Weighted Interval Scheduling KT 6.1

Frank

October 20, 2021

1 Weighted Interval Scheduling

- ***Input***: Set of intervals $1, 2, \ldots, n$, where interval is given by $[s(i), f(i)]$.

  $v(i) \geq 0$ value of interval i

- ***Output***: Feasible subset of intervals of maximum value.

- Note our greedy algorithm problem was a special case of this where each interval has the same value.

1.1 DP Solution

Suppose S is an optimal set of intervals (How does this set extend from some optimal solution to a smaller problem?)

- Assume the intervals $1, 2, \ldots, n$ are sorted in non-decreasing finish time.

  We have 2 cases:

  1. Interval n does not belong to $S \implies S$ is optimal for $1, 2, \ldots, n-1$ by cut-and-paste argument (If there is a better solution, cut-and-paste that solution).
  2. Interval n belongs to $S \implies S = S' \cup \{n\} \implies S'$ is optimal for $1, 2, \ldots, p(n)$, where $p(n) = max\{j : f(j) \leq s(n)\}$ and $max(\emptyset) = 0$.

  And we have found the optimal substructure (that the optimal solution S is an extension of an optimal solution S' to a smaller problem, or in the first case, **is** the optimal solution to a smaller problem).

- Now lets first find the **value** of an optimal solution, then retrofit our original problem

1.2 Subproblem

- $V(i)$ is value of optimal subset of intervals $1, 2, \ldots, i$, for $0 \leq i \leq n$ and $V(0) = 0$ $(\star)$

$$V(i) = \begin{cases} max(V(i-1), V(p(i)) + v(i)) & \text{if } i > 0 \\ 0 & \text{if } i = 0 \end{cases} \qquad (\dagger)$$

Where we take $V(i-1)$ if it is case 1, and $V(p(i)) + v(i)$ if it is case 2. Either way we have a substructure.

Note that $(\dagger)$ computes $(\star)$

1.3 Algorithm ($\star$)

```
procedure WEIGHTED IS ($\star$)
    Sort intervals in non-decreasing finish time          ▷ $f(1) \leq f(2) \leq \dots$
    for ($i = 1 \to n$) do                                ▷ assign $p(i)$
        $p(i) \leftarrow max\{j : f(j) \leq s(i)\}$
    $V(0) \leftarrow 0$
    for ($i = 1 \to n$) do
        if ($V(i-1) \geq V(p(i)) + v(i)$) then            ▷ No advantage to taking interval $i$
            $V(i) \leftarrow V(i-1)$
        else                                              ▷ Better to take $i$ compared to $V(i-1)$
            $V(i) \leftarrow V(p(i)) + v(i)$
    return $V(n)$
```

Note in line 7 we might even hurt our value by taking i (Cannot take another better interval due to overlapping caused by i). Thus we have a $\geq$ sign.

1.4 Running Time

$O(n \log n)$ for line 2 due to sorting + $O(n \log n)$ for line 3-4 since our intervals are sorted + $O(n)$ due to second for loop.

$\implies O(n \log n)$

1.5 DP Checklist

1. Subproblems to define ($\star$)
2. Recursive formula ($\dagger$) to compute ($\star$)
3. Solving original problem from subproblem (We have $V(n)$ so just work backwards).

