CS 529: Advanced Data Structures & Algorithms
Assignment 3: String Matching Algorithms & Dynamic Programming

Hunter Lawrence, Andrew Struthers

April 26, 2024

Introduction

Let s_1, s_2 be strings of length m, n respectively, and $(i, j) \in \mathbb{Z}_m \times \mathbb{Z}_n$ i.e. i, j are integers such that $0 \leq i \leq m-1, 0 \leq j \leq n-1$. We consider the `opt` algorithm as defined in algorithm opt.

Function opt(**int** i, **int** j)

Input: Indices of strings
Output: Interger cost of best string alignment

```
1 if i = m then
2 |  return 2(n - j)
3 if j = n then
4 |  return 2(m - i)
5 if s1[i] = s2[j] then
6 |  penalty ← 0
7 else
8 |  penalty ← 1
9 return min(opt (i + 1, j + 1) + penalty, opt (i + 1, j) + 2, opt (i, j + 1) + 2)
```

Runtime Complexity of `opt`

`opt(i, j)` seeks to find the penalty number for the optimal alignment of strings s_1, s_2 by exploring all possible penalty values which can be incurred by single character shifts. The results of this will allow a dynamic programming algorithm (specified below) to find out how to match the sequences according to this value.

Because opt incurs three separate recursive calls for every original call, each branch ends when either $i == n$ OR $j = m$. This results in an exponential worst case time efficiency of $O(3^n)$. Additionally, without dynamic programming, the space complexity of this algorithm is $\max(m, n)$ because the recursive tree has at most a height of the length of the longest string being analyzed.

Sequence Alignment with Dynamic Programming

While the opt algorithm directly has exponential complexity, it can be simply changed to use dynamic programming techniques, such as memoization i.e. storing results so they don't need to be calculated again, to greatly reduce the runtime. Using memoization, we can build a map of (i, j) tuples, that map to an integer result. This will allow us to look for a given (i, j) tuple before calculating the result, saving complexity whenever the algorithm produces an (i, j) tuple we already know the answer to. This will drop the runtime efficiency from exponential to $O(mn)$, which is substantially asymptotically better. Memoization will increase the space complexity to $O(mn)$.

Now that the code is optimized using dynamic programming, we can start modifying the code. We can modify this code to not only calculate the cost of the most optimal alignment, we can have it actually build the most optimal alignment. We can do this by changing minimal code from the opt function above. We will first build an $(m+1) \times (n+1)$ matrix, where we can store the result of the opt function in each row/column pair. As we are calculating the results for a given

(i, j) pair, we will also update the matrix with a few properties. Due to the depth-first nature of recursion, the result of the bottom-right most entry of our matrix will be calculated first. Then, each new cell will be the result of either the cell diagonally down-right, the cell directly below, or the cell directly to the right of the current cell. For all cells in our matrix, we will track which cell was the direct influence. This way, we will be able to traverse the matrix from $(0, 0)$ to the bottom-right most cell, following the chain of cells that directly inspired the one before. This will produce the minimum traversal, and we can build our aligned strings according to a few rules. First, we analyze the opt_dynamic algorithm to build the matrix.

Let s_1, s_2 be strings of length m, n respectively, and $(i, j) \in \mathbb{Z}_m \times \mathbb{Z}_n$ i.e. i, j are integers such that $0 \leq i \leq m-1, 0 \leq j \leq n-1$. We consider the `opt_dynamic` algorithm as defined in algorithm opt_dynamic.

Function opt_dynamic(**int** i, **int** j)

Input: Indices of strings
Output: Interger cost of best string alignment

```
1  memo ← empty_dictionary // initialize tuple-cost dictionary
2  matrix ← (m + 1) × (n + 1)matrix
3  if (i, j) ∈ memo then
4  |  return memo[(i, j)]
5  if i = m then
6  |  matrix[i, j].cost ← 2(n − j) return 2(n − j)
7  else if j = n then
8  |  matrix[i, j].cost ← 2(m − 1) return 2(m − i)
9  else
10 |  if s1[i] = s2[j] then
11 |  |  penalty ← 0
12 |  else
13 |  |  penalty ← 1
14 |  diag_traversal ←opt (i + 1, j + 1) + penalty
15 |  right_traversal ← opt (i + 1, j) + 2
16 |  down_traversal ←opt (i, j + 1) + 2
17 |  r ← min(diag_traversal, right_traversal, down_traversal)
18 |  matrix[i, j].cost ← r
19 |  if diag_traversal = min(diag_traversal, right_traversal, down_traversal) then
20 |  |  matrix[i, j].determined ← matrix[i + 1, j + 1]
21 |  |  matrix[i, j].direction ← DIAGONAL
22 |  else if right_traversal = min(diag_traversal, right_traversal, down_traversal) then
23 |  |  matrix[i, j].determined ← matrix[i + 1, j]
24 |  |  matrix[i, j].direction ← UP
25 |  else
26 |  |  matrix[i, j].determined ← matrix[i, j + 1]
27 |  |  matrix[i, j].direction ← LEFT
28 |  memo[(i, j)] ← r
29 |  return r
```

