Algorithms

授課老師：張景堯

Customizing Edit Distance

Chapter.10-Dynamic Programming

Customizing Edit Distance

- ***Table Initialization*** – The function init_matrix(m) initializes the *zeroth* row and column of the dynamic programming table, respectively.
- ***Penalty Costs*** – The functions match(c,d) and indel(c) present the costs for transforming character c to d and inserting/deleting character c. For edit distance, match costs nothing if the characters are identical, and 1 otherwise, while indel always returns 1.
(計算編輯距離時，如果兩字元相同match回傳0，否則回傳1，indel則都是回傳1；但當用在別的問題上時可能會不相同就可以客製化調整，我們後面會看到)

Customizing Edit Distance(Cont.)

- *Goal Cell Identification* – The function goal_cell returns the indices of the cell marking the endpoint of the solution. For edit distance, this is defined by the length of the two input strings. (根據問題，goal_cell會傳回不同位置)
- *Traceback Actions* – The functions match_out, insert_out, and delete_out perform the appropriate actions for each edit-operation during traceback. For edit distance, this might mean printing out the name of the operation or character involved, as determined by the needs of the application. (可根據問題的需要，進行不同的結果輸出)

Substring Matching

- Suppose that we want to find where a short pattern s best occurs within a long text t, say, searching for "Skiena" in all its misspellings (Skienna, Skena, Skina, . . .).
- Plugging this search into our original edit distance function will achieve little sensitivity, since the vast majority of any edit cost will be that of deleting the body of the text.
- Likewise, the goal state is not necessarily at the end of both strings, but the cheapest place to match the entire pattern somewhere in the text.
(在字串匹配問題上，goal_cell的位置s字串還是一樣是s的長度，但t就不一定是採用整個t的長度，而是選出最匹配就是編輯距離最短的那個位置)

Dynamic Programming(Substring Matching)

```
m = [x[:] for x in [[None]*(len(t)+1)]*(len(s)+1)]


def string_compare(s:str, t:str)->int:
    opt = {}
    init_matrix(m)
    for i in range(1, len(s)+1):
        for j in range(1, len(t)+1):
            opt['MATCH'] = m[i-1][j-1].cost + (0 if s[i-1] == t[j-1] else 1)
            opt['INSERT'] = m[i][j-1].cost + 1
            opt['DELETE'] = m[i-1][j].cost + 1
            m[i][j].cost = min(opt.values())
            m[i][j].parent = min(opt, key=opt.get)
    goal = goal_cell(m,s,t)
    return m[goal[0]][goal[1]].cost
```

Initialize Matrix (Substring Matching)

- We want an edit distance search where the cost of starting the match is independent of the position in the text. (讓t的每個字元都可以跟s字串從頭比對)

```
def init_matrix(m:list):
    for i in range(len(m)):
        for j in range(len(m[0])):
            m[i][j] = Cell()
            # if i == 0: #Row init
            #     m[0][j].cost = j
            #     if j > 0:
            #         m[0][j].parent = 'INSERT'
        m[i][0].cost = i #Column init
        if i > 0:
            m[i][0].parent = 'DELETE'
```

Goal Cell(Substring Matching)

```
def goal_cell(m:list, s:str ,t:str)->list:
    goal = [len(s), 0]
    for k in range(1, len(t)+1):
        if m[goal[0]][k].cost < m[goal[0]][goal[1]].cost:
            goal[1] = k
    return goal
```

Substring Matching Main Program

```
s = 'shot'
# s = 'sport'
t = 'The movie opens with a dreamy shot of a sunset'
# t = 'He is just starting shooting his new movie'
# t = 'This short is perfect for all sorts of sport'

m = [x[:] for x in [[None]*(len(t)+1)]*(len(s)+1)]

import time
start=time.time()
print(string_compare(s,t))
print("running time =",time.time() - start)
```

Longest Common Subsequence

Chapter.10-Dynamic Programming

Longest Common Subsequence (最長公用次序列)

- There is a given String S:
  - S = "a t g a t g c a a t"
  - Substrings of S: "g a t g c", "t g c a a t".
  - Subsequences of S: "a g g t", "a a a a".
- String
  - a segment of consecutive characters. (連續字元)
  - usually called sequence in Biology.
- Sequence
  - need not be consecutive. (序列不需要連續，但順序要一樣)
- In Biology, String = Sequence.

Longest Common Subsequence (最長公用次序列)

■Common subsequences of two string X = < B, C, B, A, C > and Y = < B, D, A, B, C > :

- BC, BA, BB, BAC, BBC, ...
- The longest common subsequence (LCS) of X and Y = BAC or BBC

■[Note]

- LCS may not only one.
- By exhaustive searching, we should generate all subsequences of string X, then check if they are in string Y and get the longest one(s). If length of X is m, it will have 2^m subsequences. That is an exponential time complexity.

