

Unit 6.3 Dynamic Programming III

Algorithms

EE3980

May 19, 2020

String Editing Problem

- Given two strings $X = "x_1 x_2 \cdots x_n"$ and $Y = "y_1 y_2 \cdots y_m"$, where x_i , $1 \leq i \leq n$, and y_j , $1 \leq j \leq m$, are members of a finite set of symbols known as the **alphabet**.
- The **string editing** problem is to transform X into Y using the following editing operations with corresponding cost and to find the sequence of operations that minimizes the total cost.
 - Delete** the symbol x_i from X with cost $D(x_i)$,
 - Insert** the symbol y_j to Y with cost $I(y_j)$,
 - Change** the symbol x_i of X into y_j with cost $C(x_i, y_j)$.
 - Note that keep x_i to become y_j has no cost.
- Example, $X = "elate"$ and $Y = "later"$. Total cost to transform X into Y is $D(e) + I(r)$.

Step	X	Y	Cost
1	<i>elate</i>		$D(e)$
2	<i>elate</i>	<i>l</i>	0
3	<i>elate</i>	<i>la</i>	0
4	<i>elate</i>	<i>lat</i>	0
5	<i>elate</i>	<i>late</i>	0
6	<i>elate</i>	<i>later</i>	$I(r)$
$D(e) + I(r)$			

Algorithm 6.3.1. Wagner Fischer Algorithm

```

// Transform  $X$  into  $Y$  with minimum cost using matrix  $M$ .
// Input: int  $n, m$ , strings  $X, Y$ , cost  $D, I, C$ 
// Output: min cost matrix  $M$ .
1 Algorithm WagnerFischer( $n, m, X, Y, D, I, C, M$ )
2 {
3      $M[0, 0] := 0$ ;
4     for  $i := 1$  to  $n$  do  $M[i, 0] := M[i - 1, 0] + D(X[i])$ ;
5     for  $j := 1$  to  $m$  do  $M[0, j] := M[0, j - 1] + I(Y[j])$ ;
6     for  $i := 1$  to  $n$  do {
7         for  $j := 1$  to  $m$  do {
8             if ( $X[i] = Y[j]$ ) then  $m_1 := M[i - 1, j - 1]$ ;
9             else  $m_1 := M[i - 1, j - 1] + C(X[i], Y[j])$ ;
10             $m_2 := M[i - 1, j] + D(X[i])$ ;
11             $m_3 := M[i, j - 1] + I(Y[j])$ ;
12             $M[i, j] := \min(m_1, m_2, m_3)$ ;
13        }
14    } // When done,  $M[n, m]$  contains the minimum cost of the transformation
15 }

```

String Editing — Example

- Example. Given $X = \text{"elate"}'$, $Y = \text{"later"}'$ and the cost functions $D(x) = 1$, $I(y) = 1$, $C(x, y) = 2$, $x, y \in \{A, \dots, Z, a, \dots, z\}$, $x \neq y$.

- Thus the transformation sequence is

		<i>l</i>	<i>a</i>	<i>t</i>	<i>e</i>	<i>r</i>
<i>e</i>	0	1	2	3	4	5
<i>l</i>	1	2	3	4	3	4
<i>a</i>	2	1	2	3	4	5
<i>t</i>	3	2	1	2	3	4
<i>e</i>	4	3	2	1	2	3
<i>r</i>	5	4	3	2	1	2

Step	operation	<i>Y</i>
1	Delete <i>e</i>	
2	Keep <i>l</i>	<i>l</i>
3	Keep <i>a</i>	<i>la</i>
4	Keep <i>t</i>	<i>lat</i>
5	Keep <i>e</i>	<i>late</i>
6	Insert <i>r</i>	<i>later</i>

Matrix M of

WagnerFischer algorithm.

- And the total cost is

$$D(e) + I(r) = 2.$$

- After **WagnerFischer** algorithm, the following algorithm traces the M matrix to generate the transformation sequence.
 - Note that array T has the transformation sequence but is in reverse order.

