Directed Graphs

Digraph Applications

<table>
<thead>
<tr>
<th>Digraph</th>
<th>Vertex</th>
<th>Edge</th>
</tr>
</thead>
<tbody>
<tr>
<td>financial</td>
<td>stock, currency</td>
<td>transaction</td>
</tr>
<tr>
<td>transportation</td>
<td>street intersection, airport</td>
<td>highway, airway route</td>
</tr>
<tr>
<td>scheduling</td>
<td>task</td>
<td>precedence constraint</td>
</tr>
<tr>
<td>control flow</td>
<td>code block</td>
<td>jump</td>
</tr>
<tr>
<td>Internet</td>
<td>web page</td>
<td>hyperlink</td>
</tr>
<tr>
<td>game</td>
<td>board position</td>
<td>legal move</td>
</tr>
<tr>
<td>telephone</td>
<td>person</td>
<td>placed call</td>
</tr>
<tr>
<td>food web</td>
<td>species</td>
<td>predator-prey relation</td>
</tr>
<tr>
<td>infectious disease</td>
<td>person</td>
<td>infection</td>
</tr>
<tr>
<td>citation</td>
<td>journal article</td>
<td>citation</td>
</tr>
<tr>
<td>object graph</td>
<td>object</td>
<td>pointer</td>
</tr>
<tr>
<td>inheritance hierarchy</td>
<td>class</td>
<td>inherits from</td>
</tr>
</tbody>
</table>

Directed Graphs

Digraph. Set of objects with oriented pairwise connections.

Ex. One-way street, hyperlink.

Ecological Food Web

Food web graph.
- Vertex = species.
- Edge = from prey to predator.

Some Digraph Problems

Transitive closure. Is there a directed path from v to w?

Strong connectivity. Are all vertices mutually reachable?

Topological sort. Can you draw the graph so that all of the edges point from left to right?

PERT/CPM. Given a set of tasks with precedence constraints, what is the earliest that we can complete each task?

Shortest path. Given a weighted graph, find best route from v to w?

PageRank. What is the importance of a web page?

Adjacency Matrix Representation

Adjacency matrix representation.
- Two-dimensional \( V \times V \) boolean array.
- Edge \( v \rightarrow w \) in graph: \( \text{adj}[v][w] = \text{true} \).

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
<th>G</th>
<th>H</th>
<th>I</th>
<th>J</th>
<th>K</th>
<th>L</th>
<th>M</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>B</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>C</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>D</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>E</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>F</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>G</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>H</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>I</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>J</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>K</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>L</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>M</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Digraph Representation

Vertex names.
- This lecture: use integers between 0 and \( V - 1 \).
- Real world: convert between names and integers with symbol table.

Orientation of edge matters.

Vertex indexed array of lists.
- Space proportional to number of edges.
- One representation of each edge.

Adjacency List Representation

Adjacency matrix representation.
- Two-dimensional \( V \times V \) boolean array.
- Edge \( v \rightarrow w \) in graph: \( \text{adj}[v][w] = \text{true} \).

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
<th>G</th>
<th>H</th>
<th>I</th>
<th>J</th>
<th>K</th>
<th>L</th>
<th>M</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>B</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>C</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>D</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>E</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>F</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>G</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>H</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>I</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>J</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>K</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>L</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>M</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Real world: convert between names and integers with symbol table.
Adjacency List: Java Implementation

Implementation. Same as Graph, but only insert one copy of each edge.

```java
public class Digraph {
    private int V;
    private Sequence<Integer>[] adj;

    public Digraph(int V) {
        this.V = V;
        adj = (Sequence<Integer>[]) new Sequence[V];
        for (int v = 0; v < V; v++)
            adj[v] = new Sequence<Integer>();
    }

    public void insert(int v, int w) {
        adj[v].add(w);
    }

    public Iterable<Integer> adj(int v) {
        return adj[v];
    }
}
```

Digraph Search

Digraph Representations

Digraphs are abstract mathematical objects.
- ADT implementation requires specific representation.
- Efficiency depends on matching algorithms to representations.