1.6 Algorithm

```
procedure WEIGHTED IS - $O(n^2)$ SPACE
    Sort intervals in non-decreasing finish time          ▷ $f(1) \leq f(2) \leq \dots$
    for ($i = 1 \to n$) do                                ▷ assign $p(i)$
        $p(i) \leftarrow max\{j : f(j) \leq s(i)\}$
        $S(i) \leftarrow \emptyset$                       ▷ $S$ is a sequence of sets
    $V(0) \leftarrow 0$
    $S(0) \leftarrow \emptyset$
    for ($i = 1 \to n$) do
        if ($V(i-1) \geq V(p(i)) + v(i)$) then            ▷ No advantage to taking interval $i$
            $V(i) \leftarrow V(i-1)$
            $S(i) \leftarrow S(i-1)$
        else                                              ▷ Better to take $i$ compared to $V(i-1)$
            $V(i) \leftarrow V(p(i)) + v(i)$
            $S(i) \leftarrow S(p(i)) \cup \{i\}$
    return $S(n)$
```

```
procedure WEIGHTED IS - $O(n)$ SPACE
    Sort intervals in non-decreasing finish time    ▷ $f(1) \leq f(2) \leq \dots$
    for ($i = 1 \rightarrow n$) do    ▷ assign $p(i)$
        $p(i) \leftarrow max\{j : f(j) \leq s(i)\}$
    $V(0) \leftarrow 0$
    for ($i = 1 \rightarrow n$) do
        if ($V(i-1) \geq V(p(i)) + v(i)$) then    ▷ No advantage to taking interval $i$
            $V(i) \leftarrow V(i-1)$
        else    ▷ Better to take $i$ compared to $V(i-1)$
            $V(i) \leftarrow V(p(i)) + v(i)$
    $S \leftarrow \emptyset$    ▷ Solution set
    $i \leftarrow n$
    while ($i \neq 0$) do
        if ($V(i) = V(i-1)$) then    ▷ We did not take $i$
            $i \leftarrow i - 1$
        else    ▷ We took $i$ so go back to $p(i)$
            $S \leftarrow S \cup \{i\}$
            $i \leftarrow p(i)$
    return $S$
```

2 Optimal Sequence Alignment Problem (Edit Distance Problem)

2.1 Introduction to Edits

Given 2 strings or sequences $x[1..m]$ and $y[1..n]$, how similar are they?

- An application would be a spell checker. Another would be finding similarity between genes.

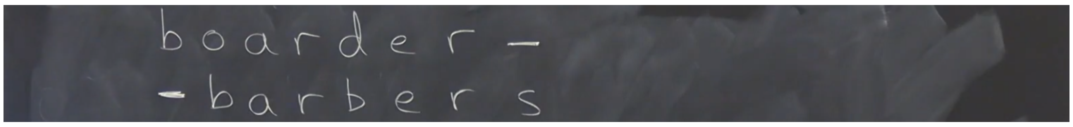
  Gene = string over $\{A, T, G, C\}$

  Edit Distance between x and y: Minimum # of single character changes (edits) to transform x to y.

  Edit: insert/delete/mutate a character

- Example: boarder vs. barbers

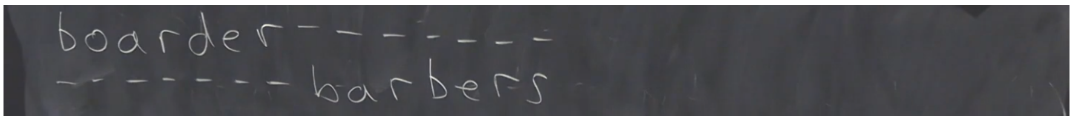
  An attempt would be an **alignment**



We can remove "b" from the first column, change "o" to "b" in the second column, change "d" to "b" in the fifth column, and finally add an "s" in the last column.

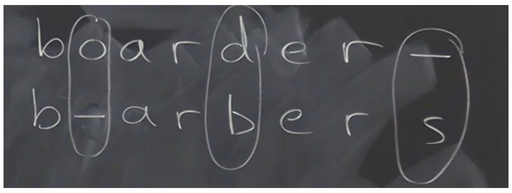
Then we have 4 edits for this alignment

But we can have really bad alignments like



In this case we will have 14 edits.

The best alignment will be



Which has 3 edits (optimal-Edit Distance)

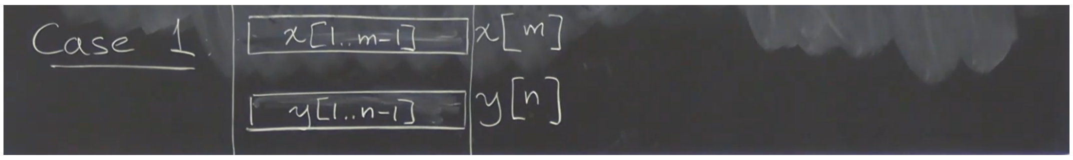
2.2 Edit Distance Problem

- ***Input:*** Strings $x[1..m]$ and $y[1..n]$
- ***Output:*** Edit distance between x and y (or find optimal alignment to gives us that edit distance)

