CS627 - IP4 – Kirsten Reid – 6/11/2019

Contents

[Background: 2](#_Toc11264435)

[Analysis: 2](#_Toc11264436)

[Inputs: 3](#_Toc11264437)

[Variables: 3](#_Toc11264438)

[Outputs: 3](#_Toc11264439)

[Recurrence Relation: 3](#_Toc11264440)

[Interim solutions: 3](#_Toc11264441)

[Variations: 4](#_Toc11264442)

[Design: 4](#_Toc11264443)

[Pseudocode: 4](#_Toc11264444)

[Time Complexity: 6](#_Toc11264445)

[Conclusion: 6](#_Toc11264446)

[References: 8](#_Toc11264447)

# Background:

Given that we want to take in two black and white “images,” which are essentially two 2D arrays of integers with 1 = black and 0 = white, we need to use an algorithm to find whether or not the images are similar. The word “similar” is both a blessing and a curse: similarity means we have some room for tolerance to account for error (“threshold” in this problem), BUT it also means that we can’t just rely on a linear array positioning solution (if x = y then true else false). In a simple case, the two strings “programming” and “pprogramming” would come up with **every single position mismatched** except the first (and one of the m’s that got lucky 😊)! Instead, we need to rely on a “minimum” difference comparison approach.

Unsurprisingly, this is not a novel problem. In the computer science world, computing the differences between two strings is commonly referred to as the Levenshtein distance, which “is defined as the minimum number of edits needed to transform one string into the other, with the allowable edit operations being insertion, deletion, or substitution of a single character.” (Rosetta Code, n.d.) Alternatively we can use Needleman/Wunsch techniques (Rouchka, n.d.) This also has foundations in “approximate string matching” and “fuzzy logic string matching”, since the allowance of shifts and faults allows us to get a more **optimal** result.

Although this is a well-known problem, there still exist many solutions. As is customary in algorithms, there are brute force techniques, naïve recursive algorithm approaches, and dynamic programming. We will use the latter to develop and understand a method of finding these minimal difference techniques.

# Analysis:

To formulate a problem using dynamic programming, we must do the following:

1. Initialize (set bounds using initial state)
2. Find / establish the recurrence relation between the subproblems
3. Solve the base cases

This problem has a few unique changes:

1. We must modify for the fact that we are comparing **sets** of strings, for every row.
2. Our instructor has mercifully made each input the same number of rows, so a simple for loop will suffice.
3. We will conglomerate each row’s minimum to create an overall difference.
4. We will compare that difference to the threshold to determine an output.

In order to develop the pseudocode, I first started by actually using the example and calculating the difference matrices. This helped me visualize what I physically needed to do to set up the algorithm.

## Inputs:

X[i, j] = X[3,5] =

Int initialImageX [3][5] = { {0,0,1,1,0},{1,1,0,0,1},{0,0,1,1,1} };

Y[i,k] = Y[3][5]

Int finalimageY[3][5] = { {0,0,1,1,0},{0,1,0,0,1},{1,0,1,1,1} };

Int i = # rows = 3

Int j = int k = # of columns = 5

## Variables:

Int D[j,k] = distance matrix calculating the min difference between each xj, yk

Int minVali = minimum difference found in the bottom row (n = i – 1)

Int totalDifferencei = sum of each minVali for every row i

Int thresh = threshold of the totalDifferencei to be “similar” or “different”

## Outputs:

Return “The images are similar.”

Return “The images are different.”

## Recurrence Relation:

Di[j,k] = min {

Di[j-1,k-1] if xj=yk

Di[j-1,k-1] +1 if xj != yk

Di[j-1,k] +1 for xi,j not in Yi

Di[j,k-1] +1 for yi,k not in Xi

}

## Interim solutions:

D1[5,5] =

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| D | 0 | X1=0 | X2=0 | X3=1 | X4=1 | X5=0 |
| 0 | **0** | **0** | **0** | 0 | 0 | 0 |
| Y1=0 | 1 | **0** | **0** | 1 | 1 | 0 |
| Y2=0 | 2 | **0** | **0** | 1 | 1 | 0 |
| Y3=1 | 3 | 1 | 1 | **0** | 1 | 1 |
| Y4=1 | 4 | 2 | 2 | 1 | **0** | 2 |
| Y5=0 | 5 | 2 | 2 | 2 | 2 | **0** |

