Emmanuel Briones

Lab 7 Report

Introduction:

The problem that was to be solved was to implement the dynamic programming algorithm called edit distance. This algorithm checks two strings and returns the difference there is between both strings to be the same. For example, you input the string “hello” and the string “yello”. The algorithm should return 1 because there is only one letter that is different in the whole string. This is what the algorithm needs to do in a way that it is efficient and does not occupy too much memory.

Proposed Solution Design and Implementation:

First, I created a method called distance and initialized a 2d array list of size n + 1 \* n + 1. This is to align every character of one string with the character of the other string and compare them. After creating the matrix, I inserted values 0 to n – 1 into the first row and column and matrix. Then, I went ahead and compared the first two characters of the two strings. If the two characters were the same, then I would use the value diagonal to the current position. If they are not the same, then I would get the minimum value surrounding the current position and add one to it. This would be the value inserted into the current position. The algorithm then proceeds to repeat this process until the matrix is all full. When the matrix is full, then I return the last element of the matrix because that element has the value of the number of characters that are different between both strings.

Experimental Results

To test this, I decided to use 4 different test cases to see its performance. The running time of this algorithm is O(n^2) because it is traversing two strings of size n and comparing each character with each other. The algorithm was able to work as planned and here are the results of the test cases that were used:

A close up of text on a black background

Description automatically generated

Appendix:

def distance(word, word2):

matrix = [[0 for i in range(len(word) + 1)] for j in range(len(word2) + 1)]

# fill first row from 0 to n

for i in range(len(word) + 1):

matrix[0][i] = i

# fill first column from 0 to n

for i in range(len(word2) + 1):

matrix[i][0] = i

for i in range(len(word)):

for j in range(len(word2)):

# check to see if the character is the same to use the diagonal value

if word[i] == word2[j]:

matrix[i + 1][j + 1] = matrix[i][j]

# get the smallest value surrounding it and add one to it

else:

matrix[i+1][j+1] = min(matrix[i][j], matrix[i][j+1], matrix[i+1][j]) + 1

# return the last element of the matrix to obtain the final answer

num\_changes = matrix[len(word2)][len(word)]

return num\_changes

def main():

word = "super"

word2 = "duper"

print("Test Case 1")

print("Word 1: ", word)

print("Word 2: ", word2)

num\_changes = distance(word, word2)

print("Difference between words: ", num\_changes)

print()

print("Test Case 2")

word3 = "awesome"

word4 = "someone"

print("Word 3: ", word3)

print("Word 4: ", word4)

num\_changes2 = distance(word3, word4)

print("Difference between words: ", num\_changes2)

print()

print("Test Case 3")

word5 = "anywhere"

word6 = "possible"

print("Word 5: ", word5)

print("Word 6: ", word6)

num\_changes3 = distance(word5, word6)

print("Difference between words: ", num\_changes3)

print()

print("Test Case 4")

word7 = "UTEP"

word8 = "UTEP"

print("Word 7: ", word7)

print("Word 8: ", word8)

num\_changes4 = distance(word7, word8)

print("Difference between words: ", num\_changes4)

main()

Conclusions:

What I learned from this lab assignment was to know how to implement a dynamic programming algorithm. I learned how useful dynamic programming algorithms are. They are really helpful and efficient to use. I was able to implement the edit distance algorithm and see the benefits it brings. This dynamic programming algorithm really facilitates the job it needs to be done. It does not require as much effort as other algorithms.