2.3 DP Solution

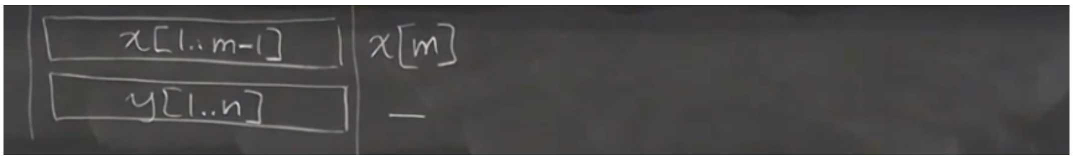
Suppose we are given optimal alignment, consider 3 cases:

1. $x[m]$ aligned with $y[n]$ to form optimal alignment



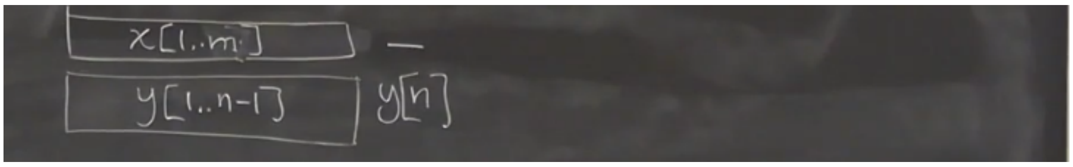
then $x[1 \ldots m-1]$ and $y[1 \ldots n-1]$ are optimally aligned for a smaller problem by cut-and-paste argument.

2. $x[m]$ aligned with a dash (delete $x[m]$)



Similarly, $x[1 \dots m-1]$ and $y[1 \dots n]$ are optimally aligned for a smaller problem by cut-and-paste argument.

3. $y[n]$ aligned with a dash (delete $y[n]$)



Similarly, $x[1 \dots m]$ and $y[1 \dots n-1]$ are optimally aligned for a smaller problem by cut-and-paste argument.

- Then define $ED(m,n)$ is the edit distance between $x[1..m]$ and $y[1..n]$, then for each cases:

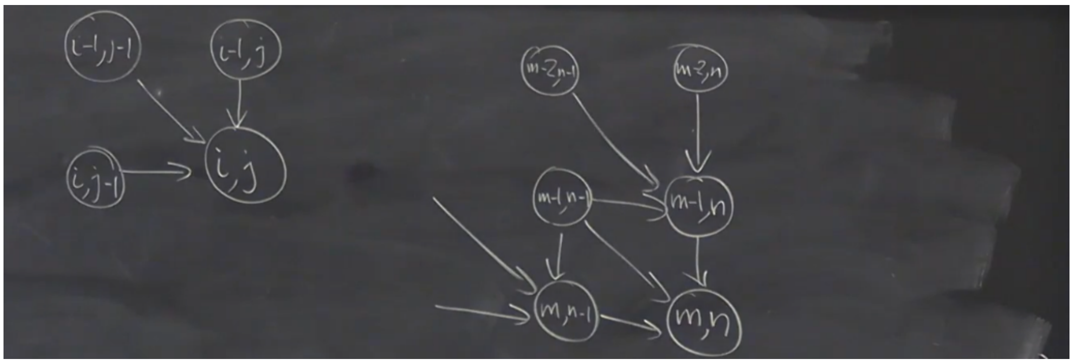
  1. $ED(m,n) = ED(m-1,n-1) + diff(m,n)$, where

  $$diff(m,n) = \begin{cases} 0 & x[m] = y[n] \\ 1 & \text{otherwise} \end{cases}$$

  2. $ED(m,n) = ED(m-1,n) + 1$, since we are guaranteed to edit once
  3. $ED(m,n) = ED(m,n-1) + 1$

  Now which case gives us the optimal alignment? (We want to minimize ED, so should choose the minimum of the 3)

- Consider the following recursive structure of dependency



To solve a problem, we need to have previously solved 3 smaller problems

- Note that

  $$ED(0, j) = j$$
  $$ED(i, 0) = i$$

  Which are the base cases of our recursion (We can stop the recursion once we have a 0 as input).

