Lab05-DynamicProgramming

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

$*$ If there is any problem, please contact TA Haolin Zhou.

$*$ Name: Zilong Li   Student ID: 518070910095   Email: logcreative-lzl@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.

   (a) **Proof. Prove by contradiction.** Suppose this subtree T' is not optimal, then there exists a subtree T'' is better than T' on expected search cost with keys $k_i, \cdots, k_j$ and dummy keys $d_{i-1}, \cdots, d_j$. Then the expeceted search cost between the two subtrees has the following relationship:

$$e'' < e' \tag{1}$$

   Assume the summation of the probability on the subtree is P. And the summation of search cost in the remaining part is C. Then the Eq. (1) follows:

$$e''P + C < e'P + C$$

   which means that if T' is replaced by the optimal subtree T'', a better binary search tree could be contructed than T. The contradiction over the optimal of the original binary search tree shows that the subtree T' is optimal. □

   (b) **Solution.** Consider the root of the optimal binary search tree containing keys $k_i, \cdots, k_j$. If the root is $k_r (i \le r \le j)$, then according to the previous subproblem, the left subtree containing $k_i, \cdots, k_{r-1}$ and the right subtree containing $k_{r+1}, \cdots, k_j$ are also the optimal binary search tree.

If $j = i - 1$, the subtree only has one dummy key d_{i-1}, and the expected search cost is $e[i, i-1] = q_{i-1}$.

If $j \geq i$, because once one tree become a child, the search cost for every node will be increased by 1 and the expected cost could be increased by the summation of probability in the subtree. Then the expected search cost could be calculated recursively by calculating for every $i \leq r \leq j$:

$$\begin{aligned} e[i,j] &= p_r + (e[i,r-1] + w[i,r-1]) + (e[r+1,j] + w[r+1,j]) \\ &= e[i,r-1] + e[r+1,j] + w[i,j] \end{aligned}$$

As a result, the state transition equation could be stated as follows:

$$e[i,j] = \begin{cases} q_{i-1}, & j = i-1, \\ \min\limits_{i \leq r \leq j} e[i,r-1] + e[r+1,j] + w[i,j], & j \geq i \end{cases}$$

The algorithm is shown in Alg. 1 and function `getExpectedCost`(i,j).

Function getExpectedCost(i,j)

Data: start i, end j
Output: the expected cost $e[i,j]$

if $e[i,j]$ *is initialized* **then return** $e[i,j]$;
$r_{\min} \leftarrow i, e_{\min} \leftarrow \infty$;
for $r \leftarrow i$ *to* j **do**
  $e' \leftarrow$ `getExpectedCost` $(i,r-1)$ + `getExpectedCost` $(r+1,j)$+$w[i,j]$;
  if $e' \leq e_{min}$ **then**
    $r_{\min} \leftarrow r$;
    $e_{\min} \leftarrow e'$;
$root[i,j] \leftarrow r_{\min}, e[i,j] \leftarrow e_{\min}$;
return e_{min};

Algorithm 1: Find the optimal binary search tree

Input: key probability $\{p_i\}_{i=1}^n$, dummy key probability $\{q_i\}_{i=0}^n$, input size n
Output: expected cost $e[1,n]$

Initialize $w[i,j] = \sum_{l=i}^{j} p_l + \sum_{l=i-1}^{j} q_l$;
Initialize $e[i,i-1] = q_j$;
Initialize $root[i,i-1] = i-1$;
`getExpectedCost` $(1,n)$

□

(c) The implementation is as follows:

Listing 1: Code-OBST.cpp

```
#include <iostream>
using namespace std;

