Graphs

Analysis of Algorithms
Graphs

• A graph is a set of vertices connected pairwise by edges

• Why study graph algorithms?
  – Thousands of practical applications
  – Hundreds of graph algorithms known
  – Interesting and broadly useful abstraction
  – Challenging branch of computer science and discrete math
What are Graphs Used For?

• Political blogosphere

![Political blogosphere and the 2004 U.S. Elections Divided They Blog, Adamic and Clance, 2005](image1)

• Logical Implications

![Logical Implications](image2)

• Wordnets

![Wordnets](image3)

• International conflicts

![International conflicts](image4)
<table>
<thead>
<tr>
<th>Application</th>
<th>Vertex Represents</th>
<th>Edge Represents</th>
</tr>
</thead>
<tbody>
<tr>
<td>Communication</td>
<td>Phone, computer</td>
<td>Fiber optic cable</td>
</tr>
<tr>
<td>Electronic circuit</td>
<td>Gate, register, CPU</td>
<td>Wire</td>
</tr>
<tr>
<td>Mechanical part</td>
<td>Joint</td>
<td>Rod, beam, spring</td>
</tr>
<tr>
<td>Finance</td>
<td>Stock, currency</td>
<td>Transaction</td>
</tr>
<tr>
<td>Transportation</td>
<td>Intersection, airport</td>
<td>Highway, route</td>
</tr>
<tr>
<td>Internet</td>
<td>Network</td>
<td>Connection</td>
</tr>
<tr>
<td>Game</td>
<td>Board position</td>
<td>Legal move</td>
</tr>
<tr>
<td>Social relationship</td>
<td>Person</td>
<td>Friendship</td>
</tr>
<tr>
<td>Neural network</td>
<td>Neuron</td>
<td>Synapse</td>
</tr>
<tr>
<td>Protein network</td>
<td>Protein</td>
<td>Interaction</td>
</tr>
<tr>
<td>Chemical compound</td>
<td>Molecule</td>
<td>Bond</td>
</tr>
</tbody>
</table>
# What are Graphs Used For?

<table>
<thead>
<tr>
<th>Application</th>
<th>Vertex Represents</th>
<th>Edge Represents</th>
</tr>
</thead>
<tbody>
<tr>
<td>WWW</td>
<td>Web page</td>
<td>Link</td>
</tr>
<tr>
<td>WordNet</td>
<td>Set of synonyms</td>
<td>Hypernym</td>
</tr>
<tr>
<td>Scheduling</td>
<td>Task</td>
<td>Precedent constraint</td>
</tr>
<tr>
<td>Finance</td>
<td>Bank</td>
<td>Transaction</td>
</tr>
<tr>
<td>Communication</td>
<td>Cell phone</td>
<td>Placed a call</td>
</tr>
<tr>
<td>Infectious disease</td>
<td>Person</td>
<td>Infection</td>
</tr>
<tr>
<td>Game</td>
<td>Board position</td>
<td>Legal move</td>
</tr>
<tr>
<td>Citation</td>
<td>Journal Article</td>
<td>Refers to</td>
</tr>
<tr>
<td>OOP</td>
<td>Class</td>
<td>Hierarchy</td>
</tr>
<tr>
<td>Control flow</td>
<td>Code block</td>
<td>Transfer to</td>
</tr>
<tr>
<td>Food chain</td>
<td>Species</td>
<td>Eats</td>
</tr>
</tbody>
</table>
Graph Terminology

• A **path** is a sequence of vertices connected by edges

• A **cycle** is a path whose first and last vertices are the same

• Two vertices are **connected** if there is a path between them

• A **connected component** is a sub-graph where each pair of vertices is connected
Directed Graphs (Digraphs)

- A directed graph (digraph) is a set of vertices connected pairwise by directed edges.
Graph Processing Problems

