Dynamic Programming II [1]

Jie Wang

University of Massachusetts Lowell
Department of Computer Science

[1] I thank Prof. Zachary Kissel of Merrimack College for sharing his lecture notes with me; some of the examples presented here are borrowed from his notes.

For a change we will look at a complexity-theoretical problem to demonstrate how we can use DP to solve decision problems.

- Let A be a language over a finite alphabet.
- The Kleene closure of A, denoted by A^*, is defined as follows:

$$A^* = \{x \mid x \text{ is a finite string over } A\}.$$

- Let P denote the set of languages accepted by polynomial-time bounded deterministic Turing machines.

Theorem. If $A \in P$, then so is A^*.
Proof. Let M_A be a DTM with time bound p_A (a polynomial) accepting A. That is,

$$M_A(x) = \begin{cases} 1, & \text{if } x \in A, \\ 0, & \text{if } x \notin A. \end{cases}$$

Proof Continued

Observation:

- $x \in A^*$ iff $x \in A$ or $x = x_1 x_2$ such that $x_1 \in A$ and $x_2 \in A^*$, where $|x_1||x_2| > 0$.

Let $|x| = n$.

Formulation: Given $i \leq n$, let

$$KC(x, M_A, i, n) = \begin{cases} 1, & \text{if substring } x[i..n] \in A^*, \\ 0, & \text{otherwise.} \end{cases}$$

- There are n subproblems.

Localization: $KC(x, M_A, i, n) = 1$ iff one of the following conditions hold:

- $M_A(x[i..n]) = 1$.
- $x[i..n] = x[i..j]x[j+1, n]$ for some $j \in [i, n)$ such that $M_A(x[i..j]) = 1$ and $KC(x, M_A, j+1, n) = 1$.

Bottom Up

$\mathrm{KC}(x, M_A, i, n)$

$T[n+1] = 1$
for $j = i$ **to** n
    $T[j] = 0$
for $j = n$ **to** i
    for $k = j$ **to** n
        if $M_A(x[i..k]) == 1$ **and** $T(k+1) == 1$
            $T[k] = 1$
return $T[i]$

Compute $KC(x, M_A, 1, n)$. If $KC(x, M_A, 1, n) = 1$ then $x \in A^*$; otherwise, $x \notin A^*$.
Running time: $O(n^2 p_A(n))$. Thus, $A^* \in P$. **End of Proof**

Edit Distance

Now back to optimization. Suppose that we want to determine if string S_1 is "similar" to S_2. This is a very active and real world problem. Applications include

- Cheating detection
- Copyright infringement detection
- Determining similarity of two DNA sequences (e.g., finding familial relationships)
- Auto correction
- Topic discoveries
- Summary extraction

We will measure the similarity of two strings using a metric called the *edit distance*.

- The Levenshtein metric.

Problem Description

When calculating the (Levenshtein) edit distance between strings S_1 and S_2 we are looking for how many operations it takes to transform S_1 into S_2.

1. Insert a character c.
2. Delete a character c at location i.
3. Replace a character c with c' at a location i.
    - Sometimes called a substitution.

Formalize the edit distance problem as follows:
Input: Two strings X and Y.
Output: The minimum cost of edit operations (insert, delete, and replace) to transform X into Y.

Solving this problem is similar to solving LCS.

Formulation and Localization

Formulation: Given a string $X = x_1 x_2 \cdots x_m$ and a string $Y = y_1 y_2 \cdots y_n$. Let $D(i, j)$ denote the least number of operations to turn suffix $X_i = x_i \cdots x_m$ into suffix $Y_j = y_j \cdots y_n$.

- There are mn subproblems.

Localization: We can arrive at the value of $D(i, j)$ by considering the following three cases:

1. Insert y_j before x_i.
   - This makes X longer. Note: This operation doesn't examine X.
2. Delete x_i.
   - This makes X shorter.
3. Replace x_i with y_j.
   - This does *not* change $|X|$.

