Today's Readings: Graph Structures: DSJ, Chapter 12
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
  - As we’ve seen, in representing connected structures as used in Git.
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 the node set $V = \{v_0, \ldots\}$, and the 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—a sequence of edges—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**

Acyclic:

```
 a -- b -- d
 |    |    |
 c    c
```

Cyclic:

```
 a -- d
 |    |
 b    c
```

**Undirected**

```
 a -- d
 |
 b -- e
```

**With Edge Labels**

```
 a -- 3 2 d
 |
 b    1 1
```

```
 a -- 3 d
 |
 b    1 2
```

Last modified: Thu Apr 14 15:01:19 2022
Connectivity (Undirected)

- A path is variously defined as a sequence of vertices \( v_0, v_1, \ldots, v_n \) where there is an edge from each \( v_i \) to \( v_{i+1} \), or as the sequence of edges between them: \( (v_0, v_1), (v_1, v_2), \ldots \).

- An undirected graph is connected if there is a path between every pair of nodes in the graph:

```
[Diagram showing two graphs: one connected and one unconnected.]
```

 Connected

 Unconnected
Connectivity (Directed)

- For directed graphs, it’s more complicated:
  - *Weakly connected*: connected if direction is removed.
  - *Unilaterally connected* (or *semiconnected*): there is a path in one direction or the other between each node pair;
  - *Strongly connected*: there are paths in both directions for each node pair.

![Weakly Connected](image)

![Unilaterally Connected](image)

![Strongly Connected](image)
**Trees and Graphs**

- 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., all the following are the same graph:
Examples of Use

• Edge = Connecting road, with length.

![Diagram showing Detroit to Chicago with a distance of 200 units.]

• Edge = Must be completed before; Node label = time to complete.

![Diagram showing Eat to Sleep with labels '1 hr' and '8 hrs'.]

• Edge = Begat

![Diagram showing Martin to George.]

Last modified: Thu Apr 14 15:01:19 2022
More Examples

- Edge = some relationship

\[
\begin{array}{c}
\text{potstickers} \quad \text{eats} \quad \text{John} \quad \text{loves} \quad \text{Mary}
\end{array}
\]

- Edge = next state might be (with probability)

\[
\begin{array}{c}
\text{hat} \quad 0.4 \quad \text{the} \quad 0.6 \quad \text{cat} \quad 0.4 \quad \text{in} \quad 0.1 \quad \text{bed}
\end{array}
\]

- 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”.)

\[
\begin{array}{c}
s \quad 0 \quad 2 \quad 3 \quad 4 \quad 0.1
\end{array}
\]

Last modified: Thu Apr 14 15:01:19 2022
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).

\[ a \xrightarrow{(2,3)} b \xrightarrow{} c \]

\[ \begin{array}{c|c}
1: & a \xrightarrow{(2,3)} \\
2: & b \xrightarrow{(3)} \\
3: & c \xrightarrow{(1,2)} \\
\end{array} \]

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

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

- **Adjacency matrix**: Represent connection with matrix entry:

\[
\begin{bmatrix}
1 & 2 & 3 \\
1 & 0 & 1 & 1 \\
2 & 0 & 0 & 1 \\
3 & 0 & 0 & 0 \\
\end{bmatrix}
\]
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:

```
0 1 2 3 4 5 6 7 ... 3N
```

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).
Recursive Depth-First Traversal of a Graph

- Can fix looping and combinatorial problems using the “bread-crumbs” method used in earlier lectures for a maze.
- That is, *mark* nodes as we traverse them and don't traverse previously marked nodes.
- Makes sense to talk about *preorder* and *postorder*, as for trees.

```c
void preorderTraverse(Graph G, Node v) {
    if (v is unmarked) {
        mark(v);
        visit v;
        for (Edge(v, w) ∈ G) 
            traverse(G, w);
    }
}

void postorderTraverse(Graph G, Node v) {
    if (v is unmarked) {
        mark(v);
        for (Edge(v, w) ∈ G) 
            traverse(G, w);
        visit v;
    }
}
```
Recursive Depth-First Traversal of a Graph (II)

