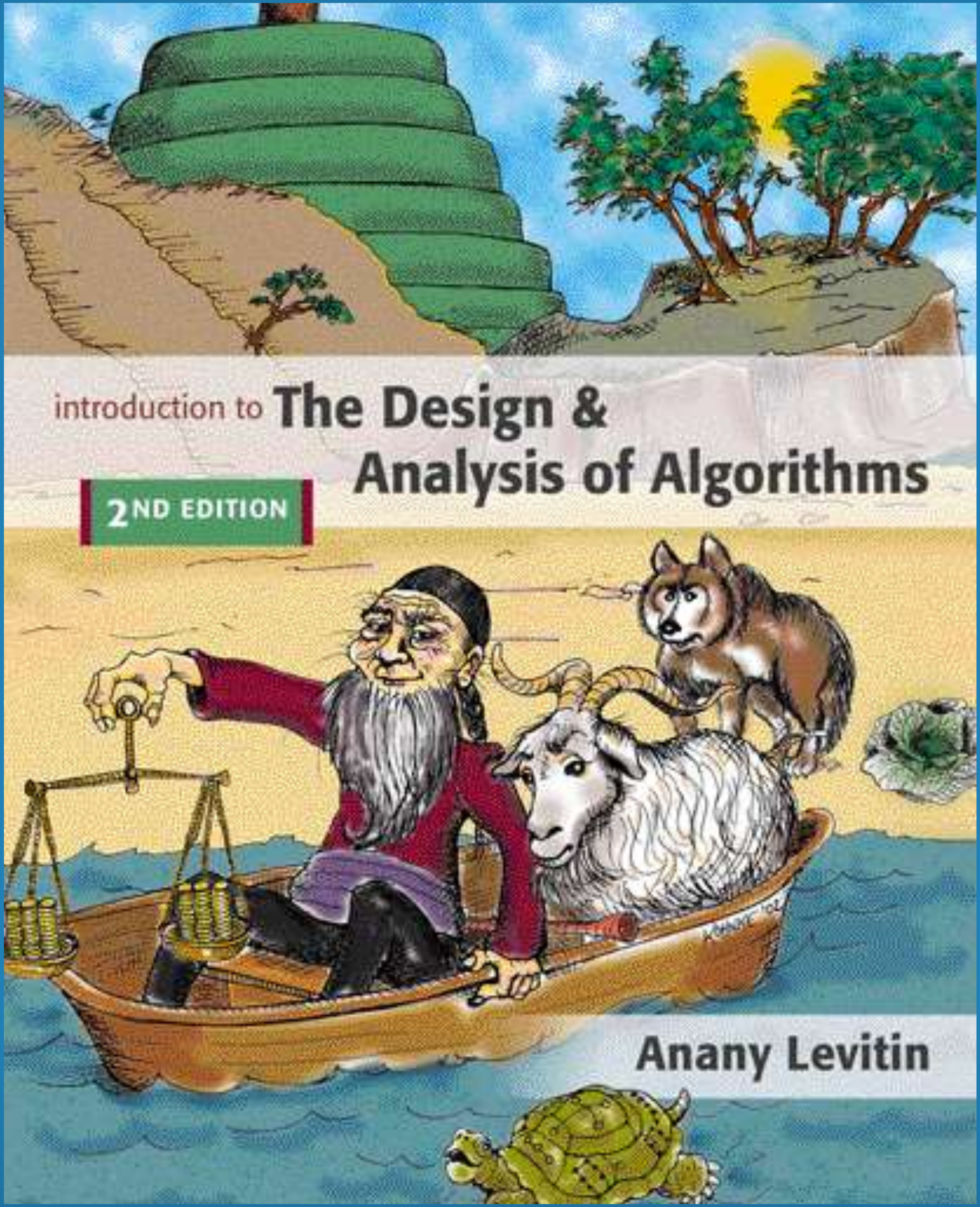
Chapter 5

Decrease-and-Conquer



Decrease-and-Conquer

1. Reduce problem instance to smaller instance of the same problem
2. Solve smaller instance
3. Extend solution of smaller instance to obtain solution to original instance

3 Types of Decrease and Conquer

- ***Decrease by a constant*** **(usually by 1):**
  - **insertion sort**
  - **graph traversal algorithms (DFS and BFS)**
  - **topological sorting**
  - **algorithms for generating permutations, subsets**
- ***Decrease by a constant factor*** **(usually by half)**
  - **binary search**
- ***Variable-size decrease***
  - **Euclid's algorithm**

Insertion Sort

To sort array A[0..*n*-1], sort A[0..*n*-2] recursively and then insert A[*n*-1] in its proper place among the sorted A[0..*n*-2]

- Usually implemented bottom up (nonrecursively)

Example: Sort 6, 4, 1, 8, 5

6 | <u>4</u> 1 8 5
4 6 | <u>1</u> 8 5
1 4 6 | <u>8</u> 5
1 4 6 8 | <u>5</u>
1 4 5 6 8