<table>
<thead>
<tr>
<th>Representation</th>
<th>Space</th>
<th>Edge from v to w?</th>
<th>Enumerate edges leaving v?</th>
</tr>
</thead>
<tbody>
<tr>
<td>List of edges</td>
<td>O(E + V)</td>
<td>O(E)</td>
<td>O(E)</td>
</tr>
<tr>
<td>Adjacency matrix</td>
<td>O(V^2)</td>
<td>O(V)</td>
<td>O(V)</td>
</tr>
<tr>
<td>Adjacency list</td>
<td>O(E + V)</td>
<td>O(outdeg(v))</td>
<td>O(outdeg(v))</td>
</tr>
</tbody>
</table>

Digraphs in practice. [use adjacency list representation]
- Real world digraphs are sparse.
- Bottleneck is iterating over edges leaving v.

Reachability

Goal. Find all vertices reachable from s along a directed path.

Depth first search. To visit a vertex v:
- Mark v as visited.
- Recursively visit all unmarked vertices w adjacent to v.

Running time. O(E) since each edge examined at most once.
**Mark-Sweep Garbage Collector**

**Roots.** Objects known to be accessible by program (e.g., stack).

**Live objects.** Objects that the program could get to by starting at a root and following a chain of pointers,

- easy to identify pointers in type-safe language

**Mark-sweep algorithm.** [McCarthy 1960]
- Mark: run DFS from roots to mark live objects.
- Sweep: if object is unmarked, it is garbage, so add to free list.

**Extra memory.** Uses 1 extra mark bit per object, plus DFS stack.

---

**Depth First Search**

**Remark.** Same as undirected version, except Digraph instead of Graph.

```java
public class DFSearcher {
    private boolean[] marked;
    public DFSearcher(Digraph G, int s) {
        marked = new boolean[G.V()];
        dfs(G, s);
    }
    private void dfs(Digraph G, int v) {
        marked[v] = true;
        for (int w : G.adj(v))
            if (!marked[w]) dfs(G, w);
    }
    public boolean isReachable(int v) { return marked[v]; }
}
```

---

**Control Flow Graph**

**Control-flow graph.**
- Vertex = basic block (straight-line program).
- Edge = jump.

**Dead code elimination.** Find (and remove) code blocks that are unreachable during execution.

- code might arise from compiler optimizations
- or careless programmer

**Infinite loop detection.** Exit block is unreachable from entry block.

**Caveat.** Not all infinite loops are detectable.

---

**DFS enables direct solution of simple digraph problems.**
- Reachability.
- Cycle detection.
- Topological sort.
- Transitive closure.
- Find path from \( s \) to \( t \).

**Basis for solving difficult digraph problems.**
- Directed Euler path.
- Strong connected components.
Breadth First Search

Shortest path. Find the shortest directed path from $s$ to $t$.

BFS. Analogous to BFS in undirected graphs.

Web Crawler: Java Implementation

```
Queue<String> q = new Queue<String>();  // queue of sites to crawl
SET<String> visited = new SET<String>();   // set of visited sites

String s = "http://www.princeton.edu";
q.enqueue(s);
visited.add(s);
while (!q.isEmpty()) {
    String v = q.dequeue();   // read in row html
    In in = new In(v);
    String input = in.readLine();  // search using regular expression
    String regexp = "http://(\\w+\.)*(\\w+)";
    Pattern pattern = Pattern.compile(regexp);
    Matcher matcher = pattern.matcher(input);
    while (matcher.find()) {
        String w = matcher.group();
        if (!visited.contains(w)) {
            visited.add(w);  // if unvisited, mark as visited
            q.enqueue(w);  // and put on queue
        }
    }
}
```

Application: Web Crawler

Web graph. Vertex = website, edge = hyperlink.