• Is there a path between \( s \) and \( t \) ?
• What is the shortest path between \( s \) and \( t \) ?
• Is there a cycle in the graph?
• Is there a cycle that uses each edge exactly once?
  – Is there an Euler tour?
• Is there a cycle that visits each vertex exactly once?
  – Is there a Hamiltonian tour? (If you add weights to the edges and want to find the lowest weight tour, then this is known as the *Traveling Salesman* problem)
Graph Processing Problems

• Is there a way to connect all of the vertices?
• What is the best (lowest cost) way to connect all of the vertices? (Where is the minimum spanning tree?)
• Is there a vertex whose removal disconnects the graph? (Is the graph biconnected?)
• Can you draw the graph in the plane with no crossing edges? (Is the graph planar?)
• Do two adjacency lists represent the same graph? (Are two graphs isomorphic?)
Digraph Processing Problems

- Is there a directed path from $s$ to $t$?
- What is the shortest directed path from $s$ to $t$?
- Can a digraph be drawn such that all edges point in same direction? (topological sort)
- Is there a directed path between all pairs of vertices in a group? (strong connectivity)
- For which vertices $v$ and $w$ is there a path from $v$ to $w$? (the transitive closure)
Question: Consider an undirected graph that has $n$ vertices, no parallel edges or self-loops, and is connected (i.e., “in one piece”). What is the minimum and maximum number of edges that the graph could have, respectively?

- $n - 1$ and $n(n - 1)/2$
- $n - 1$ and $n^2$
- $n$ and $2^n$
- $n$ and $n^n$
Question: Approximately how many different digraph are there that have \( n \) vertices? (parallel edges and self-loops are not allowed)

a. \( n \)
b. \( n^2 \)
c. \( 2^n \)
d. \( 2^{(n^2)} \)
e. \( 2^{(2^n)} \)
Graph Representations

• Drawing the graph can provide some intuition about the structure of the graph, but note that intuition can be misleading
Graph Representations

- What kind of data structure should we use to store the graph data?
  - A list of edges (either a linked list or an array)?
  - An adjacency matrix of Booleans or integers?
  - An adjacency list (a vertex-indexed array of lists)?
List of Edges

A list of edges for an undirected graph can be implemented using either a linked list or an array of size $2m$, where $m$ is the number of edges.
Adjacency Matrix

• An adjacency matrix is a two-dimensional array that has one row and one column for each vertex in the graph.
• For each edge of the graph \((v_i, v_j)\), the location at row \(i\) and column \(j\) is 1, all other locations are 0.
• For an undirected graph, the matrix will be symmetric along the diagonal.
• For a weighted graph, the adjacency matrix would have the weight of each edge in the matrix (row, column) entry, zeros along the diagonal, and \(\infty\) every place else.
Adjacency Matrix

\[
\begin{bmatrix}
1 & 2 & 3 & 4 & 5 \\
1 & 0 & 1 & 1 & 0 & 0 \\
2 & 1 & 0 & 0 & 0 & 0 \\
3 & 0 & 1 & 0 & 0 & 0 \\
4 & 0 & 0 & 1 & 0 & 1 \\
5 & 0 & 1 & 0 & 1 & 0
\end{bmatrix}
\]
Adjacency Matrix

- For \( n \) vertices we need an \( n^2 \) matrix of Boolean values if the graph is not weighted, or integer values if it is weighted.
- If the graph is undirected, we only need to store the upper (or lower) triangle, because the matrix will be symmetric.
- Good choice for \textit{dense} graphs (a graph that has many edges) because in general, a dense graph has approximately \( n^2 \) edges.
Adjacency List

• An adjacency list is a list of references, one for each vertex of the graph
• These references are the start of a linked list of vertices that can be reached from this vertex by one edge of the graph
• For a weighted graph, this list would also include the weight for each edge
Adjacency List
Adjacency List

• For $n$ vertices and $m$ edges we need space proportional to $n + m$, along with extra space for the reference to the next element
  – Assuming 4 bytes per data item and 8 bytes for the reference, we need $(4+8)n + (4+8)m = 12n + 12m$ total space

• Good choice for *sparse* graphs (a graph with few edges)
  – In general, a sparse graph has approximately $n$ edges
Matrix or List?