Pseudocode of Insertion Sort

ALGORITHM *InsertionSort*$(A[0..n-1])$

//Sorts a given array by insertion sort
//Input: An array $A[0..n-1]$ of n orderable elements
//Output: Array $A[0..n-1]$ sorted in nondecreasing order
for $i \leftarrow 1$ **to** $n-1$ **do**
    $v \leftarrow A[i]$
    $j \leftarrow i-1$
    while $j \geq 0$ **and** $A[j] > v$ **do**
        $A[j+1] \leftarrow A[j]$
        $j \leftarrow j-1$
    $A[j+1] \leftarrow v$

Analysis of Insertion Sort

- **Time efficiency**

  $C_{worst}(n) = n(n\text{-}1)/2 \in \Theta(n^2)$

  $C_{avg}(n) \approx n^2/4 \in \Theta(n^2)$

  $C_{best}(n) = n - 1 \in \Theta(n)$ **(also fast on almost sorted arrays)**

- **Space efficiency: in-place**

- **Stability: yes**

- **Best elementary sorting algorithm overall**

Graph Traversal

Many problems require processing all graph vertices (and edges) in systematic fashion

<u>Graph traversal algorithms</u>:

- Depth-first search (DFS)
- Breadth-first search (BFS)

Depth-First Search (DFS)

- Visits graph's vertices by always moving away from last visited vertex to an unvisited one, backtracks if no adjacent unvisited vertex is available.

- Recurisve or it uses a stack
  - a vertex is pushed onto the stack when it's reached for the first time
  - a vertex is popped off the stack when it becomes a dead end, i.e., when there is no adjacent unvisited vertex

- "Redraws" graph in tree-like fashion (with tree edges and back edges for undirected graph)

Pseudocode of DFS

ALGORITHM *DFS(G)*

```
//Implements a depth-first search traversal of a given graph
//Input: Graph G = ⟨V, E⟩
//Output: Graph G with its vertices marked with consecutive integers
//in the order they've been first encountered by the DFS traversal
mark each vertex in V with 0 as a mark of being "unvisited"
count ← 0
for each vertex v in V do
    if v is marked with 0
        dfs(v)
```

```
dfs(v)
//visits recursively all the unvisited vertices connected to vertex v by a path
//and numbers them in the order they are encountered
//via global variable count
count ← count + 1; mark v with count
for each vertex w in V adjacent to v do
    if w is marked with 0
        dfs(w)
```