D2[5,5] =

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| D | 0 | X1=1 | X2=1 | X3=0 | X4=0 | X5=1 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| Y1=0 | 1 | **1** | **1** | 0 | 0 | 1 |
| Y2=1 | 2 | **1** | **1** | 2 | 1 | 0 |
| Y3=0 | 3 | 3 | 2 | **1** | 2 | 2 |
| Y4=0 | 4 | 4 | 4 | 2 | **1** | 3 |
| Y5=1 | 5 | 4 | 4 | 5 | 3 | **1** |

D3[5,5] =

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| D | 0 | X1=0 | X2=0 | X3=1 | X4=1 | X5=1 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| Y1=1 | **1** | **1** | **1** | 0 | 0 | 0 |
| Y2=0 | 2 | **1** | **1** | 2 | 1 | 1 |
| Y3=1 | 3 | 3 | 2 | **1** | 2 | 1 |
| Y4=1 | 4 | 4 | 4 | 2 | **1** | 2 |
| Y5=1 | 5 | 5 | 5 | 4 | 2 | **1** |

## Variations:

Because we must go through each row and conglomerate our answers, we will have an additional for loop to ensure after each set of subproblems is solved, finalized, and decided, that those results are passed into the overall linear solution.

## Design:

Now to develop the Pseudocode, we must formulate exactly how to incorporate the subproblem solutions into the larger solution loops.

Our base cases are:

Di[xj, 0] = 0 :: which means, for any string X being compared to nothing, the difference is .. nothing.

Di[0,yk] = k :: which means, for any string x, being compared to any string y of k length, the difference is the length of the comparison.

This is kind of a crazy way to think about it, but it comes down to the terminology and setup of the problem, and how we can relate to what our goal is: the **difference** between an **input**, foundational image, and any kind of second image thereof.

# Pseudocode:

// initialize

Int i = 10; // number of rows in both X and Y images

Int j = 12; // number of columns in X image

Int k = 12; // number of columns in Y image

Int thresh = 14; // arbitrary threshold value (based on 10% of 12x12 matrix 😊)

Int totalDifferencei = 0; //initialize totalDifferencei to 0; it will be combined with all minVals for every i

// Begin row loop

For (int r = 0; r<i-1; r++)

Int minVali = 0; // initialize minVali to 0; it will be added to later, needs to be reset to 0 every loop of i

// calculate difference matrix using recursion

// find the minimum value for this row, in the last row of D

// set minVali to min of last row

// increment totalDifference

Int minDifferencei[j+1][k+1]; //begin a new difference array for i

// we need one extra row and column for the initial states

For (int p = 0; int p<j; p++)

For (int t = 0; int t<k; t++)

minDifferencei [p,0] = 0; //first row is 0

minDifferencei [0, t] = t; //first column is index

// find all options

int match = minDifferencei[p-1, t-1]

int nomatch = (minDifferencei [p-1, t-1] + 1)

int xbiggery = (minDifferencei [p-1, t] + 1)

int ybiggerx = (minDifferencei [p, t-1] + 1)

// assign the minimum value to the p,t position

minDifferencei [p,t] = std::min (match, nomatch, xbiggery, ybiggerx);

// find min for the difference matrix in the last row

minVali = min(minDifferencei [j, p])

// j is the length, so we will use that last row; we will repeat for all P

totalDifferencei = totalDifferencei + minVali; //add the latest row min val

end for // loop for all x in j

end for // loop for all y in k

end for // loop for all rows

if (totalDifferenceI > thresh)

cout<< "The images are different";

else

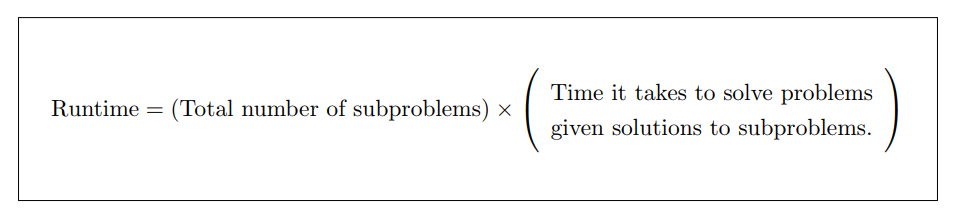
cout<<"The images are similar";

# Time Complexity:

The time complexity of this pseudocode is manifested in three different areas:

1. The i-loop repetition
2. The j,k distance matrix calculations, and
3. Utilization of memorization and subproblem solution substitution

One can use the general formula for DP (Nirkhe, 2017):



We will normalize all declarations and initializations as a constant O(1).

