Cosc 201
Algorithms and Data Structures
Lecture 23 (19/5/2025)
Dynamic Programming 2

Brendan McCane
brendan.mccane@otago.ac.nz

UNIVERSITY OF OTAGO
Te Whare Wānanga o Otāgo
NEW ZEALAND

Edit Distance

- The unix utility diff uses edit distance to find differences between two files (or strings).
- Diff, or something like it, are used in all sorts of places, but most notably in version control systems such as git.
- Edit distance can also be used to fix spelling errors.
- Edit distance is very similar to LCS.
- In LCS, we effectively only allow deletion from one string or the other in each step.
- In edit distance, we try to transform one string into the other by allowing operations: insert, delete, or substitute.
- For example, what are the fewest number of edits required to transform "morena" into "morning"?

Edit distance recurrence

The recurrence for the edit distance is (transform X to Y):

$$d_{ij} = \begin{cases} d_{i-1,j-1}, & \text{if } X[i] = Y[j] \\ \min \begin{cases} d_{i-1,j} + 1, & \text{delete } X[i] \\ d_{i,j-1} + 1, & \text{insert } Y[j] \\ d_{i-1,j-1} + 1, & \text{substitute } Y[j] \text{ for } X[i] \end{cases}, & \text{if } X[i] \neq Y[j] \end{cases}$$

where

$d_{0,j} = j$: insert all characters up to $Y[j]$

and

$d_{i,0} = i$: delete all characters up to $X[i]$

Naive algorithm

```
1:  function EDITDISTANCE(X,Y)
2:      if m=0 then return n
3:      end if
4:      if n=0 then return m
5:      end if
6:      if X[m]=Y[n] then
7:          return EditDistance(X[1...m-1],Y[1...n-1])
8:      end if
9:      delxi = 1+EditDistance(X[1...m-1],Y)
10:     insyj = 1+EditDistance(X,Y[1...n-1])
11:     sub = 1+EditDistance(X[1...m-1],Y[1...n-1])
12:     return min(delxi, insyj, sub)
13: end function
```

Memoised algorithm

```
1:  initiallise memo as 2D array, set all distances to -1
2:  function EDITDISTANCE(X,Y)
3:      if m=0 then return n
4:      end if
5:      if n=0 then return m
6:      end if
7:      if memo[m,n]!=-1 then return memo[m,n]
8:      end if
9:      if X[m]=Y[n] then
10:         memo[m,n] = EditDistance(X[1...m-1],Y[1...n-1])
11:     else
12:         delxi = 1+EditDistance(X[1...m-1],Y)
13:         insyj = 1+EditDistance(X,Y[1...n-1])
14:         sub = 1+EditDistance(X[1...m-1],Y[1...n-1])
15:         memo[m,n] = min(delxi, insyj, sub)
16:     end if
17:     return memo[m,n]
18: end function
```

An example

Edit distance for this to ship:

		s	h	i	p
	-1,	-1,	-1,	-1,	-1,
t	-1,	1, $r \rightarrow$ts	2, $i \rightarrow$s $r \rightarrow$th	3, $i \rightarrow$sh $r \rightarrow$ti	4, $i \rightarrow$shi $r \rightarrow$tp
h	-1,	2, $d \rightarrow$t $r \rightarrow$hs	1, $r \rightarrow$ts	2, $r \rightarrow$ts $i \rightarrow$i	3, $r \rightarrow$ts $i \rightarrow$i $i \rightarrow$p
i	-1,	3, $d \rightarrow$th $r \rightarrow$is	2, $r \rightarrow$ts $d \rightarrow$i	1, $r \rightarrow$ts	2, $r \rightarrow$ts $i \rightarrow$p
s	-1,	3, $d \rightarrow$thi	3, $r \rightarrow$ts $d \rightarrow$i $d \rightarrow$s	2, $r \rightarrow$ts $d \rightarrow$s	2, $r \rightarrow$ts $r \rightarrow$sp

For morena to morning:

$i \rightarrow$n $r \rightarrow$ei $r \rightarrow$ag

Elements of Dynamic Programming

Problems that are amenable to solution by dynamic programming have the following properties:

1. optimal substructure
2. overlapping subproblems.

Optimal substructure

- A problem exhibits optimal substructure if an optimal solution to a problem includes within it, an optimal solution to a subproblem.
- We've seen this with 0-1 knapsack, longest common subsequence and edit distance (but also shortest path and Huffman coding).
- All of the DP problems look very similar:

0-1 Knapsack:

$$
\begin{aligned}
V(0, w) &= 0 \\
V(k, 0) &= 0 \\
V(k, w) &= V(k-1, w) \text{ if } w_k > w \\
V(k, w) &= \max(V(k-1, w), v_k + V(k-1, w - w_k))
\end{aligned}
$$

LCS:

$$
l_{i,j} = \begin{cases} 0, & \text{if } i = 0 \text{ or } j = 0, \\ l_{i-1,j-1} + 1, & \text{if } x_i = y_j, \\ \max(l_{i,j-1}, l_{i-1,j}), & \text{otherwise} \end{cases}
$$

Edit distance:

$$
d_{ij} = \begin{cases} d_{i-1,j-1}, & \text{if } X[i] = Y[j] \\ \min \begin{cases} d_{i-1,j} + 1, & \text{delete } X[i] \\ d_{i,j-1} + 1, & \text{insert } Y[j] \\ d_{i-1,j-1} + 1, & \text{substitute } Y[j] \text{ for } X[i] \end{cases}, & \text{if } X[i] \neq Y[j] \end{cases}
$$

Subproblems are overlapping but independent

- Overlapping means that a naive recursive algorithm will solve the same problem multiple times.
- Independent means that subproblems don't share resources
- For example, if different subproblems needed to write to the same memory location. Or different subproblems involved accessing a shared physical resource (say a single robot arm).

What about greedy algorithms then?

First, let's consider the difference between greedy algorithms and DP algorithms:

- Both: make a choice at each step
- DP: the choice depends on the solution to (more than one) subproblem
- Greedy: the choice can be made before solving subproblems
- Therefore, DP is inherently bottom-up (solve subproblems, make a choice), whereas greedy is top-down (make a choice, solve subproblems)[1].
- For example, in Dijkstra's algorithm, we choose which vertex to expand (the choice), then solve subproblems (the shortest path from that vertex).
- Divide-and-conquer algorithms are also similar to greedy algorithms. For example, in quicksort, we make a choice (choose the pivot), then solve subproblems (sort the sublists).

[1] Well, not all greedy algorithms are top-down. E.g. Huffman's algorithm is bottom-up

When greedy algorithms are optimal

- Optimization problems for which greedy solutions are optimal, have some different properties compared to problems for which DP solutions are optimal.
- These include:
  1. There is only one subproblem or only one subproblem is non-empty
  2. We don't need to solve the subproblems before making a choice.
- Fractional knapsack fits both of these criteria.
- Shortest path fits the second criteria.

Next time ...

- There are many problems that cannot (yet) be efficiently solved
- Many of them are in a class of problems called NP
- All of the problems we've looked at so far are in the class P
- We'll look at these two classes next.