Lab05-DynamicProgramming

CS214-Algorithm and Complexity, Xiaofeng Gao, Spring 2021.

∗ If there is any problem, please contact TA Haolin Zhou.
∗ Name:Yanjie Ze Student ID:519021910706 Email: zeyanjie@sjtu.edu.cn

1. *Optimal Binary Search Tree.* Given a sorted sequence $K = \langle k_1, k_2, \ldots, k_n \rangle$ of n distinct keys, and we wish to build a binary search tree from these keys. For each key k_i, we have a probability p_i that a search will be for k_i. Some searches may be for values not in K, and so we also have $n+1$ *dummy keys* $d_0, d_1, d_2, \ldots, d_n$ representing values not in K. In particular, d_0 represents all values less than k_1, and d_n represents all values greater than k_n. For $i = 1, 2, \ldots, n-1$, the dummy key d_i represents all values between k_i and k_{i+1}. For each dummy key d_i, we have a probability q_i that a search will correspond to d_i. Each key k_i is an internal node, and each dummy key d_i is a leaf. Every search is either successful (finding some key k_i) or unsuccessful (finding some dummy key d_i), and so we have $\sum_{i=1}^{n} p_i + \sum_{i=0}^{n} q_i = 1$.

   (a) Prove that if an optimal binary search tree T (T has the smallest expected search cost) has a subtree T' containing keys $k_i, \ldots, k_j$, then this subtree T' must be optimal as well for the subproblem with keys $k_i, \ldots, k_j$ and dummy keys $d_{i-1}, \ldots, d_j$.

   Solution.
   Denote $E(cost^*)$ as the smallest expected search cost of an optimal binary search tree T containing keys $k_1, k_2, ..., k_n$ and dummy keys $d_0, d_1, ..., d_n$.
   Denote $E(subcost^*)$ as the smallest expected search cost of an optimal binary search tree T'_{opt} containing keys $k_i, ..., k_j$ and dummy keys $d_{i-1}, ..., d_j$ and denote $E(subcost)$ as the expected search cost of a binary search tree T'_{norm} covering the same keys as T, but T'_{norm} is not optimal though.
   By definition, $E(subcost^*) < E(subcost)$.

   To search an element ranging from d_{i-1} to d_j, we can divide this into 2 subproblems:

   - Reach the subtree. Assume the cost of reaching the subtree from the root is C.
   - Search in the subtree. The cost depends on the subtree.

   One key problem here is: Whether C is the same for both subtrees?
   The answer is YES.
   Let's assume the cost C is not the same, which means: although the normal subtree's expected cost is more than the optimal, its C is less than C of the optimal subtree. As shown in Fig. 1 and Fig. 3, although the optimal subtree costs less in itself, there needs larger cost C to reach the optimal subtree, while in Fig. 3, the subtree is not optimal but it has smaller cost C. The situation seems to be hard to judge.
   However, this is not possible. Because if the normal subtree in Fig. 3 exists, this subtree will certainly be improved to an optimal one, as shown in Fig. 4.
   Therefore, the cost of reaching the subtree for a normal subtree T'_{norm} and its corresponding optimal subtree T'_{opt}, C, should be the same.

   Then we prove the subtree should be optimal.
   If T's subtree is not optimal, assume it is T'_{norm}, as shown in Fig. 2, then:

$$E(cost^*) = E(subcost) + C > E(subcost^*) + C$$

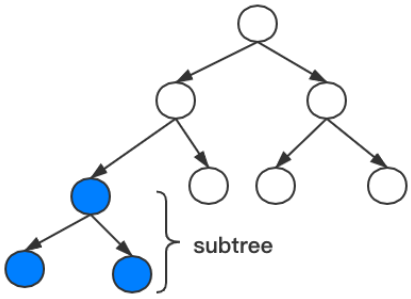

Figure 1: Optimal Subtree

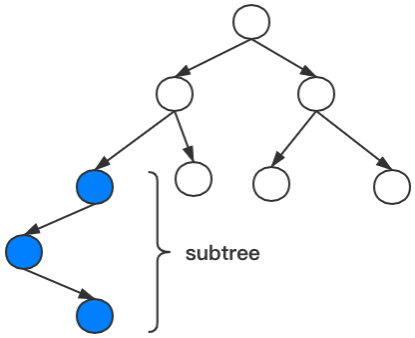

Figure 2: Normal Subtree

Which means T is not optimal, leading to a contradiction.

