Why Graphs?

- For expressing non-hierarchically related items
- Examples:
  - Networks: pipelines, roads, assignment problems
  - Representing processes: flow charts, Markov models
  - Representing partial orderings: PERT charts, makefiles

Some Terminology

- A graph consists of
  - A set of nodes (aka vertices)
  - A set of edges: pairs of nodes.
  - Nodes with an edge between are adjacent.
  - Depending on problem, nodes or edges may have labels (or weights)
- Typically call node set $V = \{v_0, \ldots\}$, and edge set $E$.
- If the edges have an order (first, second), they are directed edges, and we have a directed graph (digraph), otherwise an undirected graph.
- Edges are incident to their nodes.
- Directed edges exit one node and enter the next.
- A cycle is a path without repeated edges leading from a node back to itself (following arrows if directed).
- A graph is cyclic if it has a cycle, else acyclic. Abbreviation: Directed Acyclic Graph—DAG.

Some Pictures

- Directed
- Undirected
- Acyclic
- Cyclic
- With Edge Labels:
Trees are Graphs

- A graph is connected if there is a (possibly directed) path between every pair of nodes.
- That is, if one node of the pair is reachable from the other.
- A DAG is a (rooted) tree iff connected, and every node but the root has exactly one parent.
- A connected, acyclic, undirected graph is also called a free tree. Free: we're free to pick the root; e.g.,

```
  a  b  c  d  e
    ^   ^   
   /     \
  b     a
  e     d
```

Examples of Use

- Edge = Connecting road, with length.
  
  Detroit 200 Chicago

- Edge = Must be completed before; Node label = time to complete.
  
  Eat 1 hr
  Sleep 8 hrs

- Edge = Begat
  
  Martin George

More Examples

- Edge = some relationship
  
  potstickers eats John loves Mary

- Edge = next state might be (with probability)
  
  hat 0.4 the 0.6 cat 0.4 in 0.1 bed

- Edge = next state in state machine, label is triggering input. (Start at s. Being in state 4 means "there is a substring '001' somewhere in the input").

```
  0 1 2 3 4
  0 1 2 3 4
  1 0 1 1
  2 0 0 1
  3 0 0 0
```

Representation

- Often useful to number the nodes, and use the numbers in edges.
- Edge list representation: each node contains some kind of list (e.g., linked list or array) of its successors (and possibly predecessors).

```
  1: a (2,3) b
      c
  2: b
  3: c (1,2)
```

- Edge sets: Collection of all edges. For graph above:

  \{ (1,2), (1,3), (2,3) \}

- Adjacency matrix: Represent connection with matrix entry:

```
  1 2 3
  1 0 1 1
  2 0 0 1
  3 0 0 0
```
Traversing a Graph

- Many algorithms on graphs depend on traversing all or some nodes.
- Can't quite use recursion because of cycles.
- Even in acyclic graphs, can get combinatorial explosions:

\[ \Theta(2^N) \]

Treat 0 as the root and do recursive traversal down the two edges out of each node: \( \Theta(2^N) \) operations!
- So typically try to visit each node constant # of times (e.g., once).

General Graph Traversal Algorithm

\[
\begin{align*}
\text{foreach collection of vertices fringe} \quad \text{do} \\
\text{fringe = initial collection;}
\end{align*}
\]

while (! fringe.isEmpty())

\[
\begin{align*}
\text{Vertex v = fringe.REMOVE.HIGHEST.PRIORITY.ITEM();} \\
\text{if (! MARKED(v))} \\
\quad \text{MARK(v);} \\
\quad \text{VISIT(v);} \\
\quad \text{for each edge (v,w)} \\
\quad \quad \text{if (NEEDSPROCESSING(w))} \\
\quad \quad \quad \text{Add w to fringe;} \\
\end{align*}
\]

Replace \textit{collection_of_vertices}, \textit{initial_collection}, etc. with various types, expressions, or methods to different graph algorithms.

Example: Depth-First Traversal

Problem: Visit every node reachable from \( v \) once, visiting nodes further from start first.

\[
\text{Stack<Vertex> fringe;} \\
\text{fringe = stack containing \{v\};} \\
\text{while (! fringe.isEmpty())} \\
\quad \text{Vertex v = fringe.pop();} \\
\quad \text{if (! marked(v))} \\
\quad \quad \text{mark(v);} \\
\quad \quad \text{VISIT(v);} \\
\quad \quad \text{for each edge (v,w)} \\
\quad \quad \quad \text{if (! marked(w))} \\
\quad \quad \quad \quad \text{fringe.push(w);} \\
\]

Depth-First Traversal Illustrated

Marked:

Fringe:

Frigh:

Frigh: [a] [b, d] [c, e, d] [d, f, e, d] [f, e, d] [e, e, d] [e, d] []
**Topological Sorting**

**Problem:** Given a DAG, find a linear order of nodes consistent with the edges.

- That is, order the nodes \( v_0, v_1, \ldots \) such that \( v_k \) is never reachable from \( v_k' \) if \( k' > k \).
- Gmake does this. Also PERT charts.

```
Set<Vertex> fringe;
fringe = set of all nodes with no predecessors;
while (! fringe.isEmpty()) {
    Vertex v = fringe.removeOne ();
    ... add v to end of result list;
    for each edge (v,w) {
        if (v.dist() + weight(v,w) < w.dist()) {
            w.dist() = v.dist() + weight(v,w);
            w.back() = v;
        }
    }
}
```

**Topological Sort in Action**

**Shortest Paths: Dijkstra’s Algorithm**

**Problem:** Given a graph (directed or undirected) with non-negative edge weights, compute shortest paths from given source node, \( s \), to all nodes.

- “Shortest” = sum of weights along path is smallest.
- For each node, keep estimated distance from \( s \), . . .
- . . . and of preceding node in shortest path from \( s \).

```
PriorityQueue<Vertex> fringe;
for each node v { v.dist() = ∞; v.back() = null; }
s.dist() = 0;
fringe = priority queue ordered by smallest .dist();
add all vertices to fringe;
while (! fringe.isEmpty()) {
    Vertex v = fringe.removeFirst ();
    for each edge (v,w) {
        if (v.dist() + weight(v,w) < w.dist()) {
            w.dist() = v.dist() + weight(v,w);
            w.back() = v;
        }
    }
}
```

**Example**