Lectures 12 and 13
Dynamic programming: weighted interval scheduling

COMP 523: *Advanced Algorithmic Techniques*

Lecturer: *Dariusz Kowalski*

Overview

Last week:

- Graph algorithm: BFS and DFS, testing graph properties based on searching, topological sorting

This week:

- Dynamic programming
- Weighted interval scheduling
- Sequence alignment

Dynamic Programming paradigm

Dynamic Programming (DP):

- Decompose the problem into series of subproblems
- Build up correct solutions to larger and larger subproblems

Similar to:

- Recursive programming vs. DP: in DP subproblems may strongly overlap
- Exhaustive search vs. DP: in DP we try to find redundancies and reduce the space for searching

(Weighted) Interval scheduling

(Weighted) Interval scheduling:

Input: set of intervals (with weights) on the line, represented by pairs of points - ends of intervals

Output: finding the largest (maximum sum of weights) set of intervals such that none two of them overlap

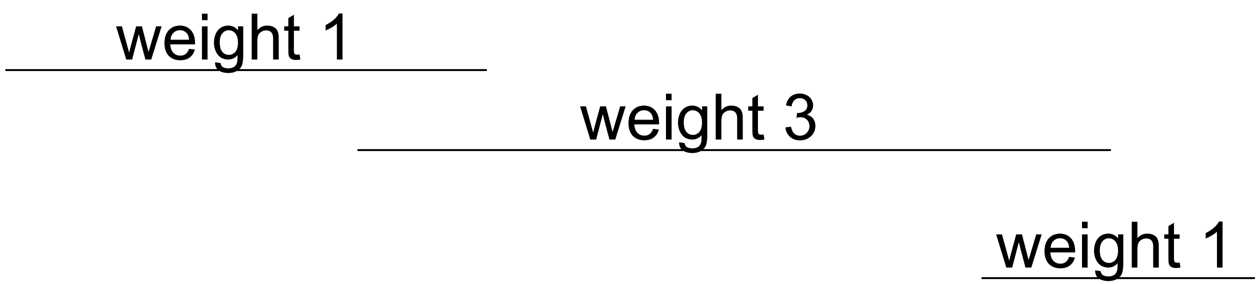
Greedy algorithm doesn't work for weighted case!

Example

Greedy algorithm:

- Repeatedly select the interval which ends first (but still not overlapping the already chosen intervals)

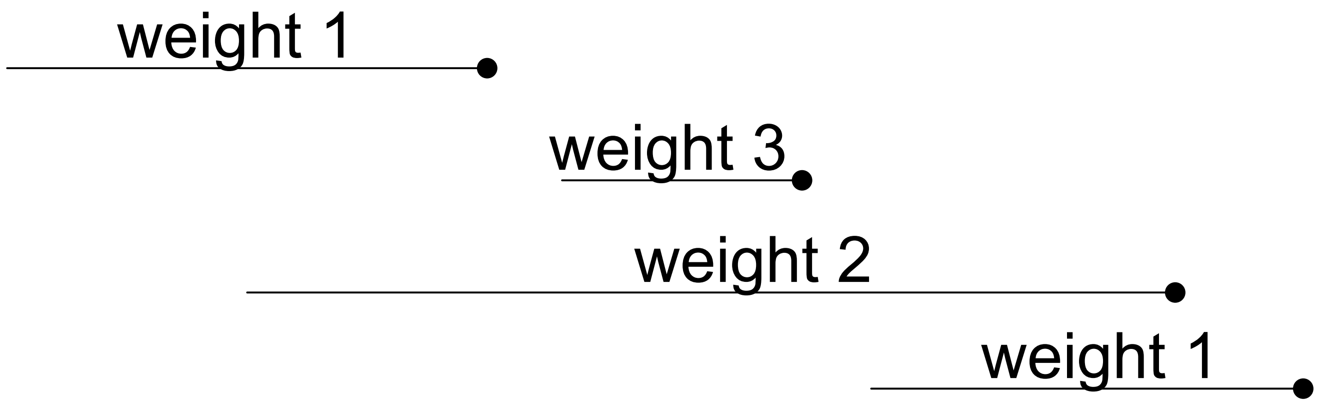
Exact solution of unweighted case.