• Question: Compare the **space** complexity of the adjacency matrix to the adjacency list

  – Assume you have a graph with 50 vertices and 1200 edges, each data item occupies 4 bytes, and a memory address occupies 8 bytes

  – Adjacency matrix: $50 \times 50 \times 4 = 10,000$ bytes

  – Adjacency list: $12 \times 50 + 12 \times 1200 = 15,000$ bytes
Matrix or List?

Space required (in bytes) for a graph with 50 vertices on a 64-bit machine:

<table>
<thead>
<tr>
<th>Edges</th>
<th>Matrix</th>
<th>List</th>
</tr>
</thead>
<tbody>
<tr>
<td>49</td>
<td>10,000</td>
<td>1,188</td>
</tr>
<tr>
<td>100</td>
<td>10,000</td>
<td>1,800</td>
</tr>
<tr>
<td>250</td>
<td>10,000</td>
<td>3,600</td>
</tr>
<tr>
<td>500</td>
<td>10,000</td>
<td>6,600</td>
</tr>
<tr>
<td>1,000</td>
<td>10,000</td>
<td>12,600</td>
</tr>
<tr>
<td>1,200</td>
<td>10,000</td>
<td>15,000</td>
</tr>
</tbody>
</table>

list better → 500
matrix better → 1,000

list better

matrix better
## Matrix or List?

Space required (in bytes) for a graph with 5000 vertices on a 64-bit machine:

<table>
<thead>
<tr>
<th>Edges</th>
<th>Matrix</th>
<th>List</th>
</tr>
</thead>
<tbody>
<tr>
<td>49</td>
<td>100,000,000</td>
<td>60,588</td>
</tr>
<tr>
<td>100</td>
<td>100,000,000</td>
<td>61,200</td>
</tr>
<tr>
<td>250</td>
<td>100,000,000</td>
<td>63,000</td>
</tr>
<tr>
<td>500</td>
<td>100,000,000</td>
<td>66,000</td>
</tr>
<tr>
<td>1,000</td>
<td>100,000,000</td>
<td>72,000</td>
</tr>
<tr>
<td>1,200</td>
<td>100,000,000</td>
<td>74,400</td>
</tr>
<tr>
<td>1,500</td>
<td>100,000,000</td>
<td>78,000</td>
</tr>
<tr>
<td>2,000</td>
<td>100,000,000</td>
<td>84,000</td>
</tr>
<tr>
<td>2,500</td>
<td>100,000,000</td>
<td>90,000</td>
</tr>
</tbody>
</table>

Space-wise, adjacency list is almost always better than a matrix.
Matrix or List?

• Question: Compare the time complexity of the adjacency matrix to the adjacency list
  – Goal: given vertex \( v \) and vertex \( w \), find out if there is an edge between them
  – Adjacency matrix: constant time, fast look-up because given \( v \) and \( w \), we can index directly into the matrix
  – Adjacency list: may be slow because after indexing to vertex \( v \), we must traverse \( v \)'s adjacency list to find \( w \)
    • But most graph algorithms are based on iterating over vertices adjacent to \( v \); thus runtime is usually proportional to \( \text{degree}(v) \) rather than \( n \)
    • For sparse graphs, degree (\( v \)) is much less than \( n \)
Matrix or List?

• In practice, you usually want to use the adjacency list representation
  – Real-world graphs tend to be very sparse

Image from www.sciencedirect.com
Matrix or List?

- For a directed graph, as with undirected graphs, in practice we usually use the adjacency list representation.
## Matrix or List?