String Editing — Transformation Trace

Algorithm 6.3.2. Trace

```
// Trace the matrix  $M[n][m]$  to find the transformation operations.
// Input: int  $n, m$ , cost  $D[], I[], C[]$  and  $M[n][m]$ 
// Output:  $T[]$  transformation.
1 Algorithm Trace( $n, m, M, D, I, C, T$ )
2 {
3      $i := n; j := m; k := 0;$ 
4     while ( $i > 0$  and  $j > 0$ ) do {
5         else if ( $i > 0$  and  $j > 0$  and ( $M[i, j] = M[i - 1, j - 1] + C(X[i], Y[j])$ )) then {
6              $T[k] := 'C'; i := i - 1; j := j - 1; k := k + 1;$  // Change  $X[i]$  to  $Y[j]$ .
7         }
8         else if ( $j = 0$  or ( $M[i, j] = M[i, j - 1] + I(Y[j])$ )) { // Add  $Y[j]$ .
9              $T[k] := 'I'; j := j - 1; k := k + 1;$ 
10        }
11        else if ( $i = 0$  or ( $M[i, j] = M[i - 1, j] + D(X[i])$ )) then { // Delete  $X[i]$ .
12             $T[k] := 'D'; i := i - 1; k := k + 1;$ 
13        }
14        else { // No changes.
15             $T[k] := '-'; i := i - 1; j := j - 1; k := k + 1;$ 
16        }
17    } // Array  $T$  has the transformation sequence but is in reverse order.
18 }
```

- In this algorithm, it is assumed that $C > I > D$.
 - If the assumption is not correct then modifications are necessary to ensure correct results.

String Editing — Complexities

- Algorithm **WagnerFischer**
 - for loop, lines 6–13, executes $n \times m$ times
 - for loops, lines 6,7, execute n and m times, separately
 - Overall time complexity $\mathcal{O}(mn)$
- Algorithm **Trace** while loop, lines 4–21, executes at most $(m + n)$ times
 - Time complexity $\mathcal{O}(m + n)$
- The longest common substring problem
 - Given two strings, X and Y , find a common substring Z such that Z has the most number of characters.
 - Example, $X = \text{"elate"}$ and $Y = \text{"later"}$ the longest common substring is $Z = \text{"late"}$. Z has 4 characters.
 - The **WagnerFischer** algorithm can be used to find the longest common substring.
 - The **Trace** algorithm needs to be modified to find and print out the common substring.

0/1 Knapsack Problem

- The **0/1 knapsack problem** is a variation of the knapsack problem.
 - Given n objects, each with profit p_i and weight w_i , $1 \leq i \leq n$, to be placed into a sack that can hold maximum of m weight. However, there is an additional constraint that each object must be placed as a whole into the sack, or not at all. That is, find x_i , $1 \leq i \leq n$, such that

$$\begin{aligned} & \text{maximize} && \sum_{i=1}^n p_i x_i, \\ & \text{subject to} && \sum_{i=1}^n w_i x_i \leq m, \\ & && \text{and } x_i = 0 \text{ or } 1, \quad 1 \leq i \leq n. \end{aligned} \tag{6.3.1}$$

- Let $f_n(m)$ be the optimal solution to n -object 0/1 knapsack problem.
- For the n 'th object it can either be placed into the sack or not, thus

$$f_n(m) = \max \left(f_{n-1}(m), f_{n-1}(m - w_n) + p_n \right). \tag{6.3.2}$$

- $f_n(m)$ must be the larger of the following two cases
- n -th object **is not** placed into the sack, $x_n = 0$,
 - In this case, $f_n(m) = f_{n-1}(m)$.
- n -th object **is** placed into the sack, $x_n = 1$,
 - In this case, $f_n(m) = f_{n-1}(m - w_n) + p_n$.

0/1 Knapsack — Recursive Algorithm

- Using Eq. (6.3.2) a recursive version of the 0/1 knapsack algorithm can be formulated.