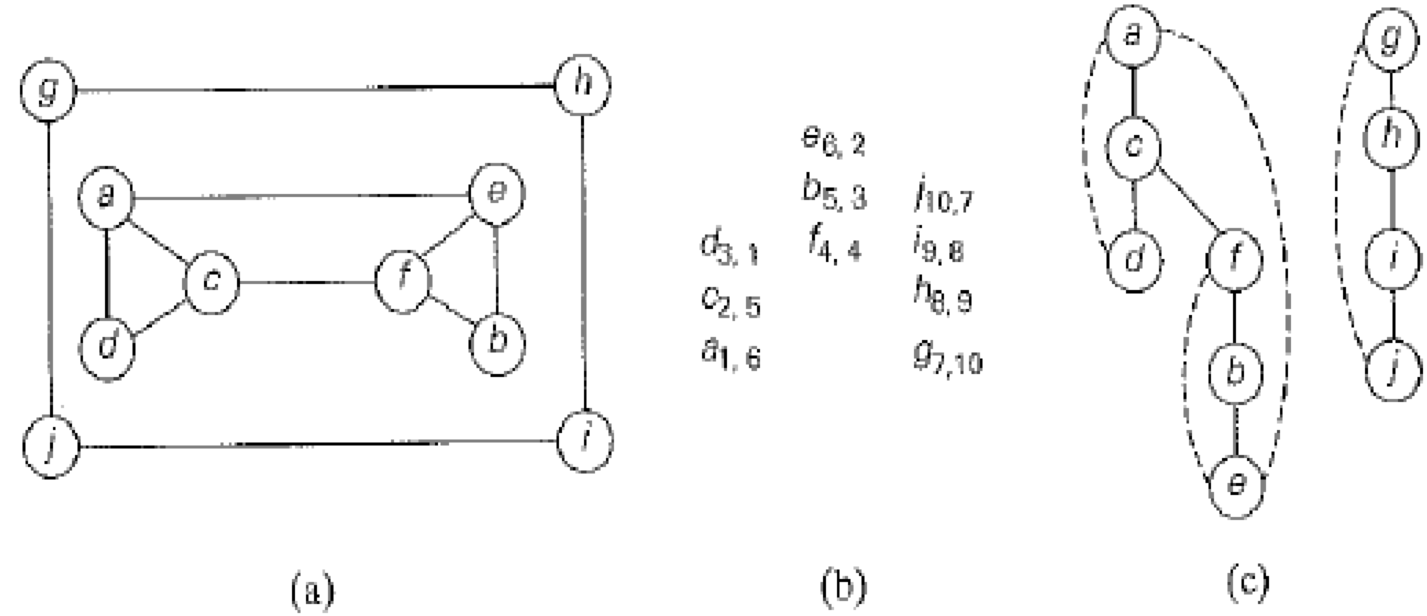
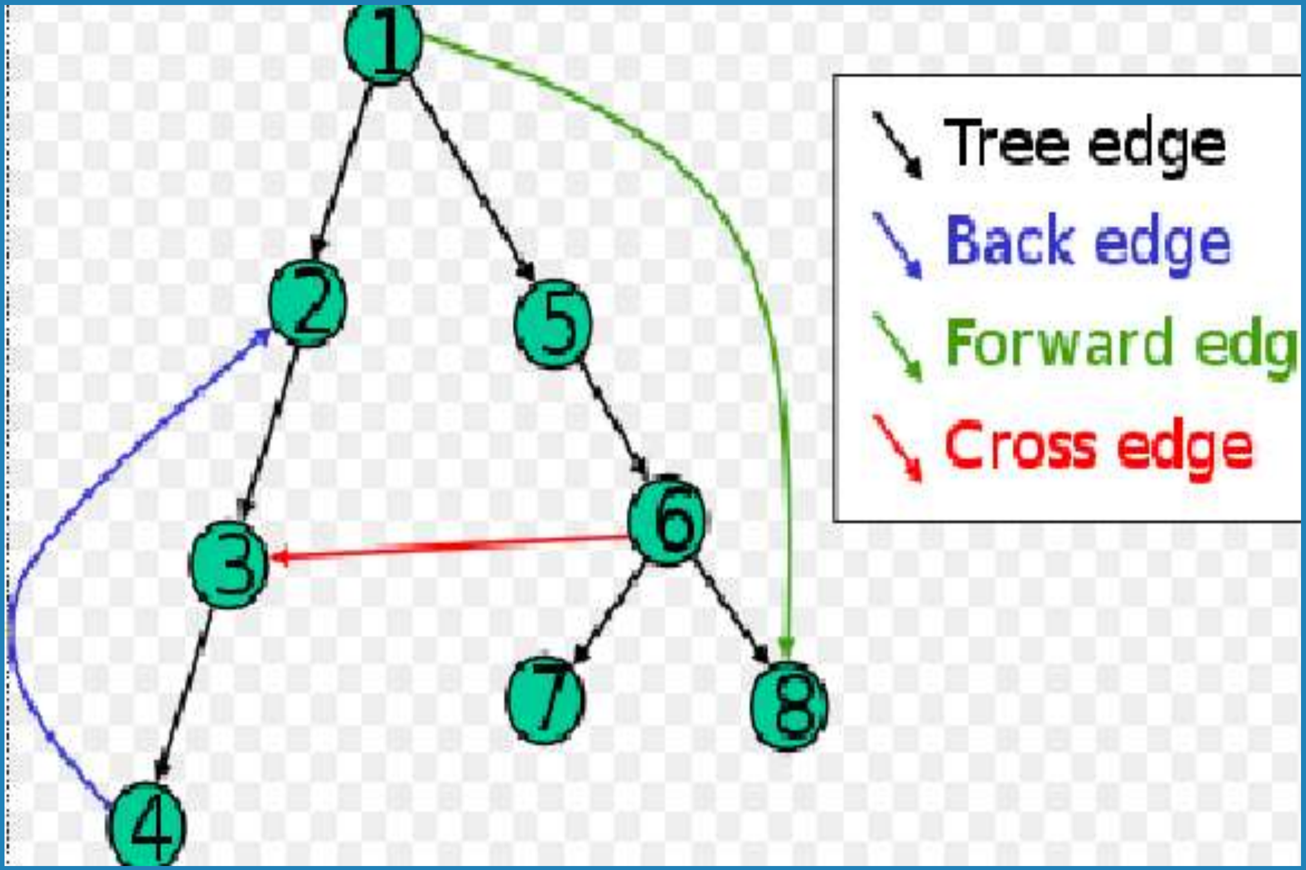


FIGURE 5.5 Example of a DFS traversal. (a) Graph. (b) Traversal's stack (the first subscript number indicates the order in which a vertex was visited, i.e., pushed onto the stack; the second one indicates the order in which it became a dead-end, i.e., popped off the stack). (c) DFS forest (with the tree edges shown with solid lines and the back edges shown with dashed lines).

