Solutions -- Topic 12, Dynamic Programming

Longest Simple Path in a Directed Acyclic Graph

Given a **directed weighted acyclic graph G=(V,E)** and two vertices **s** (start) and **t** (target), develop a **dynamic programming approach** for finding a **longest weighted simple path** from **s** to **t**.

1. Characterize the Structure of an Optimal Solution

Let **p** be a **longest path from u to t**. If u = t then p is simply ⟨u⟩ and has zero weight. Consider when u ≠ t. Then p has at least two vertices and looks like: **p = ⟨u, v … t⟩** (it is possible that v = t).
Let **p' = ⟨v … t⟩** and **prove that p' must be a longest simple path from v to t.** (This is a proof of optimal substructure.)

> **Solution:**
> (Cut and paste argument:) Suppose that p' as defined above were *not* a longest simple path from v to t. Then there must exist some path p'' that is a longer simple path from v to t; that is (extending our w notation), w(p'') > w(p'). We can construct a new path p* from u to t consisting of u → v → p'' → t. This is a legal path because G is acyclic, so u cannot occur in p''. The length of this path p* is
>
> $$w(p^*) = w(u,v) + w(p'') > w(u,v) + w(p') = w(p)$$
>
> contradicting our definition of p as the longest simple path from u to t. Therefore, p' must be a longest simple path from v to t, and the problem exhibits optimal substructure.

2. Recursively define the value of an optimal solution:

Let **dist[u]** be the distance of a longest path from u to t. **Fill out the definition to reflect the above structure:**

$$\text{dist}[u] = \begin{cases} 0 & \text{if } u = t \\ \max_{v \in Adj[u]} \{w(u,v) + \text{dist}[v]\} & \text{if } u \neq t \end{cases}$$

3. Compute the value of an optimal solution (simple recursive version):

Write a recursive procedure that computes the value of an optimal solution as defined by the above recursive definition. <u>Do not memoize yet;</u> that's the next step.

Solution: (Notice how the code follows the mathematical definition.)

```
Longest-Path-Value-Recursive (G, u, t)
  if u = t
      return 0
  else
      max = -∞
      for each v in G.Adj[u]
          alt = w(u,v)
              + Longest-Path-Value-Recursive(G, v, t)
          if max < alt
              max = alt
      return max
```

<u>4. Compute the value of an optimal solution (dynamic programming version):</u>
Memoize your procedure by passing the array `dist[1..|V|]` that records longest path distances `dist[u]` from each vertex **u** to **t**. Assume that the **caller has initialized all entries of `dist` to -∞**.

Solution (additions highlighted):

```
Longest-Path-Value-Memoized (G, u, t, dist)
  if u = t
      dist[u] = 0  // caller may expect all entries defined
      return 0
  if dist[u] > -∞
      return dist[u] // this is the reuse of prior result
  else
      // dist[u] = -∞ is not necessary as caller did it
      for each v in G.Adj[u]
          alt = w(u,v)
              + Longest-Path-Value-Memoized(G, v, t, dist)
          if dist[u] < alt
              dist[u] = alt
      return dist[u]
```

<u>5. Analyze the runtime of your solution in #4 in terms of |V| and |E|</u>

Include

(a) the runtime to initialize **`dist`** and

(b) the runtime of **`Longest-Path-Value-Memoized`** itself.

(c) the resulting total runtime

This requires aggregating across loops.

(a) $\Theta(|V|)$ to fill in the entries.

(b) $\Theta(|E|)$ since in aggregate across all calls each edge is processed once(*), and all other operations are constant. (*) can be argued two ways: it is a DAG, so we never return to the same vertex, and even if we did we would just be doing constant lookup of paths we saved in dist.

(c) Total: $\Theta(|V| + |E|)$.