Wagner-Fischer algorithm for calculating Levenshtein distance

Aim : To find the minimum edit distance for converting source string to the target string following the dynamic programming approach given the cost of insertion as 1, cost of deletion as 1 and cost of substitution as 2.

Input : i) A source string
ii) A target string
Eg : source string = "intention"
target string = "execution"

Output : i) Minimum edit distance for converting source string to the target string.
ii) Number of insertions, deletions and substitutions performed.
iii) Actual operations performed.
Eg : Minimum edit distance = 8
Number of insertions = 1
Number of deletions = 1
Number of substitutions = 3
Total number of operations = 5
Actual operations : i) delete 'i'
ii) substitute 'n' by 'e'
iii) substitute 't' by 'x'
iv) insert 'c'
v) substitute 'n' by 'u'

Algorithm :

- Function MIN_EDIT_DISTANCE (source, target) returns minimum_edit_distance
- m is the length of the source string
- n is the length of the target string

1) Create a matrix 'distance' of size (n+1, m+1) whose each element represents a distance value.

2) Initialize the zeroth row and column to be the distance from the empty string
distance[0,0] = 0
for each column i for 1 to n do
distance[i,0] = distance[i-1,0] + insertion_cost(target[i])
for each column i for 1 to m do
distance[0,i] = distance[0,i-1] + deletion_cost(source[i])

3) for each column i for 1 to n do
for each row j for 1 to m do
if source[j-1] is equal to target[i-1] then
dp[i,j] = dp[i-1,j-1]
else :
dp[i,j] = min(dp[i-1,j] + insertion_cost, dp[i-1,j-1]+ substitution_cost,
dp[i,j-1] + deletion_cost)

4) Backtrack to record the operations performed
set i=n+1 and j=m+1

```
while i and j both > 0 do
        if source[j-1] is equal to target[i-1] then
                i = i – 1
                j = j – 1
        else :
                if dp[i,j] is equal to (dp[i-1, j-1] + substitution_cost) then
                        add the substitution operation to the list_of_operations
                        i = i – 1
                        j = j – 1
                else if dp[i,j] is equal to (dp[i-1, j] + insertion_cost) then
                        add the insertion operation to the list_of_operations
                        i = i – 1
                else :
                        add the deletion operation to the list_of_operations
                        j = j – 1

while j is not equal to 0 do
        add the deletion operation to the list_of_operations
        j = j – 1

while i is not equal to 0 do
        add the insertion operation to the list_of_operations
        i = i – 1
```

5) Reverse the list of operations [as they are added in reverse order because of backtracking]
   return (distance[n+1,m+1], list_of_operations)

More details :

The minimum edit distance problem has the two properties : overlapping subproblems and optimal substructure. Thus dynamic programming approach can be applied for solving this problem. It takes into consideration all possible combinations of the source string and target string. The algorithm goes step-by-step, considers each combination and calculate the minimum edit distance for it. The result of this subproblem is then used to find the solution of the next following combination and thus we reach to the final solution.

Time complexity for the algorithm = O(m * n)
Space complexity for the algorithm = O(m * n)
where, m is the length of the source string
        n is the length of the target string