Algorithms & Data Structures II (course 1DL231)
Uppsala University – Autumn 2020
Report for Assignment 2 by Team 23

William EULAU Adam ERIKSSON

26th November 2020

Part 1
Search-String Replacement

A Introduction

In this part we will present an algorithm, heavily influence by the Levenshtein Distance algorithm, that given two strings u and r is able to compute the cost (score) of matching these strings with each other, using insertion of dashes in both strings. The cost may vary depending on criteria for cost computation and may in this example be based on distances between characters on a QWERTY keyboard or a constant penalty of 1 for non-aligned character. Nonetheless, the algorithm is not bounded to a sort cost matrix. In this part, we will present our implementation in the sections, one presenting the dynamic programming matrix build and returning the minimum cost of alignment, and the other also returning the two strings aligned using a matrix traversal. We will also present a time complexity analysis of the algorithm, which we will argue is $\mathcal{O}(|u| \cdot |r|)$.

B Dynamic Programming Implementation

The algorithm will have three objects inputted. Two of these are strings which minimal score of alignment is to be computed. The third input is a resemblance matrix R which for each combination of two letters returns a number corresponding to the cost of matching those characters. In order to present the minimal alignment cost — minimal distance — a two-dimensional dynamic programming matrix M (in our code called dp_matrix) is initiated. The two axis of the matrix represent every character of the inputted strings and a dash. The matrix will be filled in a bottom-up method. As a first step of filling the matrix, the base cases are stated. Position M_{00} , corresponding to alignment of two empty strings is given the value 0. This is due