• We are often interested in traversing all nodes of a graph, not just those reachable from one node.
• So we can repeat the procedure as long as there are unmarked nodes.

```java
void preorderTraverse(Graph G) {
    clear all marks;
    for (v ∈ nodes of G) {
        preorderTraverse(G, v);
    }
}

void postorderTraverse(Graph G) {
    clear all marks;
    for (v ∈ nodes of G) {
        postorderTraverse(G, v);
    }
}
```
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.

Graph (two views) Possible Orderings

```
A   C   C
B   A   F
C   C   G
D   A
E   F
G   A
   B
H   D
```

```
A
B
C
D
E
F
G
H
```

Last modified: Thu Apr 14 15:01:19 2022

CS61B: Lecture #33
**Observation:** Suppose we *reverse the links* on our graph.

- If we do a recursive DFS on the reverse graph, starting from node H, for example, we will find all nodes that must come *before* H.
- When the search reaches a node in the reversed graph and there are no successors, we know that it is safe to put that node first.
- In general, a *postorder* traversal of the *reversed* graph visits nodes only after all predecessors have been visited.

Numbers show post-order traversal order starting from G: everything that must come before G.
**General Graph Traversal Algorithm**

```java
COLLECTION_OF_VERTICES fringe;

fringe = INITIAL_COLLECTION;
while (!fringe.isEmpty()) {
    Vertex v = fringe.REMOVE_HIGHEST_PRIORITY_ITEM();

    if (!MARKED(v)) {
        MARK(v);
        VISIT(v);
        For each edge(v,w) {
            if (NEEDS_PROCESSING(w))
                Add w to fringe;
        }
    }
}
```

Replace `COLLECTION_OF_VERTICES, 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.

// Red sections are specializations of general algorithm
Stack<Vertex> fringe;

fringe = stack containing \{v\};
while (!fringe.isEmpty()) {
    Vertex v = fringe.pop();

    if (!marked(v)) {
        mark(v);
        VISIT(v);
        For each edge(v,w) {
            if (!marked(w))
                fringe.push(w);
        }
    }
}
Depth-First Traversal Illustrated

Marked: \( z \)

Fringe: \([a]\) \([b, d]\) \([c, e, d]\) \([d, f, e, d]\) \([f, e, d]\) \([e, e, d]\) \([e, d]\) \([\]\\)
Topological Sort Revisited

- Another approach to topological sorting uses the general traversal algorithm.
- We keep a sequence of output nodes, initialized to empty.
- For each node, maintain a count of the number of immediate predecessor nodes (the number of nodes with edges going to that node) that have not yet been output.
- At each step, output a node with no remaining predecessors, and update the predecessor counts.

```java
Set<Vertex> fringe; Map<Vertex, Integer> predCount;
fringe = empty set of vertices;
predCount = map from each vertex to number of predecessors;
while (!fringe.isEmpty()) {
    Vertex v = fringe.removeSomeElement();  /* Not a real operation! */
    Add v to the output;
    for each edge(v,w) {
        predCount.put(w, predCount.get(w) - 1);
        if (predCount.get(w) == 0) fringe.add(w);
    }
}
```

Last modified: Thu Apr 14 15:01:19 2022
Alternative Topological Sort in Action

Output: []

Output: [A]

Output: [A, C]

Output: [A, C, B]

Output: [A, C, B, F]

Output: [A, C, B, F, D]

Output: [A, C, B, F, D, E, G, H]
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 $s$.

```java
PriorityQueue<Vertex> fringe;
For each node v { v.dist() = \infty; 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

Final result:

--- Shortest-path tree

X\(d\) processed node at distance \(d\)

Y\(d\) node in fringe at distance \(d\)