Goal. Crawl Internet, starting from some root website.

Solution. BFS with implicit graph.

BFS.
- 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).

Q. Why not use DFS?
Transitive Closure

Transitive closure. Is there a directed path from v to w?

Implementing the transitive closure.  

```
public class TransitiveClosure {
    private boolean[][] tc;
    public TransitiveClosure(Digraph G) {
        tc = new boolean[G.V()][G.V()];
        for (int v = 0; v < G.V(); v++)
            dfs(G, v, v);   // run dfs from every vertex

        // reachability from s, make it to v
        private void dfs(Digraph G, int s, int v) {
            tc[s][v] = true;
            for (int w : G.adj(v))
                if (!tc[s][w]) dfs(G, s, w);
        }

        public boolean reachable(int v, int w) {
            return tc[v][w];  // is w reachable from v?
        }
    }
}
```

Transitive Closure: Java Implementation

Transitive closure. Is there a directed path from v to w?

**Lazy.** Run separate DFS for each query.

**Eager.** Run DFS from every vertex v.

<table>
<thead>
<tr>
<th>Method</th>
<th>Preprocess</th>
<th>Query</th>
<th>Space</th>
</tr>
</thead>
<tbody>
<tr>
<td>DFS (lazy)</td>
<td>O(1)</td>
<td>O(E + V)</td>
<td>O(E + V)</td>
</tr>
<tr>
<td>DFS (eager)</td>
<td>O(E V)</td>
<td>O(1)</td>
<td>O(V^2)</td>
</tr>
</tbody>
</table>

**Remark.** Directed problem is harder than undirected one.

Open research problem.  O(1) query, O(V^2) preprocessing time.

---

Topological Sort
Topological Sort

**DAG.** Directed acyclic graph.

**Topological sort.** Redraw DAG so all edges point left to right.

**Observation.** Not possible if graph has a directed cycle.

Application: Scheduling

**Scheduling.** Given a set of tasks to be completed with precedence constraints, in what order should we schedule the tasks?

**Graph model.**
- Create a vertex \( v \) for each task.
- Create an edge \( v \rightarrow w \) if task \( v \) must precede task \( w \).
- Schedule tasks in topological order.

**Topological Sort: DFS**

**Topologically sort a DAG.**
- Run DFS.
- Reverse postorder numbering yields a topological sort.

**Pf of correctness.** When DFS backtracks from a vertex \( v \), all vertices reachable from \( v \) have already been explored.

**Running time.** \( O(E + V) \).

**Q.** If not a DAG, how would you identify a cycle?

```
public class TopologicalSorter {
    private int cnt;
    private boolean[] visited;
    private int[] ts;

    public TopologicalSorter(Digraph G) {
        visited = new boolean[G.V()];
        ts = new int[G.V()];
        cnt = G.V();
        for (int v = 0; v < G.V(); v++)
            if (!visited[v]) tsort(G, v);
    }

    private void tsort(Digraph G, int v) {
        visited[v] = true;
        for (int w : G.adj(v))
            if (!visited[w]) tsort(G, w);
        ts[--cnt] = v;
    }
}
```

**Assign numbers in reverse DFS postorder.**
Topological sort applications.
- Causalities.
- Compilation units.
- Class inheritance.
- Course prerequisites.
- Deadlocking detection.
- Temporal dependencies.
- Pipeline of computing jobs.
- Check for symbol link loop.
- Evaluate formula in spreadsheet.

Program Evaluation and Review Technique / Critical Path Method

PERT/CPM algorithm.
- Compute topological order of vertices.
- Initialize $\text{fin}[v] = 0$ for all vertices $v$.
- Consider vertices $v$ in topological order.
  - For each edge $v \rightarrow w$, set $\text{fin}[w] = \max(\text{fin}[w], \text{fin}[v] + \text{time}[w])$