CSE 373: Data Structures and Algorithms

Lecture 19: Graphs III
Depth-first search

- **depth-first search (DFS)**: finds a path between two vertices by exploring each possible path as many steps as possible before backtracking
  
  – often implemented recursively

![Depth-first search diagram](image)
DFS template

• Pseudo-code for depth-first template:

\[
dfs(\text{Vertex } v):
\begin{align*}
& \text{mark } v \text{ as visited} \\
& \text{for each unvisited neighbor } v_i \text{ of } v \\
& \text{where there is an edge from } v \text{ to } v_i:\n& \quad \text{if( } !v_i.\text{visited } ) \\
& \quad \quad \text{dfs}(v_i)
\end{align*}
\]
• **breadth-first search (BFS):** finds a path between two nodes by taking one step down all paths and then immediately backtracking
  
  – often implemented by maintaining a list or queue of vertices to visit
  
  – BFS always returns the path with the fewest edges between the start and the goal vertices
BFS example

• All BFS paths from A to others (assumes ABC edge order)
  – A
  – A -> B
  – A -> C
  – A -> E
  – A -> B -> D
  – A -> B -> F
  – A -> C -> G

• What are the paths that BFS did not find?
BFS pseudocode

- Pseudo-code for breadth-first search:
  
  `bfs(v1, v2):`
  
  `List := {v1}.`
  
  `mark v1 as visited.`
  
  `while List not empty:`
  
  `v := List.removeFirst().`
  
  `if v is v2:`
  
  `  path is found.`
  
  `for each unvisited neighbor vᵢ of v where there is an edge from v to vᵢ:`
  
  `  mark vᵢ as visited.`
  
  `  List.addLast(vᵢ).`
  
  `path is not found.`
**BFS observations**

- **optimality:**
  - in unweighted graphs, optimal. (fewest edges = best)
  - In weighted graphs, not optimal. (path with fewest edges might not have the lowest weight)

- **disadvantage:** harder to reconstruct what the actual path is once you find it
  - conceptually, BFS is exploring many possible paths in parallel, so it's not easy to store a Path array/list in progress

- **observation:** any particular vertex is only part of one partial path at a time
  - We can keep track of the path by storing *predecessors* for each vertex (references to the previous vertex in that path)
Another BFS example

• Using BFS, find a path from BOS to SFO.
DFS, BFS runtime

• What is the expected runtime of DFS, in terms of the number of vertices $V$ and the number of edges $E$?

• What is the expected runtime of BFS, in terms of the number of vertices $V$ and the number of edges $E$?

• Answer: $O(|V| + |E|)$
  – each algorithm must potentially visit every node and/or examine every edge once.
  – why not $O(|V| \times |E|)$?

• What is the space complexity of each algorithm?
Implementing graphs
Implementing a graph

• If we wanted to program an actual data structure to represent a graph, what information would we need to store?
  – for each vertex?
  – for each edge?

• What kinds of questions would we want to be able to answer quickly:
  – about a vertex?
  – about its edges / neighbors?
  – about paths?
  – about what edges exist in the graph?

• We'll explore three common graph implementation strategies:
  – edge list, adjacency list, adjacency matrix
Edge list

• **edge list**: an unordered list of all edges in the graph

• **advantages**
  – easy to loop/iterate over all edges

• **disadvantages**
  – hard to tell if an edge exists from A to B
  – hard to tell how many edges a vertex touches (its degree)

<table>
<thead>
<tr>
<th>1</th>
<th>1</th>
<th>1</th>
<th>2</th>
<th>2</th>
<th>3</th>
<th>5</th>
<th>5</th>
<th>5</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>3</td>
<td>4</td>
<td>6</td>
<td>7</td>
<td>4</td>
<td>4</td>
</tr>
</tbody>
</table>
**Adjacency matrix**

- **adjacency matrix**: an $n \times n$ matrix where:
  - the nondiagonal entry $a_{ij}$ is the number of edges joining vertex $i$ and vertex $j$ (or the weight of the edge joining vertex $i$ and vertex $j$)
  - the diagonal entry $a_{ii}$ corresponds to the number of loops (self-connecting edges) at vertex $i$
Pros/cons of Adj. matrix

• *advantage*: fast to tell whether edge exists between any two vertices $i$ and $j$ (and to get its weight)
• *disadvantage*: consumes a lot of memory on sparse graphs (ones with few edges)
Adjacency matrix example

- The graph at right has the following adjacency matrix:
  - How do we figure out the degree of a given vertex?
  - How do we find out whether an edge exists from A to B?
  - How could we look for loops in the graph?

\[
\begin{array}{cccccccc}
1 & 2 & 3 & 4 & 5 & 6 & 7 \\
\hline
1 & 0 & 1 & 0 & 0 & 1 & 1 & 0 \\
2 & 1 & 0 & 1 & 0 & 0 & 0 & 1 \\
3 & 0 & 1 & 0 & 1 & 0 & 0 & 0 \\
4 & 0 & 0 & 1 & 0 & 1 & 0 & 1 \\
5 & 1 & 0 & 0 & 1 & 0 & 1 & 1 \\
6 & 1 & 0 & 0 & 0 & 1 & 0 & 0 \\
7 & 0 & 1 & 0 & 1 & 1 & 0 & 0 \\
\end{array}
\]
Adjacency lists

- **adjacency list**: stores edges as individual linked lists of references to each vertex's neighbors
  - generally, no information needs to be stored in the edges, only in nodes, these arrays can simply be pointers to other nodes and thus represent edges with little memory requirement
Pros/cons of adjacency list

- **advantage**: new nodes can be added to the graph easily, and they can be connected with existing nodes simply by adding elements to the appropriate arrays; "who are my neighbors" easily answered
- **disadvantage**: determining whether an edge exists between two nodes requires $O(n)$ time, where $n$ is the average number of incident edges per node
Adjacency list example

• The graph at right has the following adjacency list:
  – How do we figure out the degree of a given vertex?
  – How do we find out whether an edge exists from A to B?
  – How could we look for loops in the graph?
# Runtime table

- _n_ vertices, _m_ edges
- no parallel edges
- no self-loops

<table>
<thead>
<tr>
<th></th>
<th>Edge List</th>
<th>Adjacency List</th>
<th>Adjacency Matrix</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Space</strong></td>
<td><em>n</em> + <em>m</em></td>
<td><em>n</em> + <em>m</em></td>
<td><em>n</em>^2</td>
</tr>
<tr>
<td>Finding all adjacent</td>
<td><em>m</em></td>
<td>deg(v)</td>
<td><em>n</em></td>
</tr>
<tr>
<td>vertices to v</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Determining if v is</td>
<td><em>m</em></td>
<td>min(deg(v), deg(w))</td>
<td>1</td>
</tr>
<tr>
<td>adjacent to w</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>inserting a vertex</td>
<td>1</td>
<td>1</td>
<td><em>n</em>^2</td>
</tr>
<tr>
<td>inserting an edge</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>removing vertex v</td>
<td><em>m</em></td>
<td>deg(v)</td>
<td><em>n</em>^2</td>
</tr>
<tr>
<td>removing an edge</td>
<td><em>m</em></td>
<td>deg(v)</td>
<td>1</td>
</tr>
</tbody>
</table>