Modified Edit Distance Program

- The *longest common subsequence* (not substring) between "democrat" and "republican" is *eca*.
- A common subsequence is defined by all the identical character matches in an edit trace. To maximize the number of such traces, we must prevent substitution of non-identical characters. (不允許不同的字元所以把他設成最大數值)

```
opt['MATCH'] = m[i-1][j-1].cost + (0 if s[i-1] == t[j-1]
else float('inf'))
opt['INSERT'] = m[i][j-1].cost + 1
opt['DELETE'] = m[i-1][j].cost + 1
```

LCS Result

```
def LCS_result(m:list, s:str, t:str, i:int, j:int):
    if m[i][j].parent == -1:
        return
    elif m[i][j].parent == 'MATCH':
        yield from LCS_result(m, s, t, i-1, j-1)
        yield s[i-1] # match_out
    elif m[i][j].parent == 'INSERT':
        yield from LCS_result(m, s, t, i, j-1)
        yield '' # insert_out
    elif m[i][j].parent == 'DELETE':
        yield from LCS_result(m, s, t, i-1, j)
        yield '' # delete_out
```

LCS Main Program

```
s = 'democrat'
t = 'republican'
m = [x[:] for x in [[None]*(len(t)+1)]*(len(s)+1)]
import time
start=time.time()
LCS(s,t)
print("running time =",time.time() - start)
print(f"LCS('{s}','{t}') =","".join(LCS_result(m,s,t,len(s),len(t))))
```

Maximum Monotone Subsequence (最大單調次序列)

- A numerical sequence is *monotonically increasing* if the ith element is at least as big as the $(i-1)$st element.
- The *maximum monotone subsequence* problem seeks to delete the fewest number of elements from an input string S to leave a monotonically increasing subsequence.
- Thus a longest increasing subsequence of "243517698" is "23568."

Recurrence Idea for MMS

- Define l_i to be the length of the longest sequence ending with s_i of string S. The following recurrence computes l_i :
- $if\ len(S) \equiv 0, return\ 0;\ l_{0\ldots n-1} = 1$

$$l_i = \max_{0 \le j < i} l_j + 1 \text{ , where } (s_j < s_i)$$

- for $S = \text{"243517698"}$

index:	0	1	2	3	4	5	6	7	8
Sequence s_i	2	4	3	5	1	7	6	9	8
Length l_i	1	2	2	3	1	4	4	5	5
Predecessor p_i	-	0	0	1	-	3	3	5	5

- For each element s_i, we will store its predecessor : the index p_i of the element that appears immediately before s_i in the longest increasing sequence ending at s_i.

Reduction to LCS

- In fact, this is just a longest common subsequence problem, where the second string is the elements of S sorted in increasing order.
(你也可以用LCS來解，s就是S，t則是S的排序後結果)
- Any common sequence of these two must (a) represent characters in proper order in S, and (b) use only characters with increasing position in the collating sequence, so the longest one does the job.

The Ordered Partition Problem

Chapter.10-Dynamic Programming

Dividing the Work (工作分派)

- Suppose the job scanning through a shelf of books is to be split between k workers. To avoid the need to rearrange the books or separate them into piles, we can divide the shelf into k regions and assign each region to one worker.
- What is the fairest way to divide the shelf up?
(在不重排書籍位置下找出公平作法分給k個人掃描書籍)
- If each book is the same length, partition the books into equal-sized regions,

100 100 100 | 100 100 100 | 100 100 100

Dividing the Work(Cont.)

- But what if the books are not the same length? This partition would yield

  100 200 300 | 400 500 600 | 700 800 900

- Which part of the job would you volunteer to do? How can we find the fairest possible partition, i.e.

  100 200 300 400 500 | 600 700 | 800 900

The Linear Partition Problem(線性分割)

- Input: A given arrangement S of nonnegative numbers $\{s_1, \ldots, s_n\}$ and an integer k.
- Problem: Partition S into k ranges, so as to **minimize** the maximum sum over all the ranges, without reordering any of the numbers.
- Does a single fixed partition work for all instances of size (n, k) ?
- Does taking the average value of each part $(\sum_{i=1}^{n} \frac{s_i}{k})$ from the left always yield the optimal partition?
(以總和平均為基準進行分割，可行嗎?夠好嗎?)

