamps within vertices indicate discovery time/finishing times.