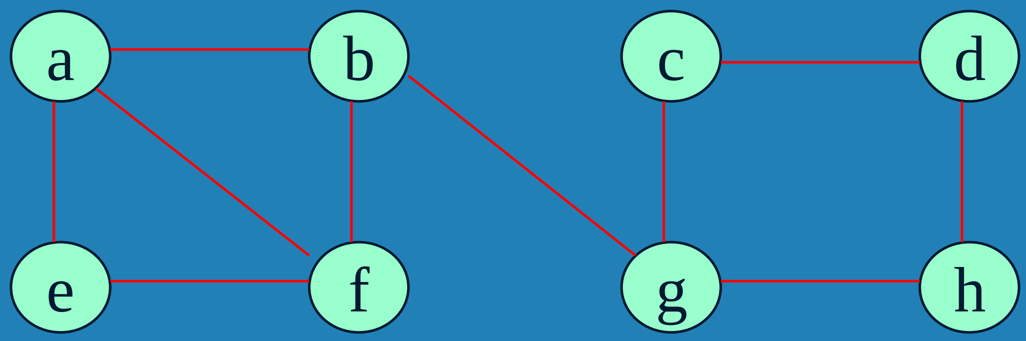
Notes on DFS

- DFS can be implemented with graphs represented as:
  - adjacency matrices: $\Theta(|V|^2)$.
  - adjacency lists: $\Theta(|V|+|E|)$.
- Yields two distinct ordering of vertices:
  - order in which vertices are first encountered (pushed onto stack)
  - order in which vertices become dead-ends (popped off stack)
- Applications:
  - checking connectivity, finding connected components
  - checking acyclicity (if no back edges)
  - finding articulation points

Contd…



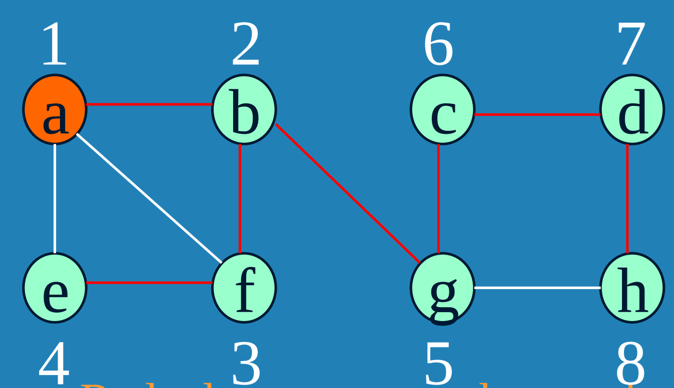
Example: DFS traversal of undirected graph



DFS traversal stack:

a
ab
abf
abfe
abf
ab
abg
abgc
abgcd
abgcdh
abgcd
…

DFS tree:



Red edges are tree edges and white edges are back edges.

Breadth-first search (BFS)

- Visits graph vertices by moving across to all the neighbors of the last visited vertex
- Instead of a stack, BFS uses a queue
- Similar to level-by-level tree traversal
- "Redraws" graph in tree-like fashion (with tree edges and cross edges for undirected graph)

Pseudocode of BFS

```
ALGORITHM BFS(G)
    //Implements a breadth-first search traversal of a given graph
    //Input: Graph G = ⟨V, E⟩
    //Output: Graph G with its vertices marked with consecutive integers
    //in the order they have been visited by the BFS traversal
    mark each vertex in V with 0 as a mark of being "unvisited"
    count ← 0
    for each vertex v in V do
        if v is marked with 0
            bfs(v)
```

```
bfs(v)
//visits all the unvisited vertices connected to vertex v by a path
//and assigns them the numbers in the order they are visited
//via global variable count
count ← count + 1;   mark v with count and initialize a queue with v
while the queue is not empty do
    for each vertex w in V adjacent to the front vertex do
        if w is marked with 0
            count ← count + 1;   mark w with count
            add w to the queue
    remove the front vertex from the queue
```