Algorithm 6.3.3. Recursive DKP

```
// Find the solution array  $x$  for the 0/1 knapsack problem.
// Input: int  $n$ , profit  $p$ , weight  $w$ ,  $m$ 
// Output: Solution  $x$ .
1 Algorithm DKPr( $n, p, w, m, x$ )
2 {
3     if ( $n = 1$ ) then {
4         if ( $m \geq w[1]$ ) then {
5              $x[1] := 1$ ; return  $p[1]$ ;
6         }
7         else {
8              $x[1] := 0$ ; return 0;
9         }
10    }
11     $f_1 := \text{DKPr}(n - 1, p, w, m, x)$ ; // object  $n$  not placed
12    if ( $m \geq w[n]$ ) then // placing  $n$ 'th object
13         $f_2 := \text{DKPr}(n - 1, p, w, m - w[n], x) + p[n]$ ;
14    else  $f_2 := 0$ ; // no room for additional objects
15    if ( $f_1 > f_2$ ) then {
16         $x[n] := 0$ ; return  $f_1$ ;
17    }
18    else {
19         $x[n] := 1$ ; return  $f_2$ ;
20    }
21 }
```

0/1 Knapsack — Example

- Given 3 objects, $(p_1, p_2, p_3) = (1, 2, 5)$, $(w_1, w_2, w_3) = (2, 3, 4)$, and $m = 6$. Find the optimal 0/1 knapsack solution, (x_1, x_2, x_3) , $x_i = 0$ or $x_i = 1$, $1 \leq i \leq 3$, that maximizes the profit,

$$P = \sum_{i=1}^3 p_i x_i.$$

- The function `DKPr` is invoked by calling $P = \text{DKPr}(3, p, w, 6, x)$
 - And the calling sequence of the function is

```
// DKPr calling sequence
DKPr(3, p, w, 6, x)
  DKPr(2, p, w, 6, x) // object 3 not placed
    DKPr(1, p, w, 6, x) // object 2 not placed
      P := 1; x := (1, 0, 0);
    DKPr(1, p, w, 3, x) // object 2 placed
      P := 3; x := (1, 1, 0);
  DKPr(2, p, w, 2, x) // object 3 placed
    DKPr(1, p, w, 2, x) // object 2 not placed
      P := 6; x := (1, 0, 1);
    DKPr(1, p, w, -1, x) // object 2 placed
      P := -∞; x := (0, 1, 1);
Maximum profit P := 6, x := (1, 0, 1).
```

0/1 Knapsack — Complexity

- Note that function `DKPr` is invoked 7 times
 - All possible combinations of $x_i = 0$ and $x_i = 1$, $1 \leq i \leq n$ are tested for the maximum profit.
- The time complexity of `DKPr` algorithm is $\mathcal{O}(2^n)$.
- Line 11 of `DKPr` algorithm can eliminate unnecessary function calls
 - If there is no room for object n then it is not necessary to call `DKPr` further.
- The worst-case complexity of `DKPr` remains as $\mathcal{O}(2^n)$.

Algorithm 6.3.4. 0/1 Knapsack

```

// Find the solution array  $x$  for the 0/1 knapsack problem.
// Input: int  $n$ , profit  $p$ , weight  $w$ ,  $m$ 
// Output: Solution  $x$ .
1 Algorithm  $DKP(n, p, w, m, x)$ 
2 {
3      $S_0^1 := \{(0, 0)\}$ ;
4     for  $i := 1$  to  $n - 1$  do {
5          $S_1^i := \{(p + p_i, w + w_i) | (p, w) \in S_0^i \text{ and } w + w_i \leq m\}$ ;
6          $S_0^{i+1} := \text{MergePurge}(S_0^i, S_1^i)$ ;
7     }
8      $(p_x, w_x) := \text{last pair in } S_0^n$ ;
9      $(p_y, w_y) := (p' + p_n, w' + w_n)$  where  $w'$  is the largest  $w'$  for any pairs
10     $(p', w') \in S_0^n$  such that  $w' + w_n \leq m$ ;
11    if  $(p_x > p_y)$  then  $x_n := 0$ ;
12    else  $x_n := 1$ ;
13    TraceBack  $x_{n-1}, \dots, x_1$ ;
14 }
    
```

0/1 Knapsack — Example Revisited

- Given 3 objects, $(p_1, p_2, p_3) = (1, 2, 5)$, $(w_1, w_2, w_3) = (2, 3, 4)$, and $m = 6$. Find the optimal 0/1 knapsack solution, (x_1, x_2, x_3) , $x_i = 0$ or $x_i = 1$,

$1 \leq i \leq 3$, that maximizes the profit, $P = \sum_{i=1}^3 p_i x_i$.