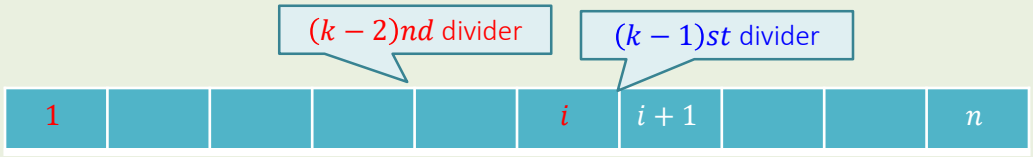
100 200 300 400 500 600 700 800 900

900 800 700 600 500 400 300 200 100

Recursive Idea

- Consider a recursive, exhaustive search approach. Notice that the kth partition starts right after we placed the $(k-1)$st divider. (第k個分割在我們放下第$k-1$個分隔板時就會產生)
- Where can we place this last divider? Between the ith and $(i+1)$st elements for some i, where $1 \leq i \leq n$.
- What is the cost of this? The total cost will be the larger of two quantities, (1) the cost of the current **last** partition $\sum_{j=i+1}^{n} s_j$ and (2) the cost of the largest partition cost formed to the **left of i**.

Recursive Idea (cont.)



- What is the size of this left partition? Use $k-2$ dividers to partition the elements $\{s_1, \ldots, s_i\}$ as equally as possible.
- *This is a smaller instance of the same problem, and hence can be solved recursively!*

(要想取得$1 \ldots n$間k個分割裡最大塊的最小值，就要從$1 \ldots i$間$k-1$個分割裡及$i+1 \ldots n$這塊中$(\forall i\colon 1 \le i \le n)$取最大塊的最小值。看出這裡面的遞迴關係了嗎？)

Dynamic Programming Recurrence

- Define $M[n,k]$ to be the minimum possible cost over all partitions of $\{s_1,\ldots,s_n\}$ into k ranges, where the cost of a partition is the largest sum of elements in one of its parts. Thus defined, this function can be evaluated:

$$M[n,k] = \min_{1\le i\le n} (\max(M[i,k-1], \sum_{j=i+1}^{n} s_j))$$

- with the natural basis cases of

$$M[1,k] = s_1, \text{for all } k > 0 \text{ and, } M[n,1] = \sum_{i=1}^{n} s_i$$

Running Time

- It is the number of cells times the running time per cell. A total of $k \cdot n$ cells exist in the table.
- Each cell depends on n others, and can be computed in linear time, for a total of $O(kn^2)$.
- Partitioning {1, 1, 1, 1, 1, 1, 1, 1, 1} into {{1, 1, 1}, {1, 1, 1}, {1, 1, 1}} (Left) ; Partitioning {1, 2, 3, 4, 5, 6, 7, 8, 9} into {{1, 2, 3, 4, 5}, {6, 7}, {8, 9}} (Right) [M for value, D for divider]

M	k		
n	1	2	3
1	3 1	4 1	1
1	2 2	3 1	1
1	1 3	2 2	1
1	4	1 2	2
1	5	3	2
1	6	3	2
1	7	4	3
1	8	4	3
1	9	5	3

D	k		
n	1	2	3
1	–	–	–
1	–	1	1
1	–	1	2
1	–	2	2
1	–	2	3
1	–	3	4
1	–	3	4
1	–	4	5
1	–	4	6

M	k		
n	1	2	3
1	9 1	14 1	1
2	7 3	12 2	2
3	4 6	9 3	3
4	10	5 6	4
5	15	9	6
6	21	11	9
7	28	15	11
8	36	21	15
9	45	24	17

D	k		
n	1	2	3
1	–	–	–
2	–	1	1
3	–	2	2
4	–	3	3
5	–	3	4
6	–	4	5
7	–	5	6
8	–	5	6
9	–	6	7

Limitations of Dynamic Programming

Chapter.10-Dynamic Programming

When can you use Dynamic Programming?

- Dynamic programming computes recurrences efficiently by storing partial results. Thus dynamic programming is efficient when there are *few* partial results to compute!
  - There are $n!$ permutations of an n-element set, so we cannot hope to store the best solution for each sub-permutation *polynomially*.
  - There are 2^n subsets of an n-element set, so we cannot hope to store the best solution for each *polynomially*.
  - But there are *only* $n(n+1)/2$ contiguous substrings of a string, so we can use it for string problems.
  - But there are *only* n possible subtrees of a rooted tree (cut edge to the root) so we can use it optimization problems on rooted trees.

When can you use Dynamic Programming? (Cont.)

- Dynamic programming works best on objects which are **linearly ordered** and **cannot be rearranged** – characters in a string, matrices in a chain, points around the boundary of a polygon, the left-to-right order of leaves in a search tree.
- Whenever your objects are ordered in a left-to-right way, you should smell dynamic programming!

The Principle of Optimality(最佳化原則)

■To use dynamic programming, the problem must observe the *principle of optimality*, that whatever the initial state is, remaining decisions must be optimal with regard the state following from the first decision.

(先前所得的部分最佳解***狀態***可以延伸下去，只需專注在求取當前每一步最佳解的做法，一步步達成整體最佳解)

- Dijkstra's algorithm works because we care about the length of the shortest path to x, not how we got there.
- Edit distance works because we care about the cheapest way edit given prefixes, not how we got here.