```

```
#define MAX 10000

const int n = 7;
double p[n + 1] = {0,0.04,0.06,0.08,0.02,0.10,0.12,0.14};
double q[n + 1] = {0.06,0.06,0.06,0.06,0.05,0.05,0.05,0.05};

int root[n + 1][n + 1];//Record the root node of the optimal subtree
double e[n + 2][n + 2];//Record the expected cost of the subtree
double w[n + 2][n + 2];//Record the probability sum of the subtree

double get_expected_cost(int i, int j) {
    if (e[i][j] != MAX) return e[i][j];
    int minr = i;
    double mine = MAX;
    for (int r = i; r <= j; ++r) {
        double tmp = get_expected_cost(i, r - 1) + get_expected_cost(r + 1, j) + w[i][j];
        if (tmp <= mine) {
            minr = r;
            mine = tmp;
        }
    }
    root[i][j] = minr;
    e[i][j] = mine;
    return mine;
}

void optimal_binary_search_tree(double *p,double *q,int n)
{

    // Initialize w
    for (int i = 1; i <= n; ++i) {
        w[i][i] = p[i] + q[i - 1] + q[i];
        for (int j = i + 1; j <= n; ++j)
            w[i][j] = w[i][j - 1] + p[j] + q[j];
    }

    // Initialize e
    for (int i = 0; i <= n+1;++i)
        for (int j = 0; j <= n+1; ++j)
            if (j == i - 1) e[i][j] = q[j];
            else e[i][j] = MAX;  // Uninitialized

    // Initialize root
    for (int i = 0; i <= n+1; ++i)
        for (int j = 0; j <= n+1; ++j)
            if (j == i - 1) root[i][j] = j;
            else root[i][j] = -1; // Uninitialized

    //The result is stored in e.
    get_expected_cost(1, n);
}

/*
You can print the structure of the optimal binary search tree based on root[][]
The format of printing is as follows:
k2 is the root
k1 is the left child of k2
d0 is the left child of k1
d1 is the right child of k1
k5 is the right child of k2
k4 is the left child of k5
k3 is the left child of k4
d2 is the left child of k3
d3 is the right child of k3
d4 is the right child of k4
d5 is the right child of k5
*/
void construct_optimal_bst(int i,int j, int rt, int loc)
{
//You can adjust the number of input parameters
    bool dummy = false;
    if (j == i - 1) dummy = true;

    cout << (dummy ? "d" : "k") << root[i][j];
    switch (loc) {
    case 0: cout << " is the root" << endl; break;
```

```
    case -1: cout << " is the left child of k" << rt << endl; break;
    case 1: cout << " is the right child of k" << rt << endl; break;
    }

    if (dummy) return;
    construct_optimal_bst(i, root[i][j] - 1, root[i][j], -1);
    construct_optimal_bst(root[i][j] + 1, j, root[i][j], 1);
}

int main()
{
    optimal_binary_search_tree(p,q,n);
    cout<<"The cost of the optimal binary search tree is: "<<e[1][n]<<endl;
    cout << "The structure of the optimal binary search tree is: " << endl;
    construct_optimal_bst(1,n,root[1][n],0);
}
```

The time complexity is analyzed as follows:

Initialization. The initalization of $w, e, root$ costs $O(n^2)$.

Calculation. The recurrence of the time complexity is:

$$\begin{aligned} T[i, i-1] &= O(1) \\ T[i, j] &= \sum_{i \le r \le j} \left(T[i, r-1] + T[r+1, j] + O(1)\right) \end{aligned}$$

A recurrence tree could be drawn in Figure 1. The diagonal is first calculated and every step in the figure is $O(1)$.

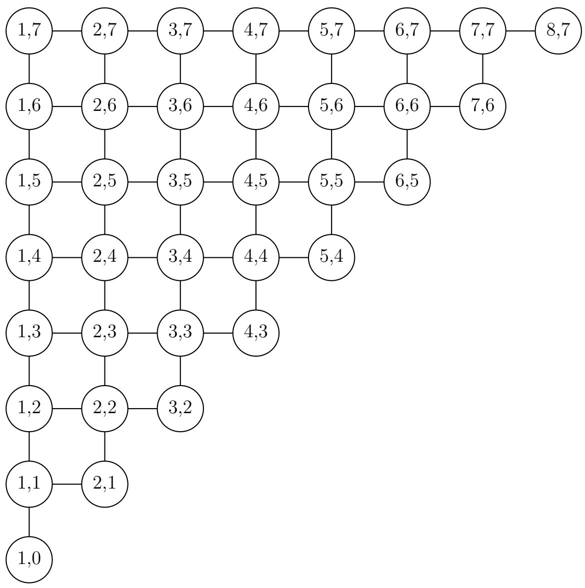

Figure 1: Time Complexity Analysis

So, the time complexity is $((n+1)^2 + n + 1)/2 = O(n^2)$.