- The sets of feasible solutions are derived as the following.

$$\begin{aligned}
 S_0^1 &= \{(0, 0)\} \\
 S_1^1 &= \{(1, 2)\} \\
 S_0^2 &= \{(0, 0), (1, 2)\} \\
 S_1^2 &= \{(2, 3), (3, 5)\} \\
 S_0^3 &= \{(0, 0), (1, 2), (2, 3), (3, 5)\}
 \end{aligned}$$

- The last pair in S^2 is $(p_x, p_y) = (3, 5)$, and $(p_y, w_y) = (6, 6)$.
- Thus the optimal solution $\sum p_i x_i = 6$ and $\sum w_i x_i = 6$.
 - Since $p_x \not> p_y$, $x_3 = 1$.
 - Note that $(p_y, w_y) - (5, 4) = (1, 2) \notin S_1^1$, thus $x_2 = 0$.
 - Trace back again, $(1, 2) \in S_1^1$, therefore $x_1 = 1$.
 - Finally we have $(x_1, x_2, x_3) = (1, 0, 1)$ and $\sum p_i x_i = 6$, $\sum w_i x_i = 6$.

0/1 Knapsack — Properties

- Note that lines 9, 10 of Algorithm (6.3.4) actually requires to evaluate S_1^n .
- For the last example, we have

$$S_1^3 = \{(5, 4), (6, 6)\}.$$

since $(7, 7)$ and $(8, 9)$ both have $w + w_n \not\leq m$.

- And the optimal solution can be found when S_0^3 and S_1^3 are merged together which is

$$S_0^4 = \{(0, 0), (1, 2), (2, 3), (3, 5), (5, 4), (6, 6)\}.$$

- Note that comparing $(3, 5)$ and $(5, 4)$, the former has smaller profit, $3 < 5$, but larger weight, $5 > 4$, thus it is not a likely solution.
- The former, $(3, 5)$, is **dominated** by the latter, $(5, 4)$.
- When merging two feasible sets, the dominated solutions should be **purged**.
- Of course, by definition, the solutions with weight larger than m are also purged.

0/1 Knapsack — Dynamic Algorithm

Algorithm 6.3.5. 0/1 Knapsack

```
1 struct PW {
2     double p, w; // for profit and weight of each object
3 }
4 Algorithm DKnap(n, p, w, x, m)
5 // p and w are arrays of n profits and weight; m capacity, x solution.
6 {
7     b[0] := 0; pair[1].p := 0; pair[1].w := 0; // S_0^1
8     t := 1; h := 1; // start and end of S_0^1
9     b[1] := next := 2; // next free spot in pair array
10    for i := 1 to n do { // generate S_0^{i+1}
11        k := t;
12        u := Largest(pair, t, h, w[i], m); // largest u, pair[u].w + w[i] ≤ m.
13        for j := t to u do { // generate S_1^i and merge
14            pp := pair[j].p + p[i]; ww := pair[j].w + w[i];
15            while ((k ≤ h) and (pair[k].w ≤ ww)) do {
16                pair[next].p := pair[k].p; pair[next].w := pair[k].w;
17                next := next + 1; k := k + 1;
18            }
19            if ((k ≤ h) and (pair[k].w = ww)) then {
20                if (pp < pair[k].p) then pp := pair[k].p; // new entry dominated
21                k := k + 1;
22            }
23            if (pp > pair[next - 1].p) then { // new entry is dominating
24                pair[next].p := pp; pair[next].w := ww;
25                next := next + 1;
26            }
27        }
28    }
```