Denote

- insertion of character a by $\uparrow a$,
- removal of character a by $\not a$,
- replacement of a with b by $a \to b$.

Localization Continued

- Inserting character y_j before x_i forces a match. However, we still know nothing about x_i.
  - This means we should consider the subproblem $D(i, j + 1)$.
  - Note: we are not actually performing any edit on the string. There is nothing dynamic about the strings.
- Deleting x_i learns nothing about y_j.
  - This means we should consider the subproblem $D(i + 1, j)$.
- Replacing x_i with y_j we know that x_i is now equal to y_j and we have a perfect match up to this point.
  - This means we should consider the subproblem $D(i + 1, j + 1)$.
- We also have two special cases that aren't covered by our edit operations; these aren't really operations at all.
  - If $x_i = y_j$ we should just skip the match and look at subproblem $D(i + 1, j + 1)$.
  - If we are trying to read past the end of one of our string (i.e., $i > m$ or $j > n$) our edit distance is 0.

Localization Continued

Define our recurrence as follows:

$$D(i,j) = \begin{cases} 0, & \text{if } i > m \text{ or } j > n, \\ D(i+1, j+1), & \text{if } x_i = y_j, \\ \min\{C(\uparrow y_j) + D(i, j+1), & \text{if } i \le m, j \le n, \text{ and } x_i \ne y_j, \\ \qquad C(\not{x}_i) + D(i+1, j), & \\ \qquad C(x_i \to y_j) + & \\ \qquad + D(i+1, j+1)\} & \end{cases}$$

where C is a cost function.

Want to compute $D(1,1)$.

Memoization

Use a global memo pad $memo[1..m, 1..n]$ with all entries initialized to $\perp$.

$\textsc{EditDistance}(i, j, X[1..m], Y[1..n])$

if $memo[i,j] \neq \perp$
    $v = memo[i,j]$
elseif $i \leq m$ **and** $j \leq n$ **and** $X[i] \neq Y[j]$
    $v = \min\{C(\uparrow y_j) + \textsc{EditDistance}(i, j+1),$
        $C(\not{x}_i) + \textsc{EditDistance}(i+1, j),$
        $C(x_i \to y_j) + \textsc{EditDistance}(i+1, j+1)\}$
elseif $X[i] == Y[j]$
    $v = \textsc{EditDistance}(i+1, j+1)$
elseif $i > m$ **or** $j > n$
    $v = 0$
$memo[i,j] = v$
return v

Running time: $\Theta(mn)$.

Connect to LCS

We can also work on prefixes of the string and generate a recurrence D', and we want to compute $D'(m, n)$.

- This is what we did when we looked at the LCS problem.
- The above becomes the LCS problem if we make $C(x_i \to y_j) = \infty$ for all i and j.
  - Deletions and insertions are basically equivalent to skipping over characters that don't match.

Single-Source Shortest Path

Suppose that we need to find the shortest path in a graph from a given source vertex to all other vertices in the graph This problem has many applications; for example,

- transportation planning
- Packet routing in communication networks
- Friend discovery in social networking
    - Think of friend recommendations in Facebook.
- Speech recognition

This problem can be formally described as follows:
Input: A weighted directed graph $G = (V, E)$ and a source vertex $s \in V$.
Output: The set of shortest paths

$$\left\{ p \mid s \overset{p}{\rightsquigarrow} v \text{ is a shortest path from } s \text{ to } v \text{ where } v \in V \right\}.$$

First DP Attempt

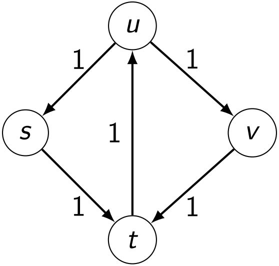
Formulation: Let $\delta(s, v)$ denote the weight of the shortest path $s \overset{p}{\leadsto} v$; i.e., the summation of weights on each edge of the path is δ.
Localization: The shortest path can be divided into smaller problems.

