Lab05-DynamicProgramming

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

∗ If there is any problem, please contact TA Haolin Zhou.
∗ Name:Renyang Guan Student ID:519021911058 Email: guanrenyang@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 $d_{i-1}, \ldots, d_j$.

   (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 $w(i, j) = \sum_{l=i}^{j} p_l + \sum_{l=i-1}^{j} q_l$).

   (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

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

Solution.

(a) **Proof.** ***Counter evidence:*** If the binary search tree T is optimal, suppose that there exists a subtree T'' whose expected cost is smaller than T'. We could cut T'' and substitute T' with T'' to make the total expected cost lower. Since the binary search tree T is optimal, the substitution is impossible. □

(b) ***State transition equation:***

$$e[i, j] = \begin{cases} q_{i-1} & j = i - 1 \\ \min\limits_{i \le r \le j} \{e[i, r-1] + e[r+1, j] + \omega(i, j)\} & i \le j \end{cases} \tag{1}$$

Algorithm 1: Pseudocode of optimal binary search tree

Input: p, q, n
Output: Array e, array $root$

initialize three 2-dimentional arrays:;
$e[1, \cdots, n+1][0, \cdots, n]$;
$\omega[1, \cdots, n+1][0, \cdots, n]$;
$root[1, \cdots, n][1, \cdots, n]$;
for $i \leftarrow 1$ **to** $n+1$ **do**
    $e[i, i-1] \leftarrow q_{i-1}$;
    $\omega[i, i-1] \leftarrow q_{i-1}$;
for $t \leftarrow 1$ **to** n **do**
    for $i \leftarrow 1$**to** $n-t+1$ **do**
        $j = i + t - 1$;
        $e[i, j] \leftarrow \infty$;
        $\omega[i, j] = \omega[i, j-1] + p_j + q_j$;
        for $l = i$ **to** j **do**
            $temp \leftarrow e[i, t-1] + e[t+1, j] + \omega[i, j]$;
            if $temp \leq e[i, j]$ **then**
                $e[i, j] = temp$;
                $root[i, j] \leftarrow l$;

(c) The source code is shown in file *Code-OBST.cpp*.

Result of the program:

```
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
k1 is the left child of k2
d1 is the right child of k1
d0 is the left child of k1
k3 is the right child of k2
d2 is the left child of k3
k4 is the right child of k3
d4 is the right child of k4
d3 is the left child of k4
k7 is the right child of k5
k6 is the left child of k7
d6 is the right child of k6
d5 is the left child of k6
d7 is the right child of k7
```

Figure 1: Program Result

Time complexity analysis:

Assume that all the operations with time complexity $O(1)$ are represented by 1 in the formula below.

$$Time\ complexity = n+1+\sum_{t=1}^{n}\sum_{i=1}^{n-t+1} t = n+1+\sum_{t=1}^{n} -t^2+(n+1)t = \frac{1}{6}n^3+\frac{1}{2}n^2+\frac{4}{3}n+1 = \Omega(n^3)$$

The minimum search cost is 3.12.

(d) The optimal binary search tree is shown in Fig. 2:

