Advanced Algorithm Design and Analysis (Lecture 6)

SW5 fall 2004

Simonas Šaltenis
E1-215b
simas@cs.aau.dk

Dynamic Programming

- Goals of the lecture:
  - *to understand the **principles** of dynamic programming;*
  - *use the examples of computing **optimal binary search trees**, **approximate pattern matching**, and **coin changing** to see how the principles work;*
  - *to be able to **apply** the dynamic programming algorithm design technique.*

Coin changing

- Problem: *Change amount* A *into as few coins as possible, when we have* n *coin denominations:*

  $$denom[1] > denom[2] > \ldots > denom[n] = 1$$

- For example:
  - $A = 12$, $denom = [10, 5, 1]$

- Greedy algorithm works fine (for this example)
  - Prove greedy choice property

- What if $A = 12$, $denom = [10, 6, 1]$?

Dynamic programming

- *Dynamic programming*:
  - A powerful technique to solve *optimization problems*
- Structure:
  - To arrive at an optimal solution *a number of choices* are made
  - Each *choice generates* a number of *sub-problems*
  - Which choice to make is decided by looking at all possible choices and the solutions to sub-problems that each choice generates
    - Compare this with a greedy choice.
  - The solution to a specific sub-problem is used many times in the algorithm

Questions to think about

- Construction:
  - *What are the sub-problems? Which parameters define each sub-problem?*
  - *Which choices have to be considered in each step of the algorithm?*
  - *In which order do we have to solve sub-problems?*
  - *How are the trivial sub-problems solved?*
- Analysis:
  - *How many different sub-problems are there in total?*
  - *How many choices have to be considered in each step of the algorithm?*

Edit Distance

- Problem definition:
  - Two strings: $s[0..m\text{-}1]$, and $t[0..n\text{-}1]$
  - Find *edit distance* $dist(s,t)$– the smallest number of edit operations that turns s into t
  - Edit operations:
    - **Replace** one letter with another
    - **Delete** one letter
    - **Insert** one letter
  - Example:

`ghost`	delete `g`
`host`	insert `u`
`houst`	replace `t` by `e`
`house`	

Sub-problmes

- What are the sub-problems?
  - *Goal* 1: To have as few sub-problems as possible
  - *Goal* 2: Solution to the sub-problem should be possible by combining solutions to smaller sub-problems.
- Sub-problem:
  - $d_{i,j} = dist(s[0..i], t[0..j])$
  - Then $dist(s, t) = d_{m-1,n-1}$

Making a choice

- *How can we solve a sub-problem by looking at solutions of smaller sub-problems to make a choice?*
  - Let's look at the last symbol: $s[i]$ and $t[j]$. Do whatever is cheaper:
    - If $s[i] = t[j]$, then turn $s[0..i-1]$ to $t[0..j-1]$, else **replace** $s[i]$ by $t[j]$ and turn $s[0..i-1]$ to $t[0..j-1]$
    - **Delete** $s[i]$ and turn $s[0..i-1]$ to $t[0..j]$
    - **Insert** insert $t[j]$ at the end of $s[0..i-1]$ and turn $s[0..i]$ to $t[0..j-1]$

Recurrence

$$d_{i,j} = \min \begin{cases} d_{i-1,j-1} + \begin{cases} 0 & \text{if } s[i] = t[j] \\ 1 & \text{else} \end{cases} \\ d_{i-1,j} + 1 \\ d_{i,j-1} + 1 \end{cases}$$

- *In which order do we have to solve sub-problems?*
- *How do we solve trivial sub-problems?*
  - To turn empty string to $t[0..j]$, do j+1 **insert**s
  - To turn $s[0..i]$ to empty string, do i+1 **delete**s

Algorithm

```
EditDistance(s[0..m-1], t[0..n-1])
for i =-1 to m-1 do dist[i,-1] = i+1
for j = 0 to n-1 do dist[-1,j] = j+1
for i =0 to m-1 do
   for j = 0 to n-1 do
      if s[i] = t[j] then
         dist[i,j] = min(dist[i-1,j-1], dist[i-1,j]+1,
                         dist[i,j-1]+1)
      else
         dist[i,j] = 1 + min(dist[i-1,j-1], dist[i-1,j],
                         dist[i,j-1])
return dist[m-1,n-1]
```

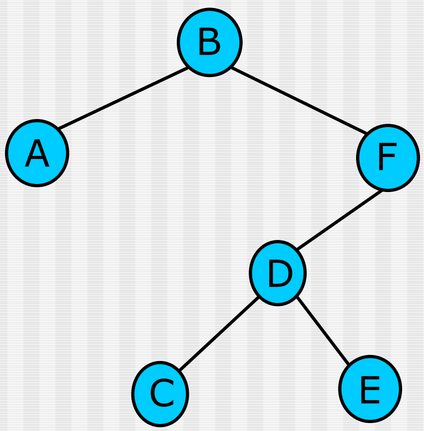
- What is the running time of this algorithm?