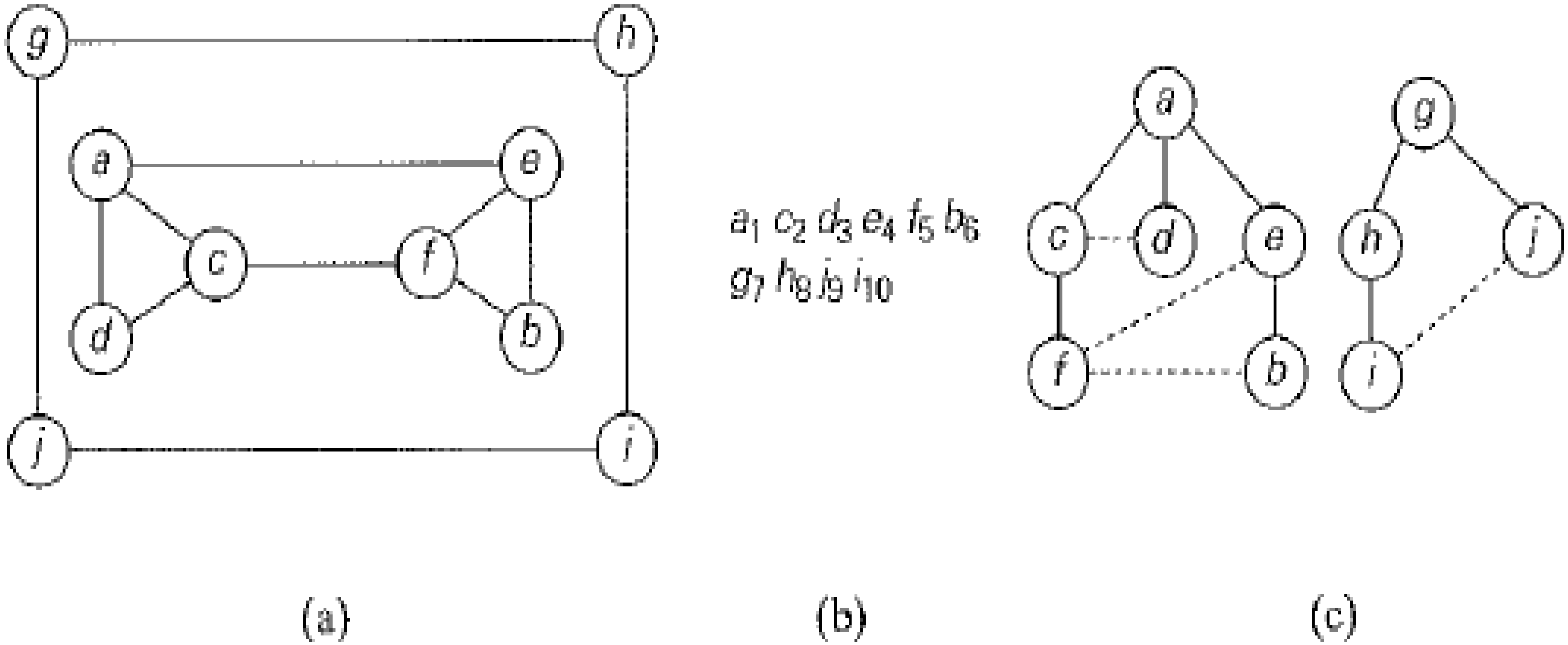


FIGURE 5.6 Example of a BFS traversal. (a) Graph. (b) Traversal's queue, with the numbers indicating the order in which the vertices were visited, i.e., added to (or removed from) the queue. (c) BFS forest (with the tree edges shown with solid lines and the cross edges shown with dotted lines).

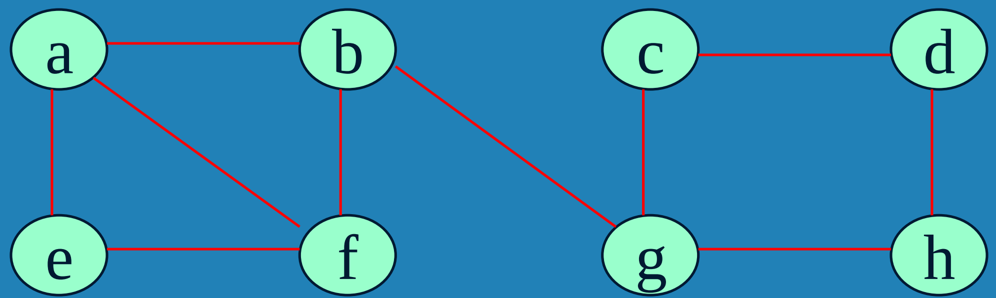
Notes on BFS

- BFS has same efficiency as DFS and can be implemented with graphs represented as:
  - adjacency matrices: $\Theta(|V|^2)$.
  - adjacency lists: $\Theta(|V|+|E|)$.

- Yields single ordering of vertices (order added/deleted from queue is the same)

- Applications: same as DFS, but can also find paths from a vertex to all other vertices with the smallest number of edges

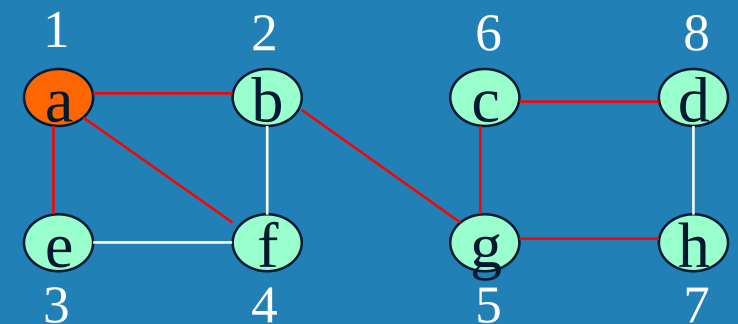
Example of BFS traversal of undirected graph



BFS traversal queue:

a
bef
efg
fg
g
ch
hd
d

BFS tree:

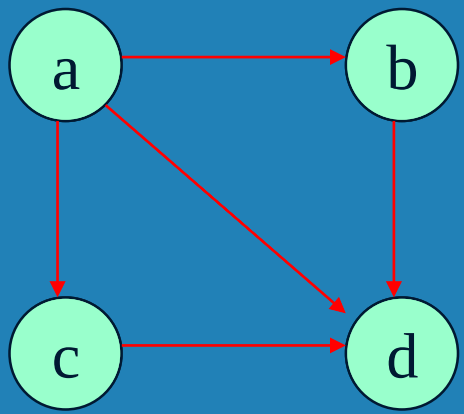


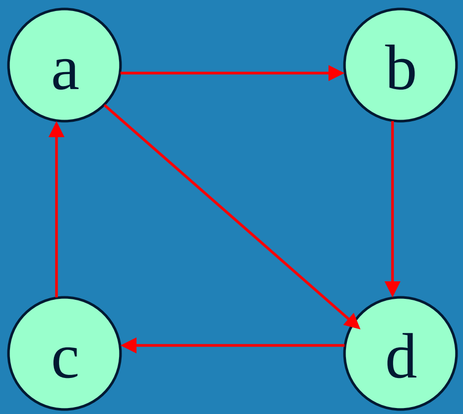
Red edges are tree edges and white edges are cross edges.

DAGs and Topological Sorting

A *dag*: a directed acyclic graph, i.e. a directed graph with no (directed) cycles

a dag





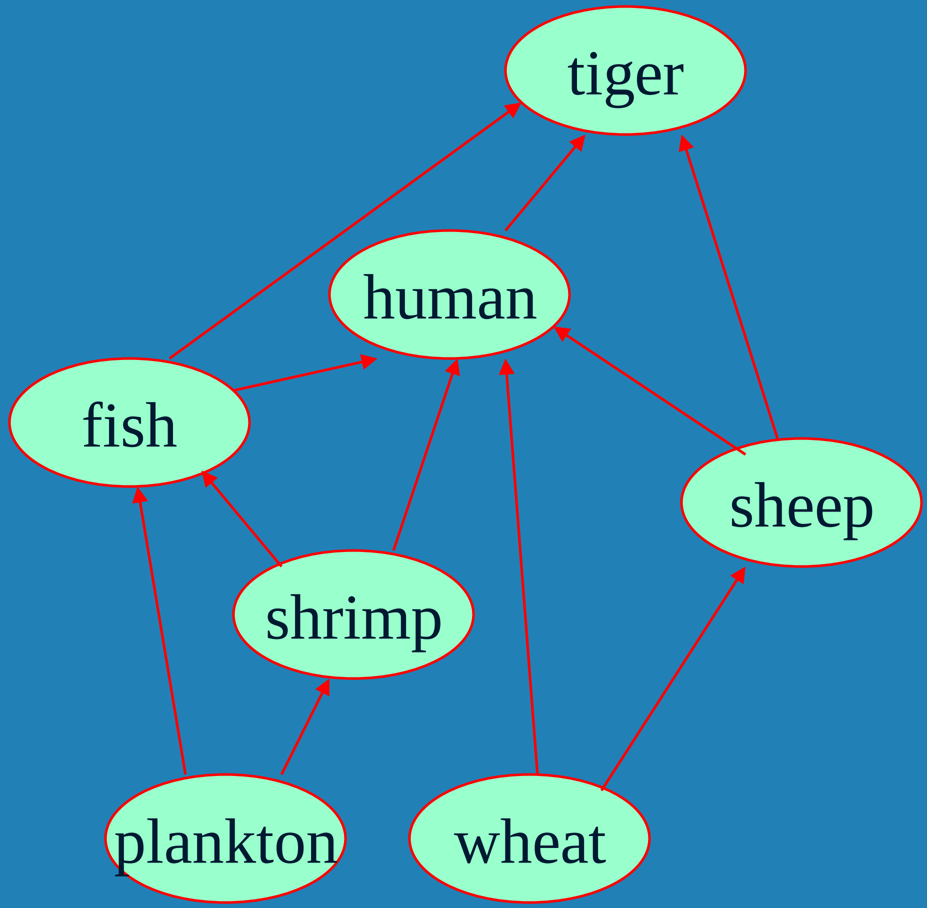
not a dag

Arise in modeling many problems that involve prerequisite constraints (construction projects, document version control)

Vertices of a dag can be linearly ordered so that for every edge its starting vertex is listed before its ending vertex (*topological sorting*). Being a dag is also a necessary condition for topological sorting to be possible.

Topological Sorting Example

Order the following items in a food chain



DFS-based algorithm for topological sorting

- Perform DFS traversal, noting the order vertices are popped off the traversal stack
- Reverse order solves topological sorting problem
- Back edges encountered?→ NOT a dag!