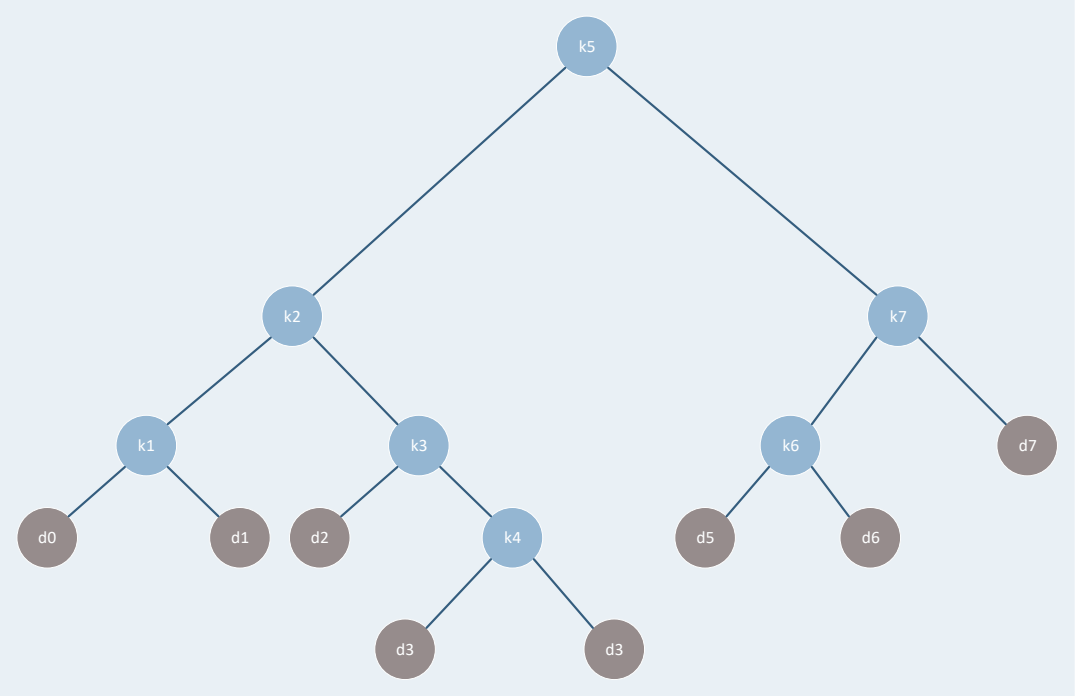

Figure 2: Optimal Binary Search Tree

Explanation of the drawing process:

The drawing process could be divided into two parts: one is the *drawing keys* and the other is *drawing dummy keys.*

k: In terms of drawing k nodes, I use a recursive method. Since element $e[i, j]$ stores the root of the subtree of $\{k_i, k_{i+1}, \cdots, k_j\}$ the subtree of $\{k_i, k_{i+1}, \cdots, k_{e[i,j]-1}\}$ is the left subtree of $k_{e[i,j]-1}$ and the subtree of $\{k_{e[i,j]+1}, k_{e[i,j]+2}, \cdots, k_j\}$ is the left subtree of $k_{e[i,j]-1}$, where $i < e[i, j] < j$. Such method could draw an optimal binary search tree of nodes $\{k_1, k_2, \cdots, k_n\}$ if the input is $i = 1$ and $j = n$.

d: The key point is to judge whether a key is a leaf of the binary search tree. Considering the property of the array e, if node $e[i, j]$ has no left subtree, $e[i, j] = i$ must be true.(Otherwise the subtree of $\{k_i, k_{i+1}, \cdots, k_{e[i,j]-1}\}$ must contains at least one *key* node) Thus we could draw dummy keys by conditional branches. If $e[i, j] = i$, the left subtree of *key* node $K_{e[i,j]}$ is a *dummy node* $d_{e[i,j]-1}$. If $e[i, j] = j$, the right subtree of *key* node $K_{e[i,j]}$ is a *dummy node* $d_{e[i,j]}$.

□

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.

(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.

Explanation of vagueness:

1. *Normalized distance*

$$Normalized\ distance = \frac{\sum_{(i,j)\in\ path} DTW[i, j]}{Number\ of\ nodes\ in\ path}$$

2. *The rule of window constraint condition*

   The sentence *The window constraint requires that* $DTW(i, j)$ *be computed only when* $|i - j| \leq w$. in the question means that $DTW(i, j)$ does not exist and we have no idea computing it when $|i - j| \leq w$, so we could set $+\infty$ to $DTW(i, j)$ mathematically in such condition.

Result of the program:

```
Case 1, no window constraint: 0
Case 1, window constraint=0 : 52.2
Case 1, window constraint=1 : 0

Case 2, no window constraint: 8.66667
Case 2, window constraint=0 : 18.8
Case 2, window constraint=1 : 12.6364
```

Figure 3: Program Result

Time complexity analysis:

The process of computing the *time normalized distance* has three serial parts:

Fill in the DTW **matrix:** The process only need a two-level for loop to traverse the DTW matrix of size $n \times m$, so the time complexity of it is $\Theta(mn)$.

Identify the warping path: The operation will trace from $D[n, m]$ back to $D[0, 0]$. The worst case is that no step is diagonal, of which the time complexity is $O(m + n)$. As a result, the average time complexity will not exceed $O(m, n)$.

Calculate th time normalized distance: The operation sums the DTW value of each node in the path. The longest possible length of the path is $n + m - 1$, so the worst time complexity is $O(m + n)$. The Average time complexity will not exceeds $O(m + n)$.

Conclusion:

$$Time\ complexity = \Theta(mn) + 2O(m + n) = \Theta(mn)$$

Modification of window constraint: The two modifications of the source code is explained as *comment* in the line $27 - 30$ and line $54 - 59$ of the file *Code-DTW.cpp*. Screen shoots of the two comments are shown in Fig. **??**. and Fig. **??**

```
27  /*
28  Modication 1:
29  If you input the window_constraint, the comparison will be operated. Otherwise, things will go on and we have no need to deal with the confusin
30  */
```

Figure 4: Screen shoot of modification 1

```
54  /*
55  Modification 2
56  Since that if the window constraint is too small, D[n,m] will no longer exist if |n-m|>window_constrainst.
57  I don't think that I had a deep understanding of the rule of modication. Whatever, I assume that D[n,m] can't be computed if |n-m|>window_const
58  As a result, I do the "tracking back" operation beginning with D[n,n+window_constraint] or D[m+window_constraint,m] if |n-m|>window_constraint.
59  */
```

Figure 5: Screen shoot of modification 2

□

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.