2.4 DP Solution Checklist

1. Subproblems to solve:

   ED(i, j) = edit distance of $x[1..i]$ and $y[1..j]$, $0 \leq i \leq m$, $0 \leq j \leq n$ ($\star$)

2. Recursive Formula

   $$ED(i, j) = \begin{cases} j & i = 0 \\ i & j = 0 \\ min[ED(i-1, j-1) + diff(i, j), ED(i-1, j) + 1, & \\ ED(i, j-1) + 1] & \text{o/w} \end{cases} \quad (\dagger)$$

3. Now solve the problem with our subproblem (which is the problem in this case).

2.5 Algorithm

```
1: procedure EDIT-DIST($x[1..m]$, $y[1..n]$)
2:     for ($i = 0 \to m$) do                                  ▷ set base case
3:         $ED(i, 0) \leftarrow i$
4:     for ($j = 1 \to n$) do                                  ▷ set base case
5:         $ED(0, j) \leftarrow j$
6:     for ($i = 1 \to m$) do
7:         for ($j = 1 \to n$) do
8:             $ED(i, j) \leftarrow (\dagger)$     ▷ assign edit distance from the bottom of the recursion
9:     return $ED(m, n)$
```

Note we can switch the 2 for-loops to compute ED column by column. Right now we are computing Row by row. Either way, we must respect our dependency diagram.

2.6 Observations about DP

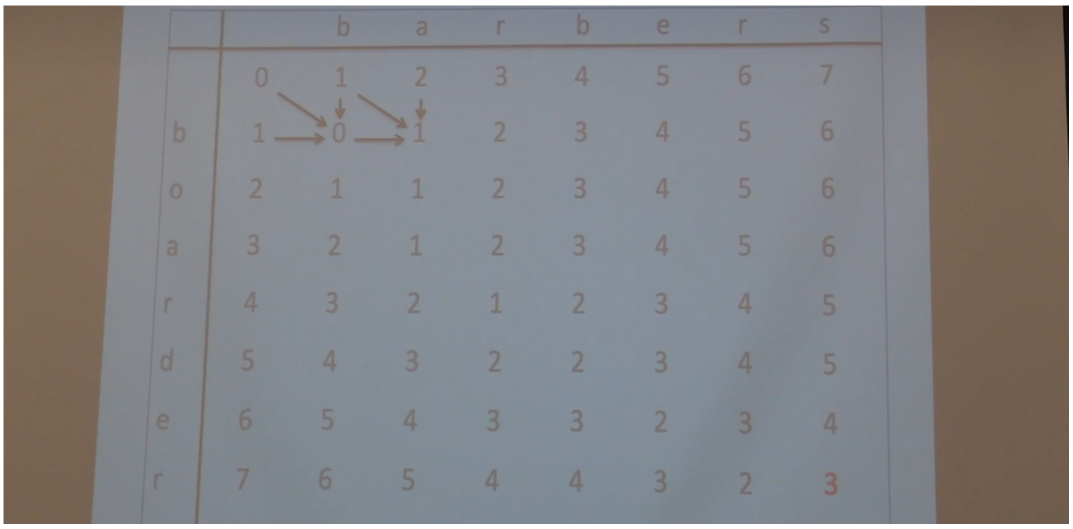
At first, we start from a optimal solution and investigate what is this optimal solution extended from. We usually have different cases for how the optimal solution at any stage is extended from a optimal solution of a substructure or smaller problem. After we figure the different cases of optimal solutions for a smaller problem (substructure), then we have a Optimal Substructure. We can construct a sub problem with this optimal substructure that can be used to solve the original problem, where we will define a recursive formula to compute the sub problem (whose correctness will be proven using our Optimal substructure), but at this point we already have the DP algorithm from this formula. For the algorithm, Note that we will now reverse our strategy (recurse from bottom-up with our formula) and construct a substructure that will be intially empty, but we will gradually extend it into the optimal solution using the cases that we defined and our formula to reach a solution for our sub problem. At this point, we can trivially solve the original problem.

