CS61B Lecture #33

Today's Readings:  Graph Structures: DSIJ, 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 graph with no cycles.

- Example:
  - Nodes: a, b, c, d
  - Edges: a → b, b → c, c → d, d → a

**Undirected**

- A graph with no directed edges.

- Example:
  - Nodes: a, b, c, d, e
  - Edges: a → b, b → e, c → d

**Cyclic:**

- A graph containing at least one cycle.

- Example:
  - Nodes: a, b, c, d
  - Edges: a → b → c → a

**With Edge Labels:**

- A graph with labels on its edges.

- Example:
  - Nodes: a, b, c, d, e
  - Edges: a → b (label 1), b → c (label 1), c → d (label 2), d → a (label 3), e → b (label 0)
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:

  ![Connected Graph](image1)
  ![Unconnected Graph](image2)
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](image1)

![Unilaterally Connected](image2)

![Strongly Connected](image3)
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 of Detroit to Chicago with a length of 200 miles.]

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

  ![Diagram of Eat (1 hr) to Sleep (8 hrs).]

- **Edge = Begat**

  ![Diagram of Martin to George.]

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

- Edge = some relationship

- Edge = next state might be (with probability)

- 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”.)
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).

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

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

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

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

Treating 0 as the root and doing 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.

```
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

\[
\begin{array}{ccc}
A & C & C \\
C & A & F \\
D & B & G \\
E & D & A \\
F & C & B \\
G & E & D \\
H & H & H \\
\end{array}
\]
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.
General Graph Traversal Algorithm

\textit{COLLECTION\_OF\_VERTICES} fringe;

fringe = \textit{INITIAL\_COLLECTION};
while (\textit{fringe}.isEmpty()) {
    \textit{Vertex v} = \textit{fringe}.\texttt{REMOVE\_HIGHEST\_PRIORITY\_ITEM}();

    if (!\textit{MARKED(v)}) {
        \textit{MARK(v)};
        \textit{VISIT(v)};
        \texttt{For each edge(v,w)} {
            \textit{if (NEEDS\_PROCESSING(w))}
            \texttt{Add w to fringe;}
        }
    }
}

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.

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

fringe = \textit{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.

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);
    }
}
Alternative Topological Sort in Action

Output: []

[ ]

[A]

[A, C]

[A, C, B]

[A, C, B, F]

[A, C, B, F, D]

[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() = ∞; 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; }
    }
}
```

Last modified: Thu Apr 14 15:01:19 2022 CS61B: Lecture #33 21
Example

Final result:

Shortest-path tree

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

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