Therefore, the subtree of T should be optimal, and naturally the subtree is optimal for the subproblems with keys $k_i, ..., k_j$ and dummy keys $d_{i-1}, ..., d_j$.

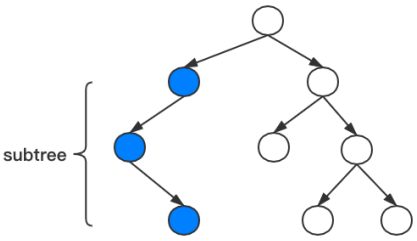

Figure 3: Is C the same?(1)

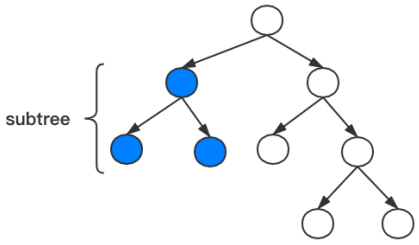

Figure 4: Is C the same?(2)

□

(b) We define $e[i, j]$ as the expected cost of searching an optimal binary search tree containing the keys $k_i, \ldots, k_j$. Our goal is to compute $e[1, n]$. Write the state transition equation and pseudocode using **dynamic programming** to find the minimum expected cost of a search in a given binary tree. (**Remark**: You may use $w(i, j) = \sum_{l=i}^{j} p_l + \sum_{l=i-1}^{j} q_l$).

Solution.

Denote r as the root of the subtree.

Denote $w(i, j) = \sum_{l=i}^{j} p_l + \sum_{l=i-1}^{j} q_l$, as the probability of reaching root r from last node.

Then we can divide the calculation of $e[i, j]$ into subproblems:

- Figure out every $e[i, r-1]$ and $e[r+1, j]$, for $i \leq r \leq j$.
- Find r that can minimize $e[i, r-1] + e[r+1, j] + w[i, j]$.

Therefore, we have such state transition equation:

$$e[i,j] = \begin{cases} 0, & when \quad i - j > 1 \\ q_j, & when \quad i - j = 1 \\ \min\limits_{i \leq r \leq j} \{e[i, r-1] + e[r+1, j] + w[i, j]\}, & when \quad i \leq j \end{cases}$$

Another essential problem is how we calculate $e[i, j]$ in a bottom-up manner.

It's acknowledged that the brute force algorithm is much time-consuming, thus we need to figure out a tabular method.
Obviously:

$$\begin{aligned} e[i,i] &= e[i,i-1] + e[i+1,i] + w[i,i] \\ &= q[i-1] + q[i] + w[i,i] \end{aligned} \tag{1}$$

So we first compute these $e[i,i]$ using the formula 1, as shown in Fig. 5. In Fig. 5, the red dots represent these computed $e[i,j]$.

Then taking $e[1,2]$ as the example, we show what can be computed next step.

$$\begin{aligned} e[1,2] &= min\{e[1,0] + e[2,2] + w[1,2], e[1,1] + e[3,2] + w[1,2]\} \\ &= min\{q_0 + e[2,2] + w[1,2], q_2 + e[1,1] + w[1,2]\} \end{aligned} \tag{2}$$

Similarly, we can compute $e[i,i+1]$ next step, based on $e[i,i]$ and $e[i+1,i+1]$, so we have Fig. 6.The blue dots represent these computed $e[i,i+1]$ and the dotted lines between them represent the connection of computation.

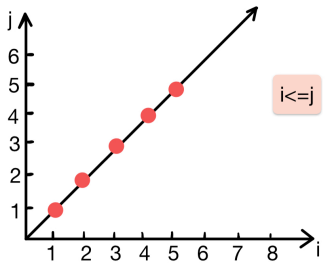

Figure 5: Compute $e[i,i]$

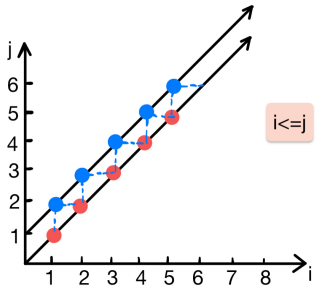

Figure 6: Compute $e[i,i+1]$

Finally, we are able to conclude the correct computation path of $e[i,j]$, shown in Fig. 7. We start from calculating each $e[i,i]$, then each $e[i,i+1]$, each $e[i,i+2]$, ... Intuitively, this computation path is the same as the colorful line.