As a result, the time complexity comes to $O(n^2)$.

And the running result is:

The cost of the optimal binary search tree is: 3.12

(d) The continued running result is:

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

According to the output, the binary search tree could be drawn in Fig. 2.

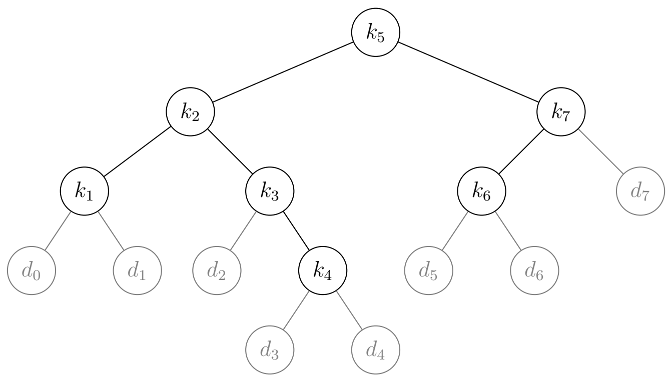

Figure 2: The optimal binary search tree

The drawing process could be described in function `ConstructOptimalBst(`i,j,rt,loc`)`, which is in the mid order by recurrence.

Function ConstructOptimalBst(i,j,rt,loc)

Data: start i, end j, the previous root rt and the relative location to the previous root loc

if $j == mi - 1$ **then** $dummy \leftarrow true$;
else $dummy \leftarrow false$;
Output the (dummy) key;
switch loc **do**
    case *0* **do** Output root;
    case *1* **do** Output left of rt;
    case *2* **do** Output right of rt;
if $dummy = true$ **then return**;
`ConstructOptimalBst` (i,$root[i,j] - 1$,$root[i][j]$,-1);
`ConstructOptimalBst` ($root[i,j] + 1$,j,$root[i][j]$,1);

---

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. (a) The implementation code.

