Breadth-First Search (BFS)

Explore graph moving across to all the neighbors of last visited vertex

Similar to level-by-level tree traversals

Instead of a stack (LIFO), breadth-first uses queue (FIFO)

Applications: same as DFS
BFS Example – undirected graph

Input Graph
(Adjacency matrix / linked list)

BFS forest
(Tree edge / Cross edge)

Queue
# BFS algorithm

**ALGORITHM BFS(G)**

//Input: Graph $G = \langle V, E \rangle$
//Output: Graph $G$ with its
//vertices marked with
//consecutive integers in the
//order they’ve been visited by
//BFS traversal

<table>
<thead>
<tr>
<th>ALGORITHM BFS(G)</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>count ← 0</code></td>
</tr>
<tr>
<td><code>mark each vertex with 0</code></td>
</tr>
<tr>
<td><code>for each vertex v in V do</code></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
</tbody>
</table>

**bfs(v)**

`count ← count + 1`
`mark v with count`
`initialize queue with v`

while queue is not empty do

for each vertex $w$ adjacent to the front vertex do

if $w$ is marked with 0

`count ← count + 1`
`mark w with count`
`add w to the end of the queue`

remove the front vertex from the queue
Example – Directed Graph

BFS traversal:
BFS Forest and Queue

How many cross edges? 4

Queue

BFS forest
Breadth-first search: Notes

BFS has same efficiency as DFS and can be implemented with graphs represented as:

• Adjacency matrices: $\Theta(|V|^2)$
• Adjacency linked lists: $\Theta(|V|+|E|)$

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

- **DFS**
  - Uses a stack
  - Yields two distinct ordering of vertices:
    - Preorder traversal: as vertices are first encountered (pushed onto stack)
    - Postorder traversal: as vertices become dead-ends (popped off stack)
  - Result in a DFS forest
    -- Tree edges, back edges, forward edges, and cross edges

- **BFS**
  - Uses a queue
  - Yields one ordering of vertices
  - Result in a BFS forest with tree edges and cross edges

- **Both DFS and BFS have efficiency**
  - Adjacency matrices: $\Theta(|V|^2)$
  - Adjacency linked lists: $\Theta(|V|+|E|)$
Directed Acyclic Graph (DAG)

A directed graph with no cycles

Arise in modeling many problems, eg:
  - prerequisite structure
  - food chains

A digraph is a DAG if its DFS forest has no back edge.
Example:

(a) $DG$

(b) $DFS$ forest

Not a DAG!
Topological Sorting

Problem: find an order of vertices such that for every edge in the graph, the starting vertex is listed before the ending vertex.

Example:

Five courses have the prerequisite relation shown in the left. Find the right order to take all of them sequentially.

Note: problem is solvable iff graph is DAG.
Topological Sorting Algorithms

DFS-based algorithm:
- DFS traversal: note the order with which the vertices are popped off stack (dead end)
- Reverse order solves topological sorting
- Back edges encountered? → NOT a DAG!

Source removal algorithm
- Repeatedly identify and remove a source vertex, i.e., a vertex that has no incoming edges
An Example: DFS-based Topological Sorting

C₂ C₁ C₃ C₄ C₅

The popping-off order: C₅, C₄, C₃, C₁, C₂
The topologically sorted list: C₂ C₁ C₃ C₄ C₅

𝛑(V+E) using adjacency linked lists
An Example: Source removal

C1 C2 C3 C4 C5

Θ(V+E) using adjacency linked lists

How to implement it?
Comparison

DFS based algorithm and the source removal algorithm may produce different valid topological order lists.

C2 → C1 → C3 → C4 → C5

C1 → C2 → C3 → C4 → C5
Variable-Size-Decrease: Binary Search Trees

- Every element in the left subtree is smaller than the root.
- Every element in the right subtree is larger than the root.
- Search a key in a binary search tree is reduced to search in a subtree in each iteration.
- The height of the subtree changes each time.

variable-size-decrease
Search a Key in a Binary Search Tree

Basic operation: key comparison

# of comparisons in the worst case: $h+1$

$$\log |V| \leq h \leq |V| - 1$$

Worst case: the tree degrades to a singly linked list $\Theta(|V|)$

Average case: $\Theta(\log |V|)$
Searching and insertion in binary search trees

Searching – straightforward

Insertion – search for key, insert at leaf where search terminated

Example 1: 5, 10, 3, 1, 7, 12, 9

Example 2: 4, 5, 7, 2, 1, 3, 6
Reading Assignments

Chapter 5.3, 5.4 and 5.5
The most well-known algorithm design strategy:

Divide instance of problem into two or more smaller instances of the same problem, ideally of about the same size

Solve smaller instances recursively

Obtain solution to original (larger) instance by combining these solutions obtained for the smaller instances
Divide-and-conquer technique

- A problem of size n
  - Subproblem 1 of size n/2
    - A solution to subproblem 1
  - Subproblem 2 of size n/2
    - A solution to subproblem 2

A solution to the original problem
An Example

Compute the sum of \( n \) numbers \( a_0, a_1, \ldots, a_{n-1} \).

Question: How to design a divide-and-conquer algorithm to solve this problem and what is its complexity?

Use divide-and-conquer strategy:

\[
a_0 + \ldots + a_{n-1} = (a_0 + \ldots + a_{\lfloor n/2 \rfloor-1}) + (a_{\lfloor n/2 \rfloor} + a_{n-1})
\]

What is the recurrence and the complexity of this recursive algorithm?

Does it improve the efficiency of the brute-force algorithm?
**General Divide and Conquer Recurrence**

\[ C(n) = 2C\left(\frac{n}{2}\right) + 1, \text{ for } n > 1 \quad C(1) = 0 \]

\[ T(n) = aT(n/b) + f(n) \quad \text{where } f(n) \in \Theta(n^k) \]

- \( a < b^k \) \quad T(n) \in \Theta(n^k)
- \( a = b^k \) \quad T(n) \in \Theta(n^k \log n)
- \( a > b^k \) \quad T(n) \in \Theta(n^{\log_b a})

\( a=2, \ b=2, \ k=0 \)
\( a>b^k, \ C(n) \text{ belongs to } \Theta(n) \)