Greedy algorithm gives total weight 2 instead of optimal 3

Basic structure and definition

- Sort the intervals according to their right ends
- Define function p as follows:
  - $p(1) = 0$
  - $p(i)$ is the number of intervals which finish before i^{th} interval starts



$p(1)=0$

$p(2)=1$

$p(3)=0$

$p(4)=2$

Basic property

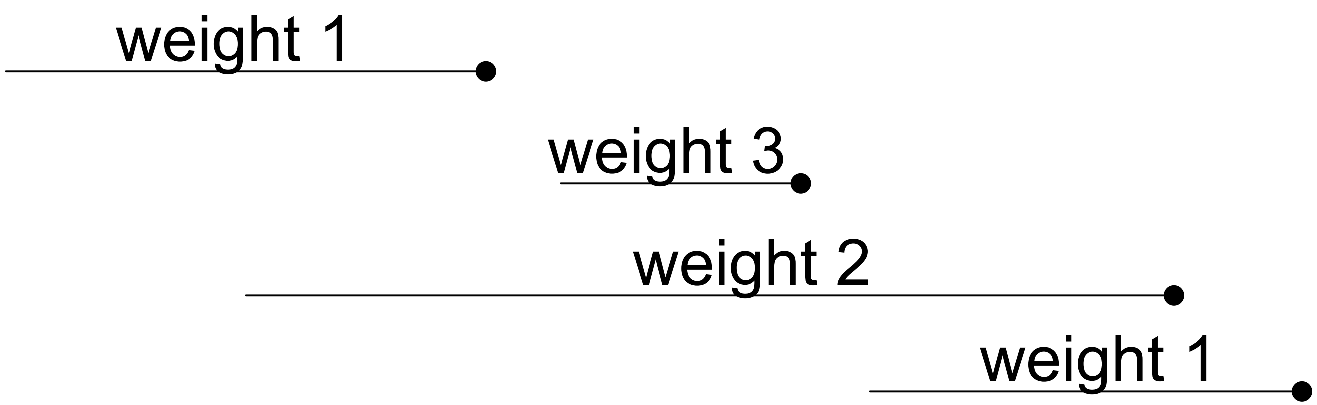
- Let w_j be the weight of j^{th} interval
- Optimal solution for the set of first j intervals satisfies

$$OPT(j) = \max\{\, w_j + OPT(p(j))\, ,\, OPT(j\text{-}1) \,\}$$

Proof:

If j^{th} interval is in the optimal solution **O** then the other intervals in **O** are among intervals $1,\ldots,p(j)$.

Otherwise search for solution among first j-1 intervals.