Listing 2: Code-DTW.cpp

```
#include <iostream>
#include <vector>
#include <cmath>
#include <numeric>
/*
The process to calculate the dynamic can be divided into four steps:
1.Create an empty cost matrix DTW with X and Y labels as amplitudes of the two series to be compared.
2.Use the given state transition function to fill in the cost matrix.
3.Identify the warping path starting from top right corner of the matrix and traversing to bottom left. The traversal path is identified based on the neighbor with minimum value.
i.e., When we reach the point (i, j) in the matrix, the next position is to choose the point with the smallest cost among (i-1,j-1), (i,j-1), and (i-1,j),
For the sake of simplicity, when the cost is equal, the priority of the selection is (i-1,j-1), (i,j-1), and (i-1,j) in order.
4.Calculate th time normalized distance. We define it as the average cost of the selected points.
*/
using namespace std;
double distance(vector<int> x, vector<int> y) {
    int n = x.size();
    int m = y.size();
    vector<vector<int>> DTW(n, vector<int>(m, -1));
    // Use the given state transition function to fill in the cost matrix. -- diagonal calculation
    // Clear the left and bottom border first.
    DTW[0][0] = abs(x[0] - y[0]);
    for (int i = 1; i < n; ++i)
        DTW[i][0] = abs(x[i] - y[0]) + DTW[i - 1][0];
    for (int j = 1; j < m; ++j)
        DTW[0][j] = abs(x[0] - y[j]) + DTW[0][j - 1];
    // traverse diagonally as a snake
    int i = 1, j = 1;
    int dir = 1;
    bool flag = false;
    while (true) {
        DTW[i][j] = abs(x[i] - y[j]) + min(min(DTW[i][j - 1], DTW[i - 1][j]), DTW[i - 1][j - 1]);
        if (!flag && (j == 1 || j == m - 1)) {
            if (i + 1 == n) ++j;
            else ++i;
            dir *= -1; flag = true;
        }
        else if (!flag && (i == 1 || i == n - 1)) {
            if (j + 1 == m) ++i;
            else ++j;
            dir *= -1; flag = true;
        }
        else {
            if (i == n - 1 && j == m - 1) break;
            i = i + dir; j = j - dir; flag = false;
        }
```

```
    }

    vector<int> d;
    //Identify the warping path. (n - 1,m - 1) -> (0,0)
    i = n - 1; j = m - 1;
    while (i >= 0 && j >= 0) {
        d.push_back(DTW[i][j]);

        if (i == 0) { j = j - 1; continue; }
        else if (j == 0) { i = i - 1; continue; }

        if (DTW[i - 1][j - 1] <= DTW[i][j - 1]) {
            if (DTW[i - 1][j - 1] <= DTW[i - 1][j]) { i = i - 1; j = j - 1; }
            else i = i - 1;
        }
        else if (DTW[i][j - 1] <= DTW[i - 1][j]) j = j - 1;
        else i = i - 1;
    }

    double ans = 0;
    //Calculate th time normalized distance
    for (auto di : d) ans += di;
    return ans / d.size();
}

int main(){
    vector<int> X,Y;
    //test case 1
    X = {37,37,38,42,25,21,22,33,27,19,31,21,44,46,28};
    Y = {37,38,42,25,21,22,33,27,19,31,21,44,46,28,28};
    cout<<distance(X,Y)<<endl;
    //test case 2
    X = {11,14,15,20,19,13,12,16,18,14};
    Y = {11,17,13,14,11,20,15,14,17,14};
    cout<<distance(X,Y)<<endl;
    //Remark: when you modify the code to add the window constraint, the distance function has thus three
         inputs: X, Y, and the size of window w.
    return 0;
}
```

The result is:

```
0
8.66667
```

The time complexity of every step is:

i. Initalization is regraded as $O(1)$.

ii. Clearing the border costs $O(m+n)$, and traversing in a diagonal way costs $O(mn)$.

iii. Finding the path costs $O(m+n)$.

iv. Calculating the average costs $O(m+n)$.

So, it is of $O(mn)$ time complexity.

(b) The implementation with window constraint is as follows:

Listing 3: Code-DTWW.cpp

```
#include <iostream>
#include <vector>
#include <cmath>
#include <numeric>
/*
The process to calculate the dynamic can be divided into four steps:
1.Create an empty cost matrix DTW with X and Y labels as amplitudes of the two series to be compared.
2.Use the given state transition function to fill in the cost matrix.
3.Identify the warping path starting from top right corner of the matrix and traversing to bottom left. The
     traversal path is identified based on the neighbor with minimum value.
i.e., When we reach the point (i, j) in the matrix, the next position is to choose the point with the
     smallest cost among (i-1,j-1), (i,j-1), and (i-1,j),
```

```
For the sake of simplicity, when the cost is equal, the priority of the selection is (i-1,j-1), (i,j-1),
    and (i-1,j) in order.
4.Calculate th time normalized distance. We define it as the average cost of the selected points.
*/

#define VOID -1
using namespace std;

int minChoice(vector<vector<int>>& DTW, int& i, int& j) {
    if (DTW[i - 1][j - 1] != VOID && DTW[i - 1][j - 1] <= DTW[i][j - 1]) {
        if (DTW[i - 1][j] == VOID || DTW[i - 1][j - 1] <= DTW[i - 1][j]) { i = i - 1; j = j - 1; }
        else i = i - 1;
    }
    else if (DTW[i][j - 1] != VOID && DTW[i][j - 1] <= DTW[i - 1][j]) j = j - 1;
    else if (DTW[i - 1][j] != VOID) i = i - 1;
    else { i = i - 1; j = j - 1; }
    return DTW[i][j];
}

double distance(vector<int> x, vector<int> y, int w) {
    int n = x.size();
    int m = y.size();
    vector<vector<int>> DTW(n, vector<int>(m, VOID));
    // Use the given state transition function to fill in the cost matrix. -- diagonal calculation
    // Clear the left and bottom border first.
    DTW[0][0] = abs(x[0] - y[0]);
    for (int i = 1; i <= (w>n-1?n-1:w); ++i)
        DTW[i][0] = abs(x[i] - y[0]) + DTW[i - 1][0];
    for (int j = 1; j <= (w>m-1?m-1:w); ++j)
        DTW[0][j] = abs(x[0] - y[j]) + DTW[0][j - 1];
    //traverse only in a diagonal way
    int pi = 1, pj = 1;
    while (pi <= n - 1 && pj <= m - 1) {
        for (int i = pi, j = pj; abs(i - j) <= w && i <= n - 1 && i>=1 && j <= m - 1 && j>=1; ++i, --j) {
            int temp_i = i, temp_j = j;
            DTW[i][j] = abs(x[i] - y[j]) + minChoice(DTW, temp_i, temp_j);
        }
        if (abs(pj + 1 - pi) <= w && pj + 1 <= m - 1) ++pj;
        else ++pi;
    }

    vector<int> d;
    //Identify the warping path. (n - 1,m - 1) -> (0,0)
    int i = n - 1, j = m - 1;
    while (true) {
        d.push_back(DTW[i][j]);
        if (i == 0 || j == 0) break;
        minChoice(DTW, i, j);
    }

    double ans = 0;
    //Calculate th time normalized distance
    for (auto di : d) ans += di;
    return ans / d.size();
}

int main(){
    vector<int> X,Y;
    int w = 1;
    //test case 1
    X = {37,37,38,42,25,21,22,33,27,19,31,21,44,46,28};
    Y = {37,38,42,25,21,22,33,27,19,31,21,44,46,28,28};
    cout<<distance(X,Y,w)<<endl;
    //test case 2
    X = {11,14,15,20,19,13,12,16,18,14};
    Y = {11,17,13,14,11,20,15,14,17,14};
    cout<<distance(X,Y,w)<<endl;
    //Remark: when you modify the code to add the window constraint, the distance function has thus three
        inputs: X, Y, and the size of window w.
    return 0;
}
```

When $w = 0$, the result is:

```
52.2
18.8
```

When $w = 1$, the result is:

```
52.2
16.9
```

Note that the traverse methods in the two subproblems are slightly different, shown in Figure 3.

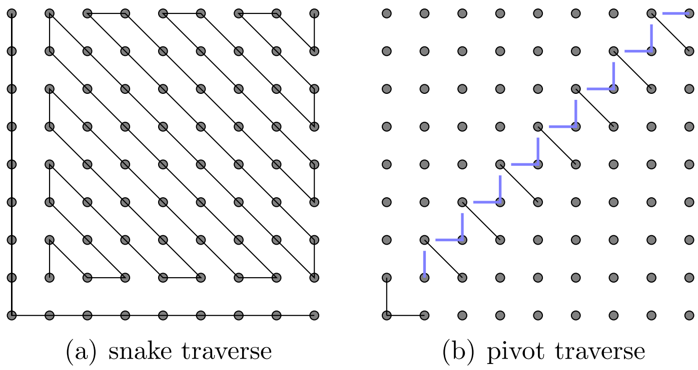

Figure 3: Different traverse methods

The first subproblem without the window constraint uses a snake approach to make turns when meeting the boundaries. The second subproblem with the window constraint uses a pivot coordinate (shown in blue line) to record the starting point on each diagonal and stop when the absolute value constraint or the border is met. □

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.