We can call opt_dynamic by `opt(0, 0)`. This will build our state matrix that will represent the optimal alignment. We will now create two new variables, representing the aligned x and y strings. To actually build the optimal alignment, we start at the bottom-right most cell, traversing up through the chain of cells that the current cell impacted, until we get to the top left most cell. If we move diagonally, we take the character in the ith row of the x sequence and add it to the aligned x string, and the character in the jth row of the y sequence gets added to the aligned y string. If we move up, the character in the ith row of the x sequence and add it to the aligned x string and we add a "-" to the aligned y string. Likewise, if we instead move left, the character in the jth row of the y sequence gets added to the aligned y string, and a "-" is added into the aligned x sequence. We build the strings from right to left, and we end with two fully built and aligned strings. The use of dynamic programming, in this case, allows us to avoid calculating elements in the $(m + 1) \times (n + 1)$ matrix multiple times. We only do $(m + 1) \cdot (n + 1)$ recursive calls, meaning that the building of the optimal alignment is $O(m * n)$ time with dynamic programming, once again a substantial improvement over the

exponential time without dynamic programming.

Finding an Optimal Alignment

We were to assume a penalty of for a mismatch between two characters and a penalty of 2 for a gap in the sequence, represented by a "-". We wanted to find the optimal alignment of the two sequences:

```
C C G G G T T A C C A
  G G A G T T C A
```

Using the dynamic programming algorthm opt_dynamic, we calculated that the optimal alignment will have a cost of 8. Running the algorithm and building the string alignment, we got the following result:

```
C C G G G T T A C C A
- G G A G T T - - C A
```

which, as we can see, has 3 gaps and 2 character mismatches. Since we assumed a penalty of 1 for a mismatch and 2 for a gap, we have $2 * 1 + 3 * 2 = 8$, so we have indeed found the optimal alignment of these two strings given our punishment criteria.

Python Implementation

```
import pprint
from dataclasses import dataclass
from enum import Enum


class Direction(Enum):
    NONE = 0
    DIAG = 1
    UP = 2
    LEFT = 3


@dataclass
class alignment_slot:
    cost: int
    i: int = 0
    j: int = 0
    determined: 'alignment_slot' = None
    direction: Direction = Direction.NONE

x_in = input("Enter string 1: ")
y_in = input("Enter string 2: ")

x = "AACAGTTACC" if len(x_in) == 0 else x_in
y = "TAAGGTCA" if len(y_in) == 0 else y_in

m = len(x)
n = len(y)

alignment_table = [[alignment_slot(0) for _ in range(n + 1)] for _ in range(m + 1)]
memo = {}

def opt(i, j) -> int:
    if (i, j) in memo:
        return memo[(i, j)]

    alignment_table[i][j].i = i
    alignment_table[i][j].j = j

    if i == m:
        alignment_table[i][j].cost = 2*(n-j)
        return 2*(n-j)
    elif j == n:
        alignment_table[i][j].cost = 2*(m-i)
        return 2*(m-i)
    else:
        penalty = 0 if x[i] == y[j] else 1

    diag = opt(i + 1, j + 1) + penalty
    inc_i = opt(i + 1, j) + 2
    inc_j = opt(i, j + 1) + 2
    result = min(diag, inc_i, inc_j)
    alignment_table[i][j].cost = result
```

```
    if diag < min(inc_i, inc_j):
        alignment_table[i][j].determined = alignment_table[i + 1][j + 1]
        alignment_table[i][j].direction = Direction.DIAG

    elif inc_i < inc_j:
        alignment_table[i][j].determined = alignment_table[i + 1][j]
        alignment_table[i][j].direction = Direction.UP

    else:
        alignment_table[i][j].determined = alignment_table[i][j + 1]
        alignment_table[i][j].direction = Direction.LEFT

    memo[(i, j)] = result
    return result


result = opt(0, 0)

slot = alignment_table[0][0]
slots = []
while slot.determined != None:
    slots.append(slot)
    slot = slot.determined
slots.append(slot)
slots.reverse()

x_p = ["" for _ in range(len(slots)-1)]
y_p = ["" for _ in range(len(slots)-1)]
index = len(slots)-2

for i in range(len(slots)-1):
    next_slot = slots[i+1]

    if next_slot.direction == Direction.DIAG:
        x_p[index] = x[next_slot.i]
        y_p[index] = y[next_slot.j]

    elif next_slot.direction == Direction.UP:
        x_p[index] = x[next_slot.i]
        y_p[index] = "-"

    else:
        x_p[index] = "-"
        y_p[index] = y[next_slot.j]

    index -= 1

print(f"\nOptimal alignment: \n\n{''.join(x_p)}")
print(f"{''.join(y_p)}")
print(f"\nCost of alignment: {result}")
```