We have three for loops, for the rows (i), and columns (j,k). However, because we are only comparing a current solution to previous solutions already found for each, the only new cost is O(j\*k), because: “Each entry of the table requires O(1) calculations to fill given previous entries.” (Nirkhe, 2017).

Finally, we incur this matrix difference problem for each row i, so add another time complexity element of i.

Therefore, the total time worst complexity in Big O notation is: O ((j\*k) \*i)

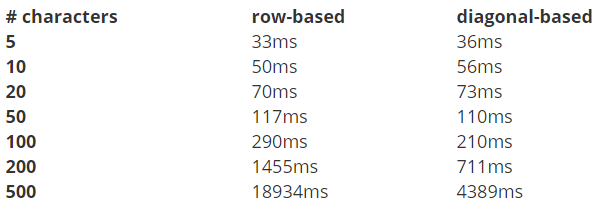
# Conclusion:

This example is a great example for Dynamic programming because of the ability to use recurrence to find minimal distances. Because we’ve already calculated previous min distances, we can logically know the best solution to the subproblems.

We saw several variations to arrive at the solution where we added an extra loop for every “string” and had an image of many “strings”; our threshold would therefore need to accommodate that, because the number of differences just by chance would increase as the size of the inputs increase. For example, a 2000x2000 pixel image with only 400 differences in pixels is less than 0.01% difference – probably really similar! But a 200x200 pixel image with 400 differences Is 1% - is that truly a similar image? IT would be circumstantial.

The time complexity was O(j\*k) but changed when we had to loop through i. Other factors that can come into play for this include an optimization where we only calculate the diagonal – as you can see from the brute force patters in the analysis, the minimum value for this problem was always in the diagonal. Referring to Musing, 2007: “The values [in a diagonal] can be calculated purely on the basis of the values [along the diagonal], which means they can be calculated in a loop rather than using recursion.”

A comparative analysis showed that as j,k grow larger, the diagonal calculations get shorter.



The trick would be to adjust for differently-sized inputs.

# References:

1. Geeks for Geeks (n.d.) *Multidimensional Arrays in C/C++.* Retrieved from <https://www.geeksforgeeks.org/multidimensional-arrays-c-cpp/>
2. Rouchka, E. (n.d.) *Dynamic Programming.* Retrieved from <http://avatar.se/lectures/molbioinfo2001/dynprog/dynamic.html>
3. Rosetta Code (n.d.) *Levenshtein Distance*. Retrieved from <http://rosettacode.org/wiki/Levenshtein_distance>
4. Tregoat, J. (2018, Jan 9) *An Introduction to Fuzzy String Matching.* Medium. Retrieved from <https://medium.com/@julientregoat/an-introduction-to-fuzzy-string-matching-178805cca2ab>
5. Jokinen, P., Tarhio, J., & Ukkonen, E. (1988, Jan) *A Comparison of Approximate String Matching*. University of Helsinki. Retrieved from <https://www.cs.hut.fi/~tarhio/papers/jtu.pdf>
6. Skiena, S. (1997, Jun 2) *Approximate String Matching*. Retrieved from <https://www8.cs.umu.se/kurser/TDBAfl/VT06/algorithms/BOOK/BOOK2/NODE46.HTM>
7. Park, J. (2015, Jun 29) *Dynamic Programming*. Stanford University. Retrieved from <https://web.stanford.edu/class/cs97si/04-dynamic-programming.pdf>
8. Le, P. (2017, Apr 16) *Course 1 – Algorithmic Toolbox – Part 4: Dynamic Programming*. Towards Data Science. Retrieved from <https://towardsdatascience.com/course-1-algorithmic-toolbox-part-4-dynamic-programming-223ffc01984a>
9. Sanfoundry (n.d.) *C++ program to find the largest and smallest elements in an array*. Retrieved from <https://www.sanfoundry.com/cpp-program-find-largest-smallest-elements-array/>
10. Nirkhe, C. (2017, May 31). *Dynamic Programming*. UC Berkley. Retrieved from <https://people.eecs.berkeley.edu/~nirkhe/cs38notes/dynamicprogramming.pdf>