Today: Shortest Paths (Ch 24, 25, DataStr. B Ch 22) (L#10,11)

Types of Problems

1) Single Origin - Single destination
2) Single Origin -(all destinations)
3) All pairs: All origin/destination combinations

(2) included (11) and currently best way to solve (11) for most cases.

For all input: A directed simple graph \( G = (V; E) \)
with edge weights \( w(e), e \in E \).

For (2), a special node \( \$ \) (origin)
We will deal with (2) first and then move to (3).

Single Origin SP

Three algorithms depending on input data.

(a) (Directed) Acyclic Algorithm
(b) Dijkstra Algorithm
(c) Bellman-Ford Algorithm

(a) require \( G \) to have no directed cycles.
Edge weights can be arbitrary.
**DAG algorithm**

First we need to check whether or not input graph is acyclic. There are two ways to test one more efficient than the other. We will show both since there are important lessons from each.

**Claim:** If G is acyclic, there exists a vertex with no incoming edge.

**Proof (Pf):** Suppose not. 1 has incoming edge say from 2. 2 has incoming edge, say from 3 and so on.

At some point, we must repeat a node (since our graphs have finite # of vertices). This shows A has a cycle which is a contradiction.
Call this vertex 1. If we remove 1 from G, the new graph is acyclic and it has such a vertex → Number 2. and so on. So graph looks like

1 → 2 → 3 → ... → n

All edge go from Smaller number to a larger #. So no cycle.

This ordering of vertices (which is only possible for DAG) is called a topological ordering And the process is called Topological Sorting.

Topological Sorting can only be done for DAG.

When you don’t know if G is acyclic, you CANNOT topologically sort.
More efficient way is to use DFS (Depth First Search). Although DFS can be done on undirected graphs, we want a version on directed graphs with the sole purpose of checking if it is acyclic and if so get a topological order.

So we are adapting this to our purposes. (See Ch 22), we will do this as a computer program.

Input: A list of vertices and an adjacency list that gives outgoing connections. (Edges are not needed for this)

An example will best illustrate this:

<table>
<thead>
<tr>
<th>V</th>
<th>Adj(u)</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>[B, C]</td>
</tr>
<tr>
<td>B</td>
<td>[C, E]</td>
</tr>
<tr>
<td>C</td>
<td>[D, F]</td>
</tr>
<tr>
<td>D</td>
<td>[E, F]</td>
</tr>
<tr>
<td>E</td>
<td>[G, H]</td>
</tr>
<tr>
<td>F</td>
<td>[H]</td>
</tr>
<tr>
<td>G</td>
<td>[H, I]</td>
</tr>
<tr>
<td>H</td>
<td>[I]</td>
</tr>
<tr>
<td>I</td>
<td>[∅]</td>
</tr>
</tbody>
</table>

Given input with adj lists,

Do NOT DRAW the graph

Do DFS on adj lists directly and use vertices in the order provided in the adj list.

This G is not DAG. (Ex: Show this)

I: [∅] no outgoing edge // It means no node has
DFS output

1. Two time stamps for each node: first one is called START TIME and the second is FINISH TIME. These are distinct (no two equal) integers from 1 to 2|V|.

2. A DFS Forest consisting of directed trees.

3. Edges classified into one of four mutually exclusive and collectively exhaustive groups:
   - (a) In-tree or In-Forest edge
   - (b) FORWARD edge
   - (c) CROSS edge
   - (d) BACK edge

// The presence of these indicates a cycle: G is not a DAG. We stop when this happens after completing DFS on the entire graph.

Now DFS algorithm

1. Select (arbitrarily) a node to start the process.
   (This will also be done if we return to this node and not yet processed the entire graph.)
   Give it the next time stamp (not used so far) as its START TIME.

2. Go to the first node in the adjacency list of the node selected in Step 1.
3. If the node is new (unprocessed so far) it gets the next time stamp as START TIME, and go to this node in Step 2.

4. If it is already processed, this edge is classified into FORWARD, CROSS, or BACK as appropriate (we will discuss this later) and go to step 2.

The description is more complicated than the example.

Example: *: Arbitrary decision.
Black numbers: START TIME
Red "": FINISH TIME

5. When we have no unprocessed node in the adjacency list, the node gets the next number as FINISH TIME stamp. (This is shown in the next figure on the next page.) And we back track on the tree structure to its parent node.
When the next node in the adj list (as in adj(G)) is to a node which has both START and FINISH time stamps, this edge will be classified as the edge (G, I). Since it starts at an Ancestor G and goes to a descendant I, this edge is a FORWARD edge as shown above. The next such edge is (E, H) also a FORWARD edge. The one after this is (F, H) which is neither from an ancestor to a descendant nor from a descendant to an Ancestor. Such edges will be classified as CROSS edges.