0/1 Knapsack — Dynamic Algorithm, II

```

27         while ((k ≤ h) and (pair[k].p ≤ pair[next - 1].p)) do k := k + 1;
28     }
29     while (k ≤ h) do { // merge remaining terms from S1i
30         pair[next].p := pair[k].p; pair[next].w := pair[k].w;
31         next := next + 1; k := k + 1;
32     }
33     t := h + 1; h := next - 1; b[i + 1] := next; // initialize for S0i+1
34 }
35 TraceBack(n, p, w, m, pair, x); // find solution x
36 }

```

• In the above algorithm

- *pair* is an array to store all feasible solutions, $S_0^i, 0 \leq i \leq n$.
- *b* is an array to store the indices of S_0^i in *pair* array
- Function **Largest**(*pair*, *t*, *h*, *w*[*i*], *m*) finds the largest *u* satisfying

$$\text{pair}[u].w + w[i] \leq m, \quad t \leq u \leq h$$

- The **for** loop of lines 10–34 generates $S_0^i, 1 \leq i \leq n$.
- First S_0^{i-1} is copied into S_0^i
- Then S_1^{i-1} is generated and merged into S_0^i
- Lines 19–26 remove dominated entries

0/1 Knapsack — Example

- Given 3 objects, $(p_1, p_2, p_3) = (1, 2, 5)$, $(w_1, w_2, w_3) = (2, 3, 4)$, and $m = 6$. Find the optimal 0/1 knapsack solution, (x_1, x_2, x_3) , $x_i = 0$ or $x_i = 1$,

$$1 \leq i \leq 3, \text{ that maximizes the profit, } P = \sum_{i=1}^3 p_i x_i.$$

- After executing the algorithm **DKnap**, we have

	1	2	3	4	5	6	7	8	9	10	11	12
<i>pair</i> [<i>·</i>]. <i>p</i>	0	0	1	0	1	2	3	0	1	2	5	6
<i>pair</i> [<i>·</i>]. <i>w</i>	0	0	2	0	2	3	5	0	2	3	4	6
	↑	↑		↑				↑				
	<i>b</i> [0]	<i>b</i> [1]		<i>b</i> [2]				<i>b</i> [3]				

- Note that $(p, w) = (3, 5) \in S_0^3$ but not S_0^4 since it is dominated by $(5, 4)$.
- The last entry, $(pp, ww) = (6, 6)$, is the optimal solution.

0/1 Knapsack — Example

- To find if each object is placed into the sack or not, $x[i], 1 \leq i \leq n$.
- One starts from $i = n$ and trace back to 1.
 - The optimal solution is (pp, ww) ,
 - If $(pp, ww) \in S_0^n$ then $x[n] = 0$
 - $(pp_{n-1}, ww_{n-1}) = (pp, ww)$.
 - Otherwise $x[n] = 1$,
 - $(pp_{n-1}, ww_{n-1}) = (pp - p[n], ww - w[n])$.
- Repeat checking for S_0^{n-i} and update (pp_{n-i}, ww_{n-i}) , one finds the solution $x[i], 1 \leq i \leq n$.
- For the last example,
 - $(6, 6) \notin S^2$, thus $x[3] = 1$,
 - $(1, 2) \in S^1$, and $x[2] = 0$,
 - $(1, 2) \notin S^0$, thus $x[1] = 1$.
 - Optimal solution $x = (1, 0, 1)$, $(p, w) = (6, 6)$.

0/1 Knapsack — Complexity

- Let the space needed to store S_0^i in *pair* be $|S_0^i|$, then

$$|S_0^i| \leq 2^{i-1}$$

And the total space needed for *pair* is

$$\sum_{i=1}^n |S_0^i| \leq \sum_{i=1}^n 2^{i-1} = 2^n - 1$$

- Thus the space complexity is $\mathcal{O}(2^n)$
- The time needed to generate S_0^i is $\Theta(|S_0^{i-1}|)$, therefore the total time to generate all pairs is

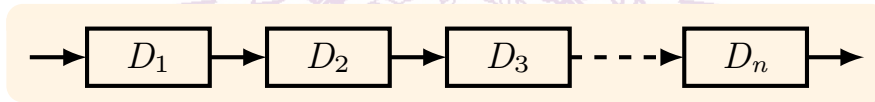
$$\sum_{i=1}^n |S_0^{i-1}| \leq \sum_{i=1}^{n-1} 2^{i-1} = 2^{n-1} - 1$$

and the time complexity is $\mathcal{O}(2^n)$.