(像編輯距離每一格只需決定**當前狀態下**替代、插入、刪除哪個比較好，不用去管之前怎麼操作的)

Example: The Traveling Salesman Problem

- Combinatorial problems may observe this property but still use too much memory/time to be efficient.
- Let $T(i, \{1, 2, \ldots, i\})$ be the cost of the optimal tour from 1 to i that goes thru each of the other cities $\{1, 2, \ldots, i\}$,once

$$T(i, \{1, i\}) = d_{1i}$$

$$T(i, \{1,2, \ldots, i\}) = \min_{1<j<i} (T(j, \{1, 2, \ldots, j\}) + d_{ji})$$

$$TSP = \min_{i \neq 1} (T(i, \{1, 2, \ldots, i\}) + d_{i1})$$

- The table has $n2^n$ entries(one per any subset of $\{1, 2, \ldots, n\}$ and city), and it takes about n time to fill each entry.
- Hence it is $O(n^2 2^n)$ instead of $n!$

Exercises

Problems of the Day - Cutting String

- A certain string processing language allows the programmer to break a string into two pieces. Since this involves copying the old string, it costs n units of time to break a string of n characters into two pieces.
- Suppose a programmer wants to break a string into many pieces. The order in which the breaks are made can affect the total amount of time used.
- For example suppose we wish to break a 20 character string after characters 3,8, and 10:
  - If the breaks are made in left-right order, then the first break costs 20 units of time, the second break costs 17 units of time and the third break costs 12 units of time, a total of 49 steps.

Problems of the Day (Cont.)

- If the breaks are made in right-left order, the first break costs 20 units of time, the second break costs 10 units of time, and the third break costs 8 units of time, a total of only 38 steps.

■Recurrence idea:

- String length is n. S_i is the position of the i-th cut. There have total p cuts. cut 0 & $p+1$ are boundaries of string$(0, n)$. $C_{i,j}$ is the minimum cost of cutting string from i-th cut to j-th cut. Get $C_{0,p+1}$
- $C_{i,i+1} = 0$

$$C_{i,j} = S_j - S_i + \min_{i<k<j}\left(C_{i,k} + C_{k,j}\right)$$

■Q1: Finish the table C and get the minimum units of time to break the 20 character string after characters 3,8, and 10.

i \ j	0	1	2	3	4
0	0	0			
1		0	0		
2			0	0	
3				0	0
4					0

Problems of the Day - Knapsack

- The knapsack problem is as follows: given a set of integers $S = \{s_1, s_2, \ldots, s_n\}$, and a given target number T, find a subset of S which adds up exactly to T. For example, within $S = \{1, 9, 2, 10, 5\}$ there is a subset which adds up to $T = 22$ but not $T = 23$.
- Q2: Finish the following table, and answer bottom line.
- Program Exercise: Give a correct programming algorithm for knapsack that runs in $O(nT)$ time.

S \ T	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23
0	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F
1	F	T	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F
9	F	T	F	F	F	F	F	F	F	T	T	F	F	F	F	F	F	F	F	F	F	F	F	F
2	F	T	T	T	F	F	F	F	F	T	T	T	T	F	F	F	F	F	F	F	F	F	F	F
10	F	T	T	T	F	F	F	F	F	T	T	T	T											
5	F																							

Problems of the Day - MMS

- Define l_i to be the length of the longest sequence ending with s_i of string S. The following recurrence computes l_i :
- $if\ len(S) \equiv 0, return\ 0;\ l_{0\ldots n-1} = 1$

$$l_i = \max_{0 \le j < i} l_j + 1 \text{ , where } (s_j < s_i)$$

- Q3: for $S = \{1,3,6,7,9,4,10,5,6\}$

index:	0	1	2	3	4	5	6	7	8
Sequence s_i	1	3	6	7	9	4	10	5	6
Length l_i	1	2	3						
Predecessor p_i	-	0	1						

- For each element s_i, we will store its predecessor : the index p_i of the element that appears immediately before s_i in the longest increasing sequence ending at s_i.

Program Exercises (Moodle CodeRunner)

- Exercise 10 (close at 5/27 23:59)
  - Cutting String
    - *A certain string processing language allows the programmer to break a string into two pieces. Since this involves copying the old string, it costs* n *units of time to break a string of* n *characters into two pieces.*
    - Suppose a programmer wants to break a string into many pieces. Determine the ***cheapest break cost (minimum total units of time)*** to cut the string with breakpoints depicted in a list of character positions.
  - The Knapsack Problem:
    - *Given a set of integers S, and a given target number T, determine it can find a subset of S which adds up exactly to get T.*
    - *For example, within S={1, 9, 2, 10, 5} there is a subset which adds up to T=22 but not T=23.*