When we come back to starting point, and there are more nodes, select arbitrarily one.
A **BACK** edge will go from a descendant to an ancestor (just the opposite of **FORWARD**). If there are **BACK** edges, G is not a **Cyclic graph** and hence **acyclic** (DAG).

**A Topological order**: Decreasing order of finish time stamps.

A B C D F E G H I
18 17 16 12 11 9 8 7 6

DFS is now complete.
Complexity of DFS: $\Theta(|E|+|V|)$

Linear time for graph profile

DFS is considered "pre-processing" for DAG shortest path alg to determine applicability and if applicable to get topological order which is needed in DAG shortest path algorithm.

Now we describe some common aspects of all these algorithms. First some notation.

$\delta(s,v) =$ the length of a shortest path from $s$ to $v$ in $(G,w)$.

This depends only on the input data and is independent of algorithm used.

$d[v]: \text{"Current estimate" of } \delta(s,v) \text{ in an algorithm (i.e. alg. dependent)}$

[Book uses $u,d \rightarrow \text{object oriented notation}]

$T[e]$:

\[0 \rightarrow O \rightarrow \cdots \rightarrow O \rightarrow O\]

\[\leftarrow \text{Shortest Path or its \"current est\"}

Needed for recovering path.
All algorithms (Dijkstra, Bellman-Ford) use the following subroutine, which we describe now:

**INITIALIZE \((G, \delta)\)**

1. \(d[v] \leftarrow \infty \quad \forall v \in V\) \(\Theta(n)\)
2. \(\pi[v] \leftarrow \text{NIL} \quad \forall v \in V\)
3. \(d[\delta] \leftarrow 0\)

All algorithms use the following subroutine as their basic operation:

**RELAX \((u, v, w)\)** // Relax edge \((u, v)\) with weight \(w(u, v)\).

if \(d[v] > d[u] + w(u, v)\)

then \(d[v] \leftarrow d[u] + w(u, v)\)

\(\pi[v] \leftarrow u\)

else pass: do nothing and so not specified.

The algorithms differ in the number of times an edge is relaxed and the order in which they are done.
DAG Algorithm

After INITIALIZE is done, (after DFS is done and there are no BACK edges)
RELAX is done on all outgoing edge
out of a node; nodes are processed in the topological order. (regardless of edge weights)

Remark: If there are nodes that occur earlier than S (origin), then they can be removed as a path for these nodes is \( \infty \).

When all processing is done, \( \text{d}[W] = 8/8, v \)
and \( \text{F}[W] \) is the node that occurs just prior to \( v \) in a SP from \( S \) to \( W \).

In lecture notes, following Coloring Scheme is used:

**Nodes:**
- **Blue:** Currently processed node
- **Yellow:** Completed processing
- **White:** Yet to be processed

**Edges:**
- **Blue:** Currently being RELAXed
- **Green:** yet to be processed or RELAXed
- **Red:** permanent in \( S(8,8) \)
At the end (when all nodes have been processed), the \textcolor{red}{RED} edge form a directed tree rooted at $s$ (origin): This means nodes in the tree have a directed path from $s$ which represents a shortest path from $s$ to these nodes. Nodes not in the tree do not have a directed path from the origin. Proof of correctness will be given later (see Sec 11) (Ch 24). In this algorithm, each edge gets \textit{relaxed} at most once and the order is determined by the graph $G$ and \textbf{not} the weights.

\underline{Dijkstra Algorithm}

Here the graph may have cycles but this alg. requires \underline{edge weights $w(u,v) \geq 0 \forall (u,v) \in E$} for applicability. \cite{[There are a few exceptions, one of which is in your assignment #6]. But we will assume $w(u,v) \geq 0 \forall (u,v) \in E$}

Here also we first \underline{initialize}. Then, edges going out of origin nodes in \underline{and increasing order of $d[u]$} values are \underline{relaxed}. 
Hence, at each step, among the nodes yet to be processed, we select the node with minimum value of $d[v]$ to process (RELAX) its outgoing edges (in any order). So here the order depends on the edge weights as well as the graph. But each edge is RELAXed at most once as in DAG algorithm. This is common to both these algorithms.

An example is worked out in Lecture Notes #10. This algorithm uses Min-heap to select the next node for processing. Key values are $d[v]$.

Color Coding: Same as in DAG algorithm.

End result: RED tree rooted at $s$ (origin) indicating shortest paths.

What to do if $G$ is not Acyclic and if there are edge weights that are negative.

Neither DAG algorithm nor Dijkstra algorithm are applicable. Now we use Bellman-Ford alg.