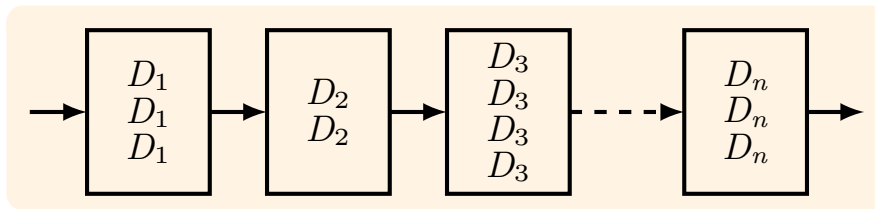
- The time complexity of the **Traceback** function is $\mathcal{O}(n^2)$ since it involves n searches in the range $b[i]$ and $b[i+1]$.
 - Each search can take $\log(|S_0^i|) = \log(2^{i-1}) = (i-1) \log 2$.
 - Total time is $\sum_{i=1}^n (i-1) \log 2 = \mathcal{O}(n^2)$.

System Reliability

- Suppose a system is composed of n stages of devices connected in series.
 - Let r_i be the reliability of device D_i – the probability that device D_i function normally.
 - Then the reliability of the system is $\prod_{i=1}^n r_i = r_1 r_2 \cdots r_n$.



- To improve the reliability of the system, one can replace stage i by multiple, m_i , devices connected in parallel.
 - Then the reliability of stage i becomes $\phi_i(m_i) = 1 - (1 - r_i)^{m_i}$.
 - The system reliability becomes $\prod_{i=1}^n \phi_i(m_i)$.



Reliability Design Problem

- Assuming device D_i costs c_i each piece, and the total cost of the entire system is c , the **reliability design problem** is to find the multiplicity of each device, m_i for each D_i such that

$$\begin{aligned} & \text{maximize} && \prod_{i=1}^n \phi_i(m_i) \\ & \text{subject to} && \sum_{i=1}^n c_i m_i \leq c \\ & && \text{and } m_i \in N \text{ and } m_i \geq 1, \quad 1 \leq i \leq n. \end{aligned} \tag{6.3.3}$$

- Since $m_i \geq 1$ and $\sum c_i = c$, we can define

$$u_i = \lfloor (c + c_i - \sum_{j=1}^n c_j) / c_i \rfloor \tag{6.3.4}$$

- And the reliability design problem can be reformulated as

$$\begin{aligned} & \text{maximize} && \prod_{i=1}^n \phi_i(m_i) \\ & \text{subject to} && \sum_{i=1}^n c_i m_i \leq c \\ & && \text{and } 1 \leq m_i \leq u_i. \end{aligned} \tag{6.3.5}$$

- Given the n stages and the total cost of the optimal solution is $f_n(c)$, then the multiplicity, m_n , for stage n should be determined by

$$f_n(c) = \max_{m_n=1}^{u_n} \left(\phi_n(m_n) \cdot f_{n-1}(c - c_n m_n) \right) \quad (6.3.6)$$

It is also assumed that $f_0(c) = 1$ for any c .

- Then this problem is similar to the 0/1 knapsack problem and the dynamic approach can be used to find the solution of the problem.
- Example, 3 devices, D_1 , D_2 and D_3 , with $r_1 = 0.9$, $r_2 = 0.8$, $r_3 = 0.5$, $c_1 = 30$, $c_2 = 15$, $c_3 = 20$, and the total cost $c \leq 105$. (It can be derived that $u_1 = 2$, $u_2 = 3$ and $u_3 = 3$).

Summary

- String editing problem
 - $\mathcal{O}(mn)$
- 0/1 knapsack problem
 - $\mathcal{O}(2^n)$
- System reliability design
 - Large time complexity