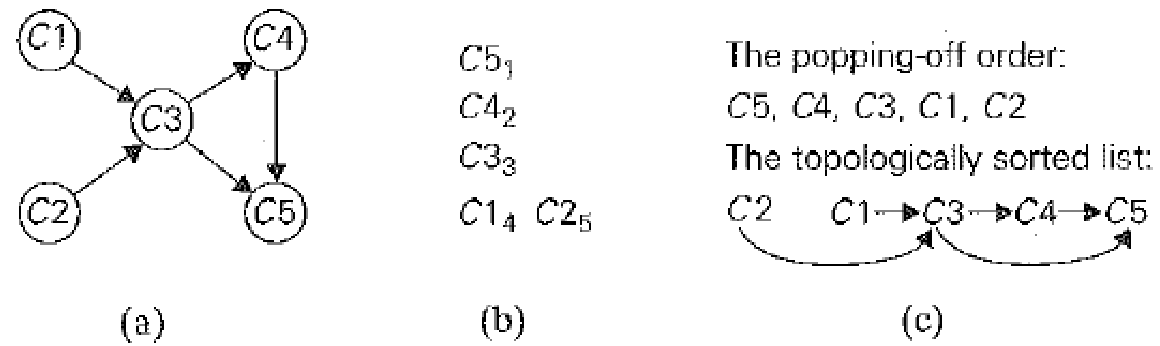
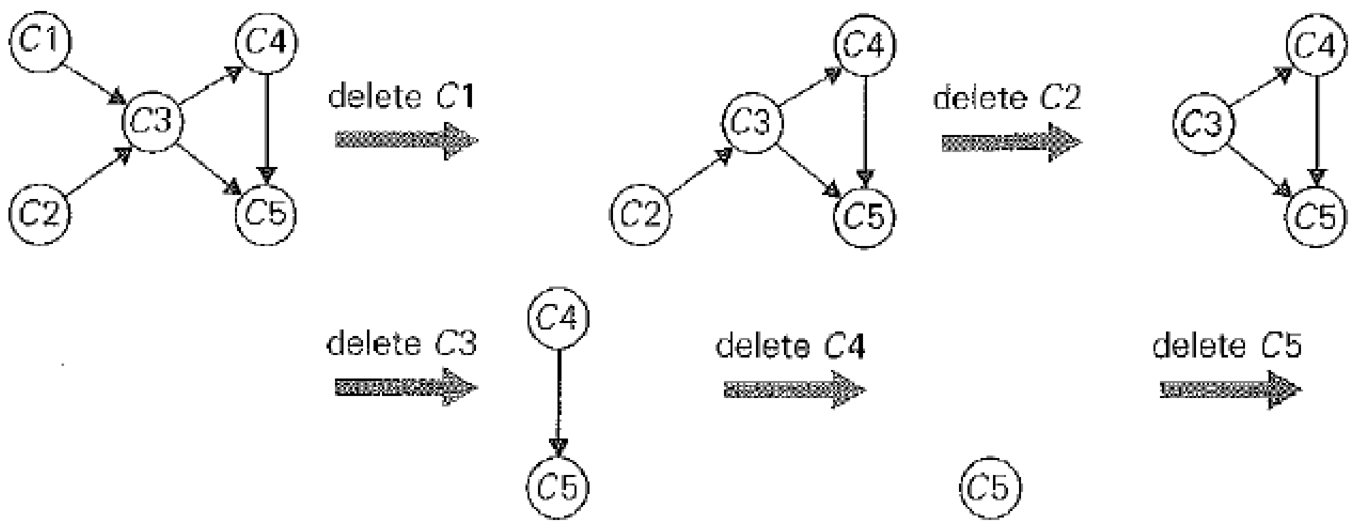


FIGURE 5.10 (a) Digraph for which the topological sorting problem needs to be solved. (b) DFS traversal stack with the subscript numbers indicating the popping-off order. (c) Solution to the problem.

Source removal algorithm

Repeatedly identify and remove a *source* (a vertex with no incoming edges) and all the edges incident to it until either no vertex is left or there is no source among the remaining vertices (not a dag)



The solution obtained is C1, C2, C3, C4, C5

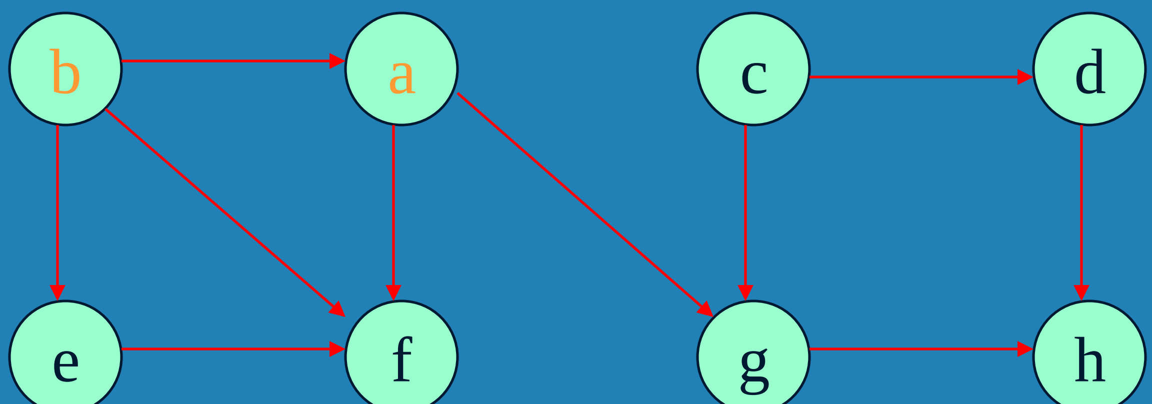
FIGURE 5.11 Illustration of the source-removal algorithm for the topological sorting problem. On each iteration, a vertex with no incoming edges is deleted from the digraph.

