Graph Traversal

Assume familiar with:
directed/undirected graph representation, connectivity, acyclicity, etc
Many problems require processing all graph vertices (and edges) in systematic fashion

Graph traversal algorithms:
• 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 unvisited one, backtracks if no adjacent unvisited vertex is available.

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)

- **Preconditions:** 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 this order, they've been located by the DFS traversal.

1. mark each vertex v in V with 0 as a mark of being "unvisited".
2. for each vertex v in V do
   - if v is marked with 0 do
     - dfs(v)

3. for each vertex v in V do
   - dfs(v)

4. Note: With recursively all the unvisited vertices connected to vertex v by a path.

<table>
<thead>
<tr>
<th>DFS traversal stack</th>
<th>DFS tree</th>
</tr>
</thead>
<tbody>
<tr>
<td>h d b f c a e g i</td>
<td>g h i</td>
</tr>
</tbody>
</table>

### Example: DFS traversal of undirected graph

- **DFS traversal stack:** h d b f c a e g i
- **DFS tree:** g h i

### Notes on DFS

1. **DFS can be implemented with graphs represented as:**
   - adjacency matrices: Θ(V^2)
   - adjacency lists: Θ(V + |E|)

2. **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)

3. **Applications:**
   - checking connectivity, finding connected components
   - checking acyclicity
   - finding articulation points and biconnected components
   - searching state space of problems for solution (AI)
**Breadth-first search (BFS)**

- Visits graph vertices by moving across to all the neighbors of 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, s)

// Implements a breadth-first search traversal of a given graph
// Input: Graph G = (V, E), where V is the set of vertices and E is the set of edges
// Output: Graph G' = (V, E'), where V is the set of vertices and E' is the set of edges

1. Mark all vertices in V as unvisited.
2. Enqueue the source vertex s.
3. while the queue is not empty do
   1. Dequeue a vertex v from the queue.
   2. Mark v as visited.
   3. For each vertex w adjacent to v do
      1. if w is unvisited then
         1. Mark w as visited.
         2. Enqueue w.
```

---

**Example of BFS traversal of undirected graph**

**BFS traversal queue:**

```
BFS tree:
```

---
Notes on BFS

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

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

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

---

Dags and Topological Sorting

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

\[ \begin{align*}
\text{a dag:} & \quad a & \quad b & \quad c & \quad d \\
\text{not a dag:} & \quad a & \quad b & \quad c & \quad d \\
\end{align*} \]

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

Vertices of a dag can be linearly ordered so that for every edge \((u,v)\), vertex \(u\) comes before vertex \(v\) (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:

\[ \begin{align*}
\text{tiger} & \quad \text{human} & \quad \text{sheep} & \quad \text{shrimp} & \quad \text{plankton} & \quad \text{fish} & \quad \text{wheat} \\
\end{align*} \]
**Design and Analysis of Algorithms**

### DFS-based Algorithm

**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

**Example:**

```
   a   b   c   d
  / \\   / \\ /   \
 e  f  g  h
```

### 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 (problem is solved) or there is no source among remaining vertices (not a dag)

**Example:**

```
   a   b   c   d
  / \\   / \\ /   \
 e  f  g  h
```

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

### Decrease-by-Constant-Factor Algorithms

In this variation of decrease-and-conquer, instance size is reduced by the same factor (typically, 2)

**Examples:**
- Binary search and the method of bisection
- Exponentiation by squaring
- Multiplication à la russe (Russian peasant method)
- Fake-coin puzzle
- Josephus problem
Exponentiation by Squaring

The problem: Compute $a^n$ where $a$ is a nonnegative integer.

The problem can be solved by applying recursively the formulas:

- For even values of $n$
  $a^n = (a^{n/2})^2$ if $n > 0$ and $a^0 = 1$
- For odd values of $n$
  $a^n = (a^{(n+1)/2})^2 a$

Recurrence: $M(n) = M\left\lfloor n/2 \right\rfloor + f(n)$, where $f(n) = 1$ or $2$.
$M(0) = 0$

Master Theorem: $M(n) = \Theta(n^\log_2 3)$ where $k = \lceil \log_2\left(\lceil\log_2 a\rceil + 1\right)\rceil$

Fake-Coin Puzzle (simpler version)

There are $n$ identically looking coins one of which is fake. There is a balance scale but there are no weights; the scale can tell whether two sets of coins weigh the same and, if not, which of the two sets is heavier (but not by how much). Design an efficient algorithm for detecting the fake coin. Assume that the fake coin is known to be lighter than the genuine ones.

Decrease by factor 2 algorithm

Decrease by factor 3 algorithm

Russian Peasant Multiplication

The problem: Compute the product of two positive integers.

Can be solved by a decrease-by-half algorithm based on the following formulas:

- For even values of $n$:
  $n \times m = \frac{n}{2} \times 2m$
- For odd values of $n$:
  $n \times m = \frac{n-1}{2} \times 2m + m$ if $n > 1$ and $m$ if $n = 1$
Example of Russian Peasant Multiplication

<table>
<thead>
<tr>
<th>n</th>
<th>m</th>
<th>Why:</th>
</tr>
</thead>
<tbody>
<tr>
<td>20</td>
<td>26</td>
<td>Good for Peasants - Don’t have to memorize multiplication tables</td>
</tr>
<tr>
<td>10</td>
<td>52</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>104</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>208</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>416</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>416</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>832</td>
<td></td>
</tr>
</tbody>
</table>

Note: Method reduces to adding m’s values corresponding to odd n’s.

Why: Good for Peasants – Don’t have to memorize multiplication tables
     Good for Computers – Really easy to double and divide in half (bit shifts)