Approximate Text Searching

- *Given $p[0..m\text{-}1]$, find a sub-string of t ($w = t[i,j]$), such that $dist(p, w)$ is minimal.*
  - Brute-force: compute edit distance between *p* and all possible sub-strings of *t*. Running time?
  - What are the sub-problems?
  - $ad_{i,j} = \min\{dist(p[0..i], t[l..j]) \mid 0 \le l \le j+1\}$
  - The same recurrence as for $d_{i,j}$!
  - The edit distance from *p* to the best match then is the minimum of $ad_{m-1,0}, ad_{m-1,1}, \ldots , ad_{m-1,n-1}$
  - Trivial problems are solved different:
    - Think how.

Optimal BST

- Static database ⇒ the goal is to optimize searches
  - Let's assume all searches are successful



Node (k_i)	Depth	Probability (p_i)	Contribution
A	1	0.1	0.2
B	0	0.2	0.2
C	3	0.16	0.64
D	2	0.12	0.36
E	3	0.18	0.72
F	1	0.24	0.48
Total:		1.00	2.6

$$\text{Expected cost of search in T} = \sum_{i=1}^{n} (depth_T(k_i) + 1) \cdot p_i = 1 + \sum_{i=1}^{n} depth_T(k_i) \cdot p_i$$

Sub-problems

- Input: keys $k_1, k_2, \ldots, k_n$
- Sub-problem options:
  - $k_1, k_2, \ldots, k_j$
  - $k_i, k_{i+1}, \ldots, k_n$
- Natural choice: pick as a root k_r ($1 \le r \le n$)
  - Generates sub-problems: $k_i, k_{i+1}, \ldots, k_j$
  - Lets denote the expected search cost $e[i,j]$.
  - If k_r is root, then

$$e(i,j) = p_r + \left(e[i,r-1] + w(i,r-1)\right) + \left(e[r+1,j] + w(r+1,j)\right),$$

where $w(i,j) = \sum_{l=1}^{j} p_l$

Solving sub-problems

Observe that

$$w(i,j) = w[i,r-1] + p_r + w[r+1,j].$$

Thus,

$$e(i,j) = e[i,r-1] + e[r+1,j] + w(i,j)$$

- *How do I solve the trivial problem?*

$$e(i,j) = \begin{cases} p_i & \text{if } i = j \\ \min\limits_{i \le r \le j}\{e[i,r-1] + e[r+1,j] + w(i,j)\} & \text{if } i < j \end{cases}$$

- *In which order do I have to solve my problems?*

Finishing up

- I can compute $w(i,j)$ using $w(i,j\text{-}1)$
  - $w(i,j) = w(i,j\text{-}1) + p_j$
  - An array $w[i,j]$ is filled in parallel with $e[i,j]$ array
- Need one more array to note which root k_r gave the best solution to (i, j)-sub-problem
- *What is the running time*?

Elements of Dynamic Programming

- Dynamic programming is used for optimization problems
  - A number of choices have to be made to arrive at an optimal solution
  - At each step, consider all possible choices and solutions to sub-problems induced by these choices (compare to greedy algorithms)
  - The order of solving of the sub-problems is important – from smaller to larger
- Usually a table of sub-problem solutions is used

Elements of Dynamic Programming

- To be sure that the algorithm finds an optimal solution, the *optimal sub-structure* property has to hold
  - the simple “cut-and-paste” argument usually works,
  - but not always! Longest simple path example – no optimal sub-structure!

Coin Changing: Sub-problems

- A =12, *denom* = [10, 6, 1]?

- *What could be the sub-problems? Described by which parameters?*
- *How do we solve sub-problems?*

$$c(i,j) = \begin{cases} c(i+1,j) & \text{if } denom[i] > j \\ \min\{c(i+1,j), 1 + c(i, j - denom[i])\} & \text{if } denom[i] \le j \end{cases}$$

- *How do we solve the trivial sub-problems?*
- *In which order do I have to solve sub-problems?*