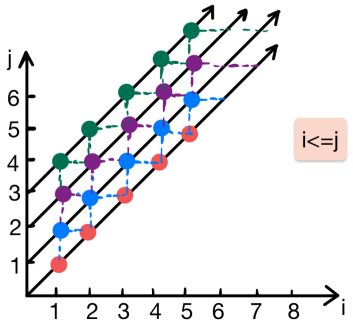

Figure 7: Compute All $e[i,j]$

Based on the computation path constructed before, we propose Alg. 1 **Optimal Binary Search Tree Bottom-up Dynamic Programming**. Line 1 to Line 6 is initialization.

Line 7 to Line 9 is the implementation of the computation path.

Algorithm 1: OBST Bottom-up DP

```
1  for i = 0 → n do
2      e[i+1, i] = q[i];
3      w[i+1, i] = q[i];
4  for i = 1 → n do
5      for j = i → n do
6          w[i, j] = Σ_{l=i}^{j} p_l + Σ_{l=i-1}^{j} q_l
7  for k = 0 → n − 1 do
8      for i = 1 → n − k do
9          e[i, i+k] = min_{i≤r≤i+k}{e[i, r−1] + e[r+1, i+k] + w[i, i+k]}
10 return e[1, n]
```

□

(c) Implement your proposed algorithm in C/C++ and analyze the time complexity. (The framework Code-OBST.cpp is attached on the course webpage). Give the minimum search cost calculated by your algorithm. The test case is given as following:

i	0	1	2	3	4	5	6	7
p_i		0.04	0.06	0.08	0.02	0.10	0.12	0.14
q_i	0.06	0.06	0.06	0.06	0.05	0.05	0.05	0.05

Solution.
The algorithm is implemented in **Code-OBST.cpp**.

The cost of the optimal binary search tree is: **3.12**, and the output of the program is shown in Fig. 8.

```
yanjieze@YanjieZedeMacBook-Pro Lab05-YanjieZe % ./obst
The cost of the optimal binary search tree is: 3.12
The structure of the optimal binary search tree is:
k5 is the root
k2 is the left child of k5
k7 is the right child of k5
k1 is the left child of k2
k3 is the right child of k2
d0 is the left child of k1
d1 is the right child of k1
k4 is the right child of k3
d2 is the left child of k3
d3 is the left child of k4
d4 is the right child of k4
k6 is the left child of k7
d7 is the right child of k7
d5 is the left child of k6
d6 is the right child of k6
```

Figure 8: Output of OBST Bottom-up DP

Time Complexity:$O(n^3)$
From Line 1 to Line 3, time complexity is $O(n)$.

From Line 4 to Line 6, the computation times are:

$$\begin{aligned} t_1 &= \sum_{i=1}^{n}\sum_{j=i}^{n}((j-i+1)+(j-i+2)) \\ &= \sum_{i=1}^{n}\sum_{j=i}^{n}(2j-2i+3) \\ &= \sum_{i=1}^{n}\{(n-i)^2+4(n-i)+3\} \\ &= \frac{1}{3}n^3+\frac{3}{2}n^2+\frac{7}{6}n \\ &= O(n^3) \end{aligned} \tag{3}$$

From Line 7 to Line 9, the computation times are:

$$\begin{aligned} t_2 &= \sum_{k=0}^{n-1}\sum_{i=1}^{n-k}k \\ &= \sum_{k=0}^{n-1}(nk-k^2) \\ &= \frac{n^3-n}{6} \\ &= O(n^3) \end{aligned} \tag{4}$$

Therefore, overall time complexity is $O(n^3)$.

Space Complexity:$O(n^2)$

In the algorithm we need two-dimensional arrays to store the middle results, which are e, w, and one-dimensional arrays, which are p, q.

Then the space complexity is $O(n^2)$.

□

(d) Please draw the structure of the optimal binary search tree in the test case, and explain the drawing process.

Solution.

Since we have gained $e[i, j]$ needed after running Alg. 1, we can figure out what the optimal binary search tree is by utilizing $e[i, j]$.

Here we do some changes on the original Alg. 1. Define $root[i, j]$ as the root of the subtree

containing $k_i, ..., k_j$ and $d_{i-1}, ..., d_j$.