$p(1)=0$

$p(2)=1$

$p(3)=0$

$p(4)=2$

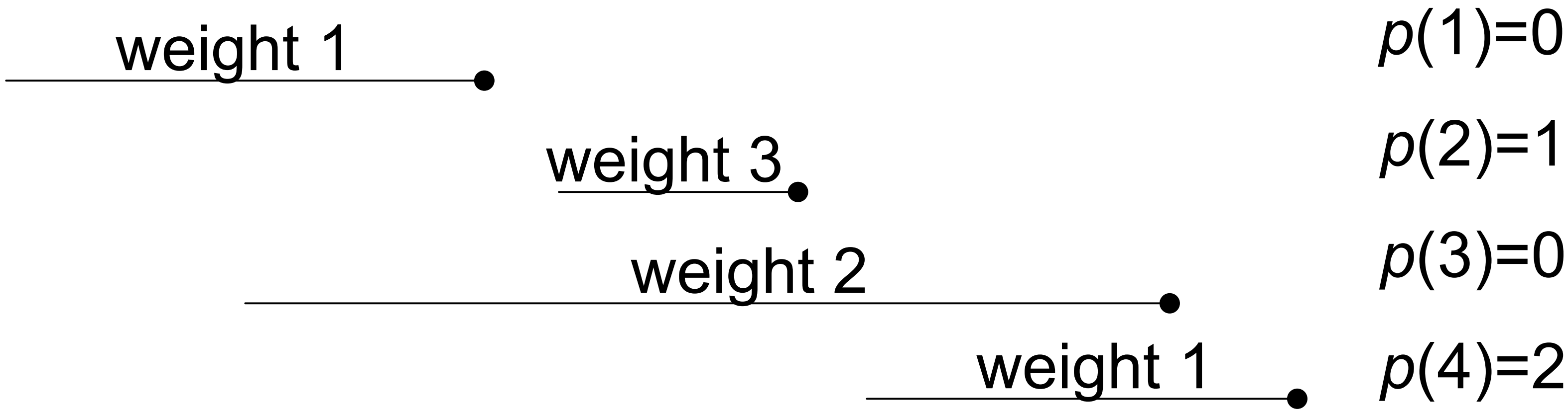
Sketch of the algorithm

- Additional array M[0…n] initialized by 0,p(1),…,p(n)

(intuitively M[j] stores optimal solution OPT(j))

Algorithm

- For j = 1,…,n do
  - Read $p(j)$ = M[j]
  - Set M[j] := max{ w_j + M[$p(j)$] , M[j-1] }

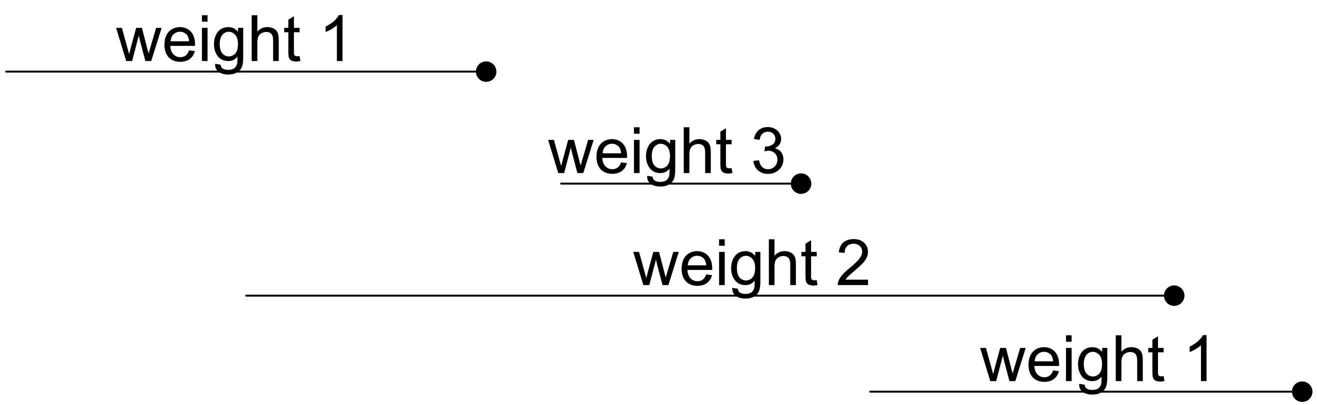


Complexity of solution

Time: O(n log n)

- Sorting: O(n log n)
- Initialization of M[0…n] by 0,p(1),…,p(n): O(n log n)
- Algorithm: n operations, each takes constant time, total O(n)

Memory: O(n) - additional array M



$p(1)=0$

$p(2)=1$

$p(3)=0$

$p(4)=2$

Sequence alignment problem

Popular problem from word processing and computational biology

- Input: two words $X = x_1 x_2 \ldots x_n$ and $Y = y_1 y_2 \ldots y_m$
- Output: largest alignment

Alignment $\boldsymbol{A}$: set of pairs $(i_1, j_1), \ldots, (i_k, j_k)$ such that

- If (i,j) in $\boldsymbol{A}$ then $x_i = y_j$
- If (i,j) is before (i', j') in $\boldsymbol{A}$ then $i < i'$ and $j < j'$ (no crossing matches)

Example

- Input: X = *c t t t c t c c*   Y = *t c t t c c*

Alignment ***A***:

```
X = c t t t c t   c c
      |     | |   | |
Y =   t     c t t c c
```

Another largest alignment ***A***:

```
X =   c t t t c t c c
      | | |   |   |
Y = t c t t   c   c
```

Finding the size of max alignment

Optimal alignment $OPT(i,j)$ for prefixes of X and Y of lengths i and j respectively:

$$OPT(i,j) = \max\{ \alpha_{ij} + OPT(i-1,j-1) , OPT(i,j-1) , OPT(i-1,j) \}$$

where α_{ij} equals 1 if $x_i = y_j$, otherwise is equal to $-\infty$

Proof:

If $x_i = y_j$ in the optimal solution **O** then the optimal alignment contains one match (x_i , y_j) and the optimal solution for prefixes of length i-1 and j-1 respectively.