- Note that if there exists a path from u to v, then there must be some vertex $t \in p$ such that $(t, v) \in E$
- Thus, $\delta(u, v) = \delta(u, t) + w(t, v)$ for some t.
- There are $\deg^-(v)$ many t's, the in-degree of v.

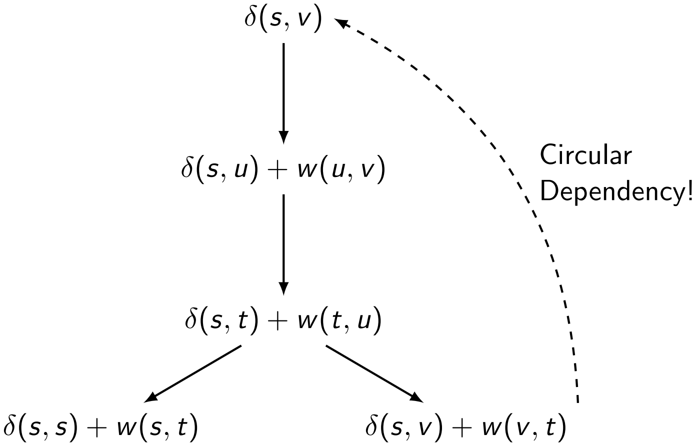
$$\delta(s, v) = \begin{cases} 0, & \text{if } s = v, \\ \min_{(t,v)\in E}\{\delta(s, t) + w(t, v)\}, & \text{otherwise.} \end{cases}$$

Does It Work?

This dynamic program has infinite running time when G has a cycle
For example, consider the following graph:



Recursion Tree for $\delta(s, v)$



The recursion tree for a graph with cycles has a circular dependency. Thus, the subproblem graph is not a DAG and therefore we can't use DP.

Make DP Work

Consider the following observations on $\delta(s, t)$.

- If the graph doesn't have negative weight edges, then a shortest path must contain no cycle; i.e., any shortest path must be a *simple path*.
- The maximum number of edges in a shortest path $s \overset{p}{\leadsto} t$ is $|V| - 1$.
- Construct a DP whose recursion tree is depth limited by the maximum number of edges possible in any maximum length simple path (i.e., $|V| - 1$).
- Let $\delta_k(s, v)$ denote the minimal weight path $s \overset{p}{\leadsto} v$ such that $|p| \leq k$.
- We tune up the recurrence relation to obtain a working DP:

$$\delta_k(s, v) = \begin{cases} 0, & \text{if } k = 0 \text{ or } s = v, \\ \min_{(t,v)\in E}\{\delta_{k-1}(s, t) + w(t, v)\}, & \text{if } k > 0 \text{ and } s \neq v. \end{cases}$$

- Want to compute $\{\delta_{|V|-1}(s, v) \mid v \in V\}$.

Memoization (A Version of Bellman-Ford)

$\textsc{ShortestPath}(E, V, s, v, k)$

```
if memo(u, v) ≠ ⊥
    v = memo(u, v)
elseif k == 0
    return 0
elseif u == v
    v = 0
elseif k > 0
    min = ∞
    for (t, v) ∈ E
        v = ShortestPath(E, V, s, t, k − 1) + w(t, v)
        if v < min
            min = v
memo(u, v) = v
return v
```

Single Source Shortest Paths

$\text{SINGLESOURCESHORTESTPATH}(E, V, s)$

```
Allocate array R[1 . . |V|)
for v ∈ V
    R[v] = ShortestPath(E, V, s, v, |V| − 1)
return R
```

Running time: $\Theta(|V||E|)$:

- For every vertex t visited in SHORTESTPATH, we perform $\deg^-(t)$ work. This gives $\sum_{t\in V} \deg^-(t) = |E|$.
- Algorithm SINGLESOURCESHORTESTPATH makes $|V|$ calls to SHORTESTPATH.