Algorithm 2: OBST Bottom-up DP Modified

```
for i = 0 → n do
    e[i + 1, i] = q[i];
    w[i + 1, i] = q[i];
for i = 1 → n do
    for j = i → n do
        w[i, j] = Σ_{l=i}^{j} p_l + Σ_{l=i-1}^{j} q_l
for k = 0 → n − 1 do
    for i = 1 → n − k do
        e[i, i + k] = min_{i≤r≤i+k}{e[i, r − 1] + e[r + 1, i + k] + w[i, i + k]};
        root[i, i + k] = r;
return e[1, n]
```

Utilizing $root[i, j]$, we can easily use **First Order Traverse** to get each roor and node.

Take the test case as an example:

$root[1, 7] = 5 \rightarrow k_5$ *is the root*

$root[1, 5 - 1] = 2 \rightarrow k_2$ *is the left child of* k_5

And the rest of the result is shown in Fig. 9.

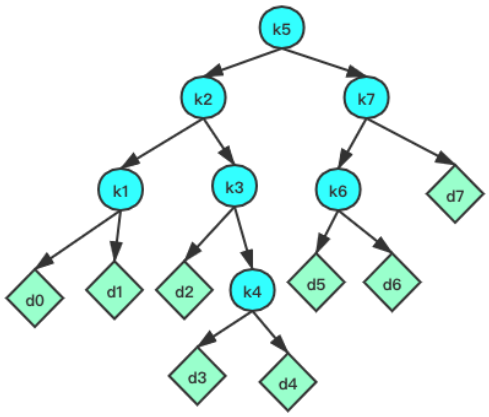

Figure 9: Optimal Binary Search Tree (test)

□

2. *Dynamic Time Warping Distance.* **DTW** stretches the series along the time axis in a dynamic way over different portions to enable more effective matching. Let $DTW(i, j)$ be the optimal distance between the first i and first j elements of two time series $\bar{X} = (x_1 \dots x_n)$ and $\bar{Y} = (y_1 \dots y_m)$, respectively. Note that the two time series are of lengths n and m, which may not be the same. Then, the value of $DTW(i, j)$ is defined recursively as follows:

$$DTW(i, j) = |x_i - y_j| + \min(DTW(i, j - 1), DTW(i - 1, j), DTW(i - 1, j - 1))$$

   (a) Implement the proposed DTW algorithm in C/C++ and analyze the time complexity of your implementation. (The framework Code-DTW.cpp is attached on the course webpage). Two test cases have been given in the source code.

Solution.
Test result:0, 9.45455.
The computation path of tabular Dynamic Programming is shown in Fig. 10, which is different from the computation path shown in Fig. 7.

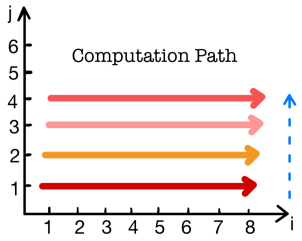

Figure 10: Computation Path for DTW DP

Time Complexity:$O(mn)$
First, we need to initialize a cost matrix DTW, which has $O(mn)$ time complexity.
Second, based on matrix DTW we consume $O(m+n)$ time to find the minimal cost path, starting from the top right corner and traversing to bottom left.
In total, Time Complexity is $O(mn)$.

□

(b) The window constraint imposes a minimum level w of positional alignment between matched elements. The window constraint requires that $DTW(i,j)$ be computed only when $|i-j| \leq w$. Modify your code to add a window constraint and give the results of $w=0$ and $w=1$ on the two test cases.

Solution.
When *window constraint* $= 0$, two results are 55.9286, 20.8889.
When *window constraint* $= 1$, two results are 0, 13.9.
The output of the program is shown in Fig. 11 . (**Note**, when *window constraint* is large, the result is the same as before).

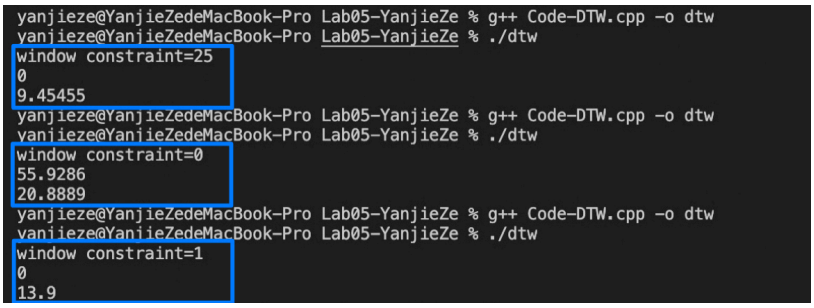

Figure 11: Results for DTW

□

Remark: You need to include your .pdf and .tex and 2 source code files in your uploaded .rar or .zip file. Screenshots of test case results are acceptable.