Otherwise at most one end is matched. It follows that either $x_1x_2\ldots x_{i-1}$ is matched only with letters from $y_1y_2\ldots y_m$ or $y_1y_2\ldots y_{j-1}$ is matched only with letters from $x_1x_2\ldots x_n$. Hence the optimal solution is either the same as for $OPT(i-1,j)$ or for $OPT(i,j-1)$.

Algorithm finding max alignment

- Initialize matrix $M[0..n,0..m]$ into zeros

Algorithm

- For $i = 1,\ldots,n$ do
  - For $j = 1,\ldots,m$ do
    - Compute α_{ij}
    - Set $M[i,j] :=$
      $\max\{\ \alpha_{ij} + M[i\text{-}1,j\text{-}1]\ ,\ M[i,j\text{-}1]\ ,\ M[i\text{-}1,j]\ \}$

Complexity

Time: $O(nm)$

- Initialization of matrix $M[0..n, 0..m]$: $O(nm)$
- Algorithm: $O(nm)$

Memory: $O(nm)$

Reconstruction of optimal alignment

Input: matrix $M[0..n, 0..m]$ containing OPT values

Algorithm

- Set $i = n$, $j = m$
- While $i, j > 0$ do
    - Compute α_{ij}
    - If $M[i,j] = \alpha_{ij} + M[i-1, j-1]$ then match x_i and y_j and set $i = i - 1$, $j = j - 1$; else
    - If $M[i,j] = M[i, j-1]$ then set $j = j - 1$ (skip letter y_j), else
    - If $M[i,j] = M[i-1, j]$ then set $i = i - 1$ (skip letter x_i)

Distance between words

Generalization of alignment problem

- Input:
  - two words $X = x_1x_2\ldots x_n$ and $Y = y_1y_2\ldots y_m$
  - mismatch costs α_{pq}, for every pair of letters p and q
  - gap penalty δ
- Output: (smallest) distance between words X and Y

Example

- Input: X = $c\ t\ t\ t\ c\ t\ c\ c$ Y = $t\ c\ t\ t\ c\ c$

Alignment $\boldsymbol{A}$: (4 gaps, 1 mismatch of cost α_{ct})

```
X = c t t t c t c c
      |     | | ^ |
Y =   t     c t t c c
```

Largest alignment $\boldsymbol{A}$: (4 gaps)

```
X =   c t t t c t c c
      | | |   |   |
Y = t c t t   c   c
```

Finding the distance between words

Optimal alignment $OPT(i,j)$ for prefixes of X and Y of lengths i and j respectively:

$$OPT(i,j) = \max\{ \alpha_{ij} + OPT(i-1,j-1) , \delta + OPT(i,j-1) , \delta + OPT(i-1,j) \}$$

Proof:

If x_i and y_j are (mis)matched in the optimal solution **O** then the optimal alignment contains one (mis)match (x_i , y_j) of cost α_{ij} and the optimal solution for prefixes of length i-1 and j-1 respectively.

Otherwise at most one end is (mis)matched. It follows that either $x_1x_2\ldots x_{i-1}$ is (mis)matched only with letters from $y_1y_2\ldots y_m$ or $y_1y_2\ldots y_{j-1}$ is (mis)matched only with letters from $x_1x_2\ldots x_n$. Hence the optimal solution is either the same as counted for $OPT(i-1,j)$ or for $OPT(i,j-1)$, plus the penalty gap δ.

Algorithm and complexity remain the same.

Conclusions

- Dynamic programming
- Weighted interval scheduling
- Sequence alignment

Textbook and Exercises

- Chapter 6 “Dynamic Programming”
- All Interval Sorting problem