CS3383 Unit 3.2: Dynamic Programming Examples

David Bremner

March 1, 2018

Outline

- Dynamic Programming
  - Longest increasing subsequence
  - Edit Distance

Contents

Longest increasing subsequence problem

Input Integers $a_1, a_2 \ldots a_n$

Output

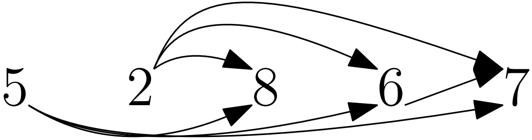
$$a_{i_1}, a_{i_2}, \ldots a_{i_k}$$

Such that

$$i_1 < i_2 \cdots < i_k$$

and

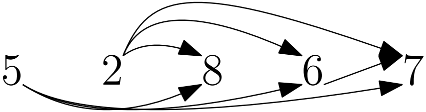
$$a_{i_1} < a_{i_2} < \cdots < a_{i_k}$$



- $(a_i, a_j) \in E$ if $i < j$ and $a_i < a_j$.
- DPV 6.2, JE 5.2

Defining subproblems

- Define $F(i)$ as the length of longest sequence starting at position i



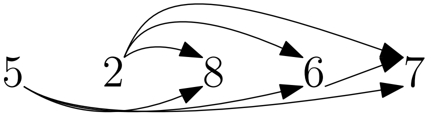
$$F(i) = 1 + \max\{F(j) \mid (i,j) \in E\}$$

- Topological sort is trivial

- We could solve this reasonably fast e.g. by memoization.

Defining subproblems

- Define $F(i)$ as the length of longest sequence starting at position i
- We could do n longest path in DAG queries.



$$F(i) = 1 + \max\{F(j) \mid (i,j) \in E\}$$

- Topological sort is trivial

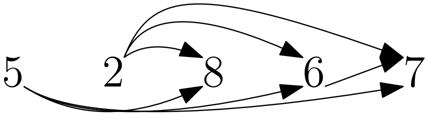
- We could solve this reasonably fast e.g. by memoization.

Defining subproblems

- Define $F(i)$ as the length of longest sequence starting at position i
- We could do n longest path in DAG queries.
- Thinking recursively:

$$F(i) = 1 + \max\{F(j) \mid (i, j) \in E\}$$

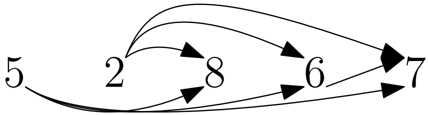
- We could solve this reasonably fast e.g. by memoization.



- Topological sort is trivial

Longest path in DAG, working backwards

- Define $L[i]$ as the longest path *ending* at a_i



```
For i = 1…n:
    L[i] = 1 + max { L(j) | (j,i) in E }
```

- total cost is $O(|E|)$, *after* computing E.

Improving memory use

- We can inline the definition of E.

Improving memory use

- We can inline the definition of E.
- $L(i) = 1 + \max\{L(j) \mid j < i \text{ and } a_j < a_i\}$

Improving memory use

- We can inline the definition of E.
- $L(i) = 1 + \max\{L(j) \mid j < i$ and $a_j < a_i\}$

```
function $\mathrm{LIS}(a_1 \ldots a_n)$
    $\forall i\ L[i] = -\infty$
    for $i \in 1 \ldots n$ do
        for $j \in 1 \ldots i-1$ do
            if $a_j < a_i$ then
                $L[i] \leftarrow \max(L[i], L[j]+1)$
            end if
        end for
    end for
    return $\max(L[1] \ldots L[n])$
end function
```

Contents

Edit (Levenshtein) Distance

- DPV 6.3, JE5.5
- Minimum number of insertions, deletions, substitutions to transform one string into another.

Example: timberlake → fruitcake

- Using mostly insertions and deletions

```
i i i i   d d d d d s
_ _ _ _ T I M B E R L A K E
F R U I T _ _ _ _ _ C A K E
```

Total cost 10.

Edit (Levenshtein) Distance

- DPV 6.3, JE5.5
- Minimum number of insertions, deletions, substitutions to transform one string into another.

Example: timberlake → fruitcake

- Using more substitutions

```
s s s s s d s
T I M B E R L A K E
F R U I T _ C A K E
```

Total cost 7.

Alignments (gap representation)

```
1 1 1 1 0 1 1 1 1 1 1 0 0 0
_ _ _ _ T I M B E R L A K E
F R U I T _ _ _ _ _ C A K E
```

- top line has letters from A, in order, or `_`
- bottom line has has letters from B or `_`
- cost per column is 0 or 1.

Alignments (gap representation)

```
1 1 1 1 0 1 1 1 1 1 1 0 0 0
_ _ _ _ T I M B E R L A K E
F R U I T _ _ _ _ _ C A K E
```

- top line has letters from A, in order, or _
- bottom line has has letters from B or _
- cost per column is 0 or 1.

Theorem (Optimal substructure)

If we remove any column from an optimal alignment, we have an optimal alignment for the remaining substrings.

Alignments (gap representation)

Theorem (Optimal substructure)

If we remove any column from an optimal alignment, we have an optimal alignment for the remaining substrings.

proof.

By contradiction □

Subproblems (prefixes)

- Define $E[i, j]$ as the minimum edit cost for $A[1 \dots i]$ and $B[1 \dots j]$

$$E[i,j] = \begin{cases} E[i, j-1] + 1 & \text{insertion} \\ E[i-1, j] + 1 & \text{deletion} \\ E[i-1, j-1] + 1 & \text{substition} \\ E[i-1, j-1] & \text{equality} \end{cases}$$

justification.

We know deleting a column removes an element from one or both strings; all edit operations cost 1. ☐

order of subproblems

$$E[i,j] = \begin{cases} E[i-1,j]+1 & \text{deletion} \\ E[i,j-1]+1 & \text{insertion} \\ E[i-1,j-1]+1 & \text{substition} \\ E[i-1,j-1] & \text{equality} \end{cases}$$

- dependency of subproblems is *exactly* the same as LCS, so essentially the same DP algorithm works.

order of subproblems

$$E[i,j] = \begin{cases} E[i-1,j]+1 & \text{deletion} \\ E[i,j-1]+1 & \text{insertion} \\ E[i-1,j-1]+1 & \text{substition} \\ E[i-1,j-1] & \text{equality} \end{cases}$$

- dependency of subproblems is *exactly* the same as LCS, so essentially the same DP algorithm works.
- or just memoize the recursion

order of subproblems

$$E[i,j] = \begin{cases} E[i-1,j]+1 & \text{deletion} \\ E[i,j-1]+1 & \text{insertion} \\ E[i-1,j-1]+1 & \text{substition} \\ E[i-1,j-1] & \text{equality} \end{cases}$$

- dependency of subproblems is *exactly* the same as LCS, so essentially the same DP algorithm works.
- or just memoize the recursion
- what are the base cases?