Directed Graphs

Digraph. Set of objects with oriented pairwise connections.
Ex. One-way street, hyperlink.

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>WordNet</td>
<td>synset</td>
<td>hypernym</td>
</tr>
<tr>
<td>Web</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>
<tr>
<td>control flow</td>
<td>code block</td>
<td>jump</td>
</tr>
</tbody>
</table>

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 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 $s$ to $t$?
- **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: $\text{adj}[v][w] = \text{true}$.

```
 0 1 2 3 4 5 6 7 8 9 10 11 12
0 1 0 0 1 1 0 0 0 0 0 0 0 0
1 0 0 0 0 0 0 0 0 0 0 0 0 0
2 0 0 0 0 0 0 0 0 0 0 0 0 0
3 0 0 0 0 0 0 0 0 0 0 0 0 0
4 0 0 0 0 1 0 0 0 0 0 0 0 0
5 0 0 0 0 1 0 0 0 0 0 0 0 0
6 0 0 0 1 0 0 0 0 0 0 0 0 0
7 0 0 0 0 0 0 0 1 0 0 0 0 0
8 0 0 0 0 0 0 0 0 0 0 0 0 0
9 0 0 0 0 0 0 0 0 0 0 1 1 1
10 0 0 0 0 0 0 0 0 0 0 0 0 0
11 0 0 0 0 0 0 0 0 0 0 0 0 1
12 0 0 0 0 0 0 0 0 0 0 0 0 0
```

### Adjacency List Representation

**Adjacency list representation.** Vertex indexed array of lists.

```
0:5 2 1 6
1:2 3:4:3
5:4 3
6:4 7 8
8:9:10
9:11
10:11:12
12:
```

### 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.**
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>Iterate over edges leaving v?</th>
</tr>
</thead>
<tbody>
<tr>
<td>List of edges</td>
<td>E + V</td>
<td>E</td>
<td>E</td>
</tr>
<tr>
<td>Adjacency matrix</td>
<td>V^2</td>
<td>1</td>
<td>V</td>
</tr>
<tr>
<td>Adjacency list</td>
<td>E + V</td>
<td>outdegree(v)</td>
<td>outdegree(v)</td>
</tr>
</tbody>
</table>

Digraphs in practice. => use adjacency list representation

- Bottleneck is iterating over edges leaving v.
- Real world digraphs are sparse.

E is proportional to V

Reachability

**Goal.** Find all vertices reachable from s along a directed path.
**Depth First Search**

**Depth first search.** Same as for undirected graphs.

\[
\text{DFS (to visit a vertex } v) \]

Mark \( v \) as visited.
Visit all unmarked vertices \( w \) adjacent to \( v \).

**Running time.** \( O(E) \) since each edge examined at most once.

**Remark.** Same as undirected version, except \( \text{Digraph} \) instead of \( \text{Graph} \).

```
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]; }
}
```

**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.

```
\[\text{Mark-sweep algorithm. } [\text{McCarthy, 1960}]\]
```

- 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.

**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.
\( \text{dead code can arise from compiler optimization (or careless programmer)} \)

**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.

Application: Web Crawler

Web graph. Vertex = website, edge = hyperlink.

Goal. Crawl Internet, starting from some root website.

Solution. BFS with implicit graph.

BFS.
- Start at some root website, say \( \text{http://www.princeton.edu} \).
- 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?

Shortest path. Find the shortest directed path from \( s \) to \( t \).

BFS. Analogous to BFS in undirected graphs.

```java
Queue<String> q = new Queue<String>();
SET<String> visited = new SET<String>();
String s = "http://www.princeton.edu";
q.enqueue(s);
visited.add(s);
while (!q.isEmpty()) {
    String v = q.dequeue();
    System.out.println(v);
    In in = new In(v);
    String input = in.readLine();
    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
                and put on queue
            )
            q.enqueue(w);
        }
    }
}
```
Transitive Closure

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

Lazy. Run separate DFS for each query.
Eager. Run DFS from every vertex \( v \); store results.

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

Transitive Closure: Java Implementation

Implementation. Use an array of DFSearcher objects.

```java
public class TransitiveClosure {
    private DFSearcher[] tc;

    public TransitiveClosure(Digraph G) {
        tc = new Reachability[G.V()];
        for (int v = 0; v < G.V(); v++)
            tc[v] = new Reachability[G, v];
    }

    public boolean reachable(int v, int w) {
        return tc[v].isReachable(w);
    }
}
```
**Topological Sort**

**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.

| 0. read programming assignment |
| 1. download files               |
| 2. write code                  |
| ...                            |
| 12. sleep                      |

**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.

**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?
**Program Evaluation and Review Technique / Critical Path Method**

**PERT/CPM.**
- Task \( v \) takes \( t[v] \) units of time.
- Can work on jobs in parallel.
- Precedence constraints: must finish task \( v \) before beginning task \( w \).
- What’s earliest we can finish each task?

<table>
<thead>
<tr>
<th>Index</th>
<th>Task</th>
<th>Time</th>
<th>Prereq</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>begin</td>
<td>0</td>
<td>-</td>
</tr>
<tr>
<td>A</td>
<td>framing</td>
<td>4</td>
<td>A</td>
</tr>
<tr>
<td>B</td>
<td>roofing</td>
<td>2</td>
<td>B</td>
</tr>
<tr>
<td>C</td>
<td>siding</td>
<td>6</td>
<td>B</td>
</tr>
<tr>
<td>D</td>
<td>windows</td>
<td>5</td>
<td>D</td>
</tr>
<tr>
<td>E</td>
<td>plumbing</td>
<td>3</td>
<td>D</td>
</tr>
<tr>
<td>F</td>
<td>electricity</td>
<td>4</td>
<td>C, E</td>
</tr>
<tr>
<td>G</td>
<td>paint</td>
<td>6</td>
<td>C, E</td>
</tr>
<tr>
<td>H</td>
<td>finish</td>
<td>0</td>
<td>F, H</td>
</tr>
</tbody>
</table>

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

**Topological Sort: Java Implementation**

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

    public TopologicalSorter(Digraph G) {
        visited = new boolean[G.V()];
        ts = new int[G.V()];
        count = 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[--count] = 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 symbolic link loop.
- Evaluate formula in spreadsheet.

**Program Evaluation and Review Technique / Critical Path Method**
**Strongly Connected Components**

**Strong Components**

**Def.** Vertices \( v \) and \( w \) are strongly connected if there is a path from \( v \) to \( w \) and a path from \( w \) to \( v \).

**Properties.** Symmetric, transitive, reflexive.

**Strong component.** Maximal subset of strongly connected vertices.

**Brute force.** \( O(EV) \) time using transitive closure.

**Computing Strongly Connected Components**

**Observation 1.** If you run DFS from a vertex in sink strong component, all reachable vertices constitute a strong component.

**Observation 2.** If you run DFS on \( G \), the node with the highest postorder number is in source strong component.

**Observation 3.** If you run DFS on \( G^R \), the node with the highest postorder number is in sink strong component.

![Kernel DAG](image-url)
Kosaraju’s Algorithm

Kosaraju’s algorithm.
- Run DFS on $G^R$ and compute postorder.
- Run DFS on $G$, considering vertices in reverse postorder.

Theorem. Trees in second DFS are strong components. (!)

Ecological Food Web

Ecological food web.
- Vertex = species.
- Edge = from producer to consumer.
- Strong component = subset of species for which energy flows from one another and back.

https://www.twingroves.district96.k12.il.us/Wetlands/Salamander/SalGraphics/salfoodweb.gif