DFS-based Algorithm

<u>DFS-based algorithm for topological sorting</u>

- Perform DFS traversal, noting the order vertices are popped off the traversal stack
- Reverse order solves topological sorting problem
- Back edges encountered?→ NOT a dag!

Example:



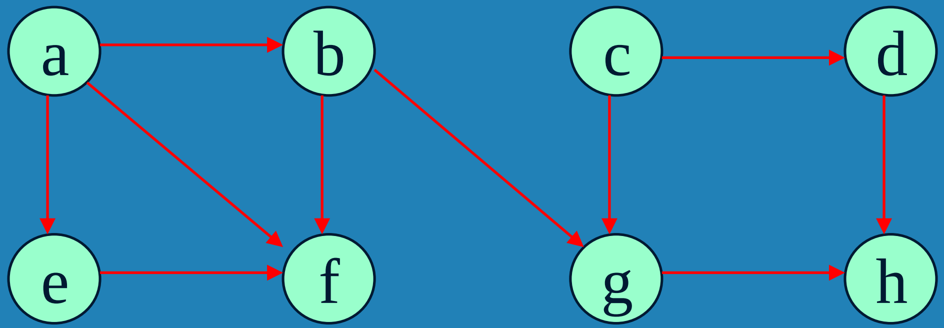
Efficiency: The same as that of DFS.

Source Removal Algorithm

Source removal algorithm

Repeatedly identify and remove a *source* (a vertex with no incoming edges) and all the edges incident to it until either no vertex is left or there is no source among the remaining vertices (not a dag)

Example:



Efficiency: same as efficiency of the DFS-based algorithm.

Algorithms for generating Combinatorial Objects

- **Generating Permutations -bottom up**
- **Start** 1
- **Insert 2 into 1 right to left** 1 2 2 1
- **Insert 3 into 1 2 right to left** 1 2 3 1 3 2 3 1 2
- **Insert 3 into 2 1 left to right** 3 2 1 2 3 1 2 1 3

Algorithm Johnson Trotter(n)

// Implements Johnson Trotter algorithm for generating permutations

//Input: A positive integer n

//Output: A list of all permutations of {1……. n}

Initialize the first permutation with $\overleftarrow{1}\ \overleftarrow{2} \ldots \overleftarrow{n}$

While the last permutation has a mobile element do

find its largest mobile element k

swap k and the adjacent integer k's arrow points to

reverse the direction of all the elements that are larger than k

add the new permutation to the list

ALGORITHM *JohnsonTrotter*(n)

//Implements Johnson-Trotter algorithm for generating permutations
//Input: A positive integer n
//Output: A list of all permutations of $\{1, \ldots, n\}$
initialize the first permutation with $\overleftarrow{1}\,\overleftarrow{2} \ldots \overleftarrow{n}$
while the last permutation has a mobile element **do**
    find its largest mobile element k
    swap k and the adjacent integer k's arrow points to
    reverse the direction of all the elements that are larger than k
    add the new permutation to the list

Continued……

Lexicographic order

If $a_{n-1} < a_n$ then transpose these last two elements.

If $a_{n-1} > a_n$ then check if $a_{n-2} < a_{n-1}$ → element just larger than n-2 in n-2th position and arrange the rest of the elements in ascending order

If $a_{n-1} < a_n$, we can simply transpose these last two elements.

example, 123 is followed by 132. If $a_{n-1} > a_n$, we have to engage a_{n-2}. If $a_{n-2} < a_{n-1}$, we should rearrange the last three elements by increasing the $(n-2)$th element as little as possible by putting there the next larger than a_{n-2} element chosen from a_{n-1} and a_n and filling positions $n-1$ and n with the remaining two of the three elements a_{n-2}, a_{n-1}, and a_n in increasing order. For example, 132 is followed by 213 while 231 is followed by 312. In general, we scan a current permutation from right to left looking for the first pair of consecutive elements a_i and a_{i+1} such that $a_i < a_{i+1}$ (and, hence, $a_{i+1} > \ldots > a_n$). Then we find the smallest element in the tail that is larger than a_i, i.e., $\min\{a_j \mid a_j > a_i,\ j > i\}$, and put it in position i; the positions from $i+1$ through n are filled with the elements $a_i, a_{i+1}, \ldots, a_n$, from which the element put in the ith position has been eliminated, in increasing order.