2.7 Running Time

Trivially $O(mn)$

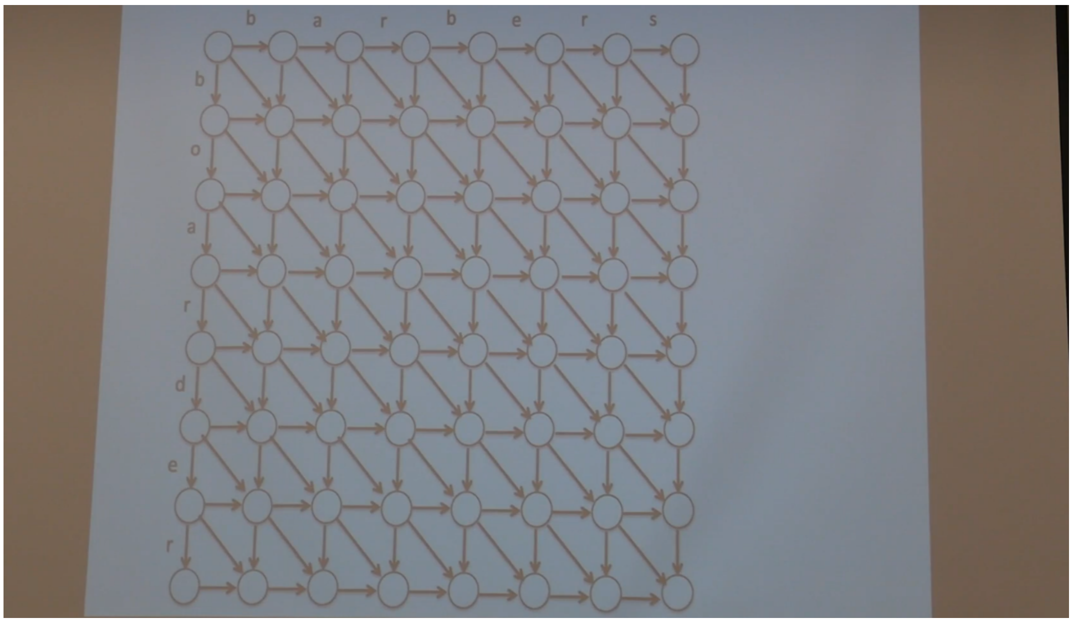
2.8 Example of the algorithm

Following is a visualization of the algorithm. Deletion corresponds to going down, insertion corresponds to going right, mutation or match corresponds to the diagonal edges (cost 0 for match, cost 1 for mutation). Look for a minimum path from upper left to bottom right.



		b	a	r	b	e	r	s
	0	1	2	3	4	5	6	7
b	1	0	1	2	3	4	5	6
o	2	1	1	2	3	4	5	6
a	3	2	1	2	3	4	5	6
r	4	3	2	1	2	3	4	5
d	5	4	3	2	2	3	4	5
e	6	5	4	3	3	2	3	4
r	7	6	5	4	4	3	2	3

Think of DP algorithm as a directed acylic graph with dependencies



2.9 Further investigation

When typing on a keyboard, switching from:

$$cost(A \to S) \ll cost(A \to P)$$

Also, with genes, certain kinds of mutations are more likely than others

- So lets define a cost function

$$\begin{aligned} &c(a, b) \text{ where } a, b \in \Gamma \cup \{-\} \\ &c(a, b) = \text{cost of an edit to change from a to b} \end{aligned}$$

2.10 Algorithm - With cost

Recursive Formula

$$ED(i, j) = \begin{cases} j & i = 0 \\ i & j = 0 \\ min[ED(i-1, j-1) + c(x[i], y[j]), & \\ ED(i-1, j) + c(x[i], -), ED(i, j-1) + c(-, y[j])] & \text{o/w} \end{cases} \qquad (\dagger)$$

2.11 Homework question

- Return an optimal alignment instead of ED.
- Longest Common Subsequence
  - Given

$$\begin{aligned} A &= a_1, a_2, \ldots, a_m \\ B &= b_1, b_2, \ldots, b_n \end{aligned}$$

  - Solve for C : subsequence of both A & B and for any subsequence C' of both A and B, $|C| \geq |C'|$
- Imagine a game with a sequence of cards 5, 3, -7, 9, -10, 9, 6. Players A and B alternate, starting with A. You can choose to take either the left-most card or the right-most card. Play until you run out of cards. The player with the most points win. Assume that both players will play optimally (choose the highest number), figure out what the best move to make is for player A or player B (Solve maximum payload for player A and B first, then solve the sequence of moves).