<table>
<thead>
<tr>
<th>Representation</th>
<th>Space</th>
<th>Cost of adding an edge</th>
<th>Check if there is an edge between v and w</th>
<th>Iterate over vertices adjacent to v</th>
</tr>
</thead>
<tbody>
<tr>
<td>List of edges</td>
<td>2m</td>
<td>constant</td>
<td>m</td>
<td>m</td>
</tr>
<tr>
<td>Adjacency matrix</td>
<td>$n^2$</td>
<td>constant</td>
<td>constant</td>
<td>n</td>
</tr>
<tr>
<td>Adjacency list</td>
<td>$n + m$</td>
<td>constant</td>
<td>degree(v)</td>
<td>degree(v)</td>
</tr>
</tbody>
</table>
Depth-First Search

• In depth-first search (DFS), we follow a path through the graph until we reach a dead end
• We then back up until we reach a vertex with an edge to an unvisited vertex
• We take this edge and again follow it until we reach a dead end
• This process continues until we have to back all the way up to the starting vertex, and there are no edges to unvisited vertices from it
• The runtime of DFS is proportional to $m + n$ where $n$ is the number of vertices and $m$ is the number of edges
Depth-First Search

• Consider the following undirected graph:

• The order of the depth-first search of this graph (assuming vertices are visited in numerical order) starting at vertex 1 is: 1 → 2 → 3 → 4 → 7 → 5 → 6 → 8 → 9
Depth-First Search

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.
Depth-First Search

DFS(G)
1  for each vertex $u \in G.V$
2      $u.color = WHITE$
3      $u.\pi = NIL$
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  $time = time + 1$  // white vertex $u$ has just been discovered
2  $u.d = time$
3  $u.color = GRAY$
4  for each $v \in G.Adj[u]$  // explore edge $(u, v)$
5      if $v.color == WHITE$
6        $v.\pi = u$
7        DFS-VISIT(G, v)
8  $u.color = BLACK$  // blacken $u$; it is finished
9  $time = time + 1$
10 $u.f = time$
Application of DFS: Reachability

• Goal: Find all vertices reachable from $s$ in a directed graph

• Notice that every edge crossing the blue boundary goes “into” the set of reachable vertices

• The runtime of reachability is proportional to $m + n$ (why?)
Application of DFS: Control Flow

• Goal: Program control flow
  – Every program is a digraph
    • Vertex = basic block of instructions
    • Edge = conditional (or a loop)
  – Find and remove unreachable code (dead code elimination)
  – Detect infinite loops
  – Determine whether exiting the program is possible
Application of DFS: Garbage Collection

- The roots are objects known to be directly accessible by program (e.g., on the stack)
- To find all objects indirectly accessible by a program: start at a root and follow a chain of pointers to the objects (DFS), marking all found objects as you go
- Any objects not marked are garbage collected
Question: Suppose that during a depth-first search of digraph G, DFS-VISIT(G,v) is called after DFS-VISIT(G,w) but before DFS-VISIT(G,w) returns. Which of the following must be true of graph G?

a. There exists a directed path from v to w
b. There exists a directed path from w to v
c. There does not exist a directed path from v to w
d. There exists a directed cycle containing both v and w
Is an Undirected Graph Connected?

- Suppose you generated a graph of $n$ vertices with random connections (assume there is a probability $p$ that any two vertices are connected by an edge).
- How would you know if the graph is connected, or is it a forest?
- Solution: perform a depth-first search
  - Keep a count of how many vertices are visited
    - For each vertex visited, if its color is black then increment a counter
  - If the number of vertices visited during the DFS is equal to $N$ (the number of vertices in the graph), then the graph is connected
Undirected Connected Components

• Goal: Partition a set of vertices of an undirected graph such that all vertices that are connected to any others are in the same partition

• The relation "is connected to" is an equivalence relation:
  – Reflexive: $v$ is connected to $v$
  – Symmetric: if $v$ is connected to $w$, then $w$ is connected to $v$
  – Transitive: if $v$ connected to $w$ and $w$ connected to $x$, then $v$ connected to $x$
Undirected Connected Components

- Recall that vertices \( v \) and \( w \) are connected if there is a path between them.
- Definition: A connected component is a \textit{maximal} set of connected vertices.
**Connected components**

- Initialize all vertices $v$ as unmarked.
- For each unmarked vertex $v$, run DFS to identify all vertices discovered as part of the same component.

<table>
<thead>
<tr>
<th>v</th>
<th>marked[]</th>
<th>id[]</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>T</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>T</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>T</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>T</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>T</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>T</td>
<td>0</td>
</tr>
<tr>
<td>6</td>
<td>T</td>
<td>0</td>
</tr>
<tr>
<td>7</td>
<td>T</td>
<td>1</td>
</tr>
<tr>
<td>8</td>
<td>T</td>
<td>1</td>
</tr>
<tr>
<td>9</td>
<td>T</td>
<td>2</td>
</tr>
<tr>
<td>10</td>
<td>T</td>
<td>2</td>
</tr>
<tr>
<td>11</td>
<td>T</td>
<td>2</td>
</tr>
<tr>
<td>12</td>
<td>T</td>
<td>2</td>
</tr>
<tr>
<td>13</td>
<td>F</td>
<td></td>
</tr>
<tr>
<td>14</td>
<td>F</td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>F</td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>F</td>
<td></td>
</tr>
<tr>
<td>17</td>
<td>F</td>
<td></td>
</tr>
<tr>
<td>18</td>
<td>F</td>
<td></td>
</tr>
<tr>
<td>19</td>
<td>F</td>
<td></td>
</tr>
<tr>
<td>20</td>
<td>F</td>
<td></td>
</tr>
<tr>
<td>21</td>
<td>F</td>
<td></td>
</tr>
<tr>
<td>22</td>
<td>F</td>
<td></td>
</tr>
<tr>
<td>23</td>
<td>F</td>
<td></td>
</tr>
<tr>
<td>24</td>
<td>F</td>
<td></td>
</tr>
<tr>
<td>25</td>
<td>F</td>
<td></td>
</tr>
<tr>
<td>26</td>
<td>F</td>
<td></td>
</tr>
<tr>
<td>27</td>
<td>F</td>
<td></td>
</tr>
<tr>
<td>28</td>
<td>F</td>
<td></td>
</tr>
<tr>
<td>29</td>
<td>F</td>
<td></td>
</tr>
<tr>
<td>30</td>
<td>F</td>
<td></td>
</tr>
</tbody>
</table>
Application of Connected Components

- Goal: Study of spread of STDs
Application of Connected Components

• Goal: Flood fill a digital image with a specific pixel color (e.g., Photoshop magic wand)

• Solution: Model the image using a graph
  • Each pixel in the image becomes a vertex in a graph
  • Two adjacent pixels with similar color becomes an edge in the graph
  • Perform DFS to find the connected components of a selected pixel
  • Set the color of all of the connected pixels to the color of the given pixel
Application of Connected Components

• Goal: Particle tracking – Given a grayscale video of astronomical data, track moving particles over time
  – A pixel becomes a vertex in a graph
  – Two adjacent pixels with similar grayscale values becomes an edge in the graph
  – Perform DFS to find connected components
  – Track connected components from frame to frame
Topological Sort

• Goal: Given a set of tasks to be completed (or courses to take) with precedence constraints, in which order should we schedule the tasks?
Topological Sort

• Topological sort requires that the problem be represented as a **DAG** (directed acyclic graph)
Topological Sort

Algorithm (first be sure that the graph is directed and acyclic):

1. Call DFS to compute the finish time for each vertex
2. As each vertex is finished, push it onto a stack (this will create the reverse post-ordering of the vertices)
3. Return the stack of vertices when DFS is done
Topological Sort:
Topological Sort Proof of Correctness

• Assert: If \((u, v)\) is a directed edge from \(u\) to \(v\), then the finish time \((u) > \text{finish time } (v)\)
  – Case 1: \(u\) is visited before \(v\) (as in tie to jacket)
    • The recursive call corresponding to \(u\) finishes after that of \(v\); therefore, \(\text{finish time } (u) > \text{finish time } (v)\)
  – Case 2: \(u\) is visited after \(v\) (as in belt to jacket)
    • The recursive call corresponding to \(v\) finishes before \(u\) starts; therefore, \(\text{finish time } (u) > \text{finish time } (v)\)
Cycle Detection

• A digraph has a topological order if and only if there is no directed cycle in the digraph
  – We should first check if the digraph has a cycle

• Other applications of cycle detection:
  – Cyclic inheritance detection in class hierarchy
  – Circular reference detection in spreadsheets

• Goal: Given a digraph, determine if there is a directed cycle in the digraph
Cycle Detection

A directed graph is acyclic if and only if a depth-first search yields no back edges

- Tree edges (T)
  - Edges that belong to the DFS tree
- Back edges (B)
  - Non-tree edges that connect to an ancestor
  - Self-loops
- Forward edges (F)
  - Non-tree edges that connect to a descendant
- Cross edges (C)
  - All other edges
  - Can go between trees
  - Cannot connect to ancestor or descendant nodes
Cycle Detection

Result of DFS, with time stamps. Intervals indicating the lifetime of a given node. If two intervals overlap, then the node with the shorter interval is a descendant of the node with the longer interval.

DFS tree, with back, forward, and cross edges labeled.
Strong Components

- Vertices $v$ and $w$ are strongly connected if there is **both** a directed path from $v$ to $w$ and a directed path from $w$ to $v$
  - Strong connectivity is an equivalence relation
  - A strong component is a maximal subset of strongly-connected vertices
Connected Vs. Strongly Connected

**Connected**: v and w are connected if there is a path between v and w

- Connected component ids
- Easy to compute with DFS

**Strongly connected**: v and w are strongly connected if there is both a directed path from v to w and a directed path from w to v

- Strongly-connected component ids
- How to compute?
Computing Strong Components

1. Run DFS on $G$ to compute finishing times
2. Compute the *reverse graph* $G^R$ by reversing the direction of all edges
3. Run DFS on $G^R$, considering the vertices in the order of decreasing finish time
4. Output the vertices in each tree formed from step 3 as a separate strongly connected component
Computing Strong Components

Step 1: Run DFS to find finish times.

Step 4: There are 4 strongly connected components.

Steps 2 and 3: Compute reverse graph and run DFS on it, in order of decreasing finish time.

Step 4: There are 4 strongly connected components.
Analysis of Strong Components

• This two-phase algorithm computes the strong components of a digraph in time proportional to $m+n$

• The time is due to running DFS twice and computing $G^R$, but since computing $G^R$ takes time proportional to $m$, the $m$ is “absorbed” into the overall running time
Application of Strong Components

• Goal: Create a software module dependency graph (vertex = software module, edge = module to a dependent module)

• Strong component = subset of mutually interacting modules
Application of Strong Components

- Goal: Build a crawler for the web (vertex = webpage, edge = hyperlink)
- Strong components: corporate web sites, online communities, small-worlds, etc.

Image from www.seobook.com
Breadth-First Search

• In breadth-first search (BFS), from the starting vertex, follow all paths of length one
• Then follow paths of length two that go to any unvisited vertices
• Continue until there are no unvisited vertices along any of the paths
• Running time is proportional to $n + m$
Breadth-First Search

- Consider the following undirected graph:

- The order of the breadth-first search of this graph (assuming vertices are visited in numerical order) starting at vertex 1 is: $1 \rightarrow 2 \rightarrow 8 \rightarrow 3 \rightarrow 7 \rightarrow 4 \rightarrow 5 \rightarrow 9 \rightarrow 6$
Breadth-First Search

\[
\text{BFS}(G, s)
\]

1. for each vertex \( u \in G.V - \{s\} \)
2. \( u.color = \text{WHITE} \)
3. \( u.d = \infty \)
4. \( u.\pi = \text{NIL} \)
5. \( s.color = \text{GRAY} \)
6. \( s.d = 0 \)
7. \( s.\pi = \text{NIL} \)
8. \( Q = \emptyset \)
9. \( \text{ENQUEUE}(Q, s) \)
10. while \( Q \neq \emptyset \)
    11. \( u = \text{DEQUEUE}(Q) \)
    12. for each \( v \in G.Adj[u] \)
        13. if \( v.color = \text{WHITE} \)
            14. \( v.color = \text{GRAY} \)
            15. \( v.d = u.d + 1 \)
            16. \( v.\pi = u \)
            17. \( \text{ENQUEUE}(Q, v) \)
        18. \( u.color = \text{BLACK} \)
Breadth-First Search
Breadth-First Search

• Using BFS, we can find the shortest un-weighted path (the fewest number of edges) from a source vertex to any destination vertex in time proportional to $m + n$
• To do this, we must keep track of the parent $\pi$ of each vertex, and use this information to create a BFS tree
• For example, the shortest path from vertex 1 (the source) to vertex 6 (the destination) is $1 \rightarrow 8 \rightarrow 7 \rightarrow 5 \rightarrow 6$
Web Crawling Using BFS

• Goal: Crawl the web, starting from a web page
• Solve with BFS using an *implicit* digraph
  – Choose root web page as source $s$
  – Maintain a queue of websites to explore
  – Maintain a set of discovered websites
  – Dequeue the next website and enqueue websites to which it links (provided you haven't done so before)
Application of BFS

• Goal: Find the fewest number of hops in a communication network
Graph Processing Challenges

• Challenge: Find a cycle in a graph
• How difficult is this challenge?
  ✓ Any programmer could do it (use DFS)
  – Typical diligent algorithms student could do it
  – Hire an expert
  – Intractable
  – No one knows
  – Impossible
Graph Processing Challenges

• Challenge: Is a graph bipartite?
  – A graph is bipartite if you can divide the vertices into two subsets such that all edges go from one subset to the other

• How difficult is this challenge?
  – Any programmer could do it
  ✓ Typical diligent algorithms student could do it (label the vertices “red” or “green” during a DFS)
  – Hire an expert
  – Intractable
  – No one knows
  – Impossible
Graph Processing Challenges

• Challenge: Can you find a cycle that uses every edge exactly once (an Euler tour)?
• How difficult is this challenge?
  – Any programmer could do it
  ✓ Typical diligent algorithms student could do it (if the graph is connected, and every vertex has even degree, then there is such a cycle)
  – Hire an expert
  – Intractable
  – No one knows
  – Impossible
Graph Processing Challenges

- Challenge: Can you lay out a graph on a 2D surface without any edges crossing (is a graph planar)?
- Related to graph (map) coloring
- How difficult is this challenge?
  - Any programmer could do it
  - Typical diligent algorithms student could do it
  ✓ Hire an expert
  - Intractable
  - No one knows
  - Impossible
Graph Processing Challenges

• Challenge: Can you find a cycle that visits every vertex exactly once (Hamilton Cycle or Traveling Salesman)

• How difficult is this challenge?
  – Any programmer could do it
  – Typical diligent algorithms student could do it
  – Hire an expert

✓ Intractable
  – No one knows
  – Impossible
Graph Processing Challenges

• Challenge: Are two graphs identical except for vertex names (graph isomorphism)?

• How difficult is this challenge?
  – Any programmer could do it
  – Typical diligent algorithms student could do it
  – Hire an expert
  – Intractable
  ✓ No one knows
  – Impossible

0 ↔ 4
1 ↔ 3
2 ↔ 2
3 ↔ 6
4 ↔ 5
5 ↔ 0
6 ↔ 1
Graph Processing Challenges

• Question: Which one of the following graph-processing problems is *unlikely* to have an algorithm whose running time is $m + n$?
  
a) Determine whether a graph is bipartite  
b) Determine whether a graph has an Euler cycle  
c) Determine whether a graph has a Hamilton cycle  
d) Determine whether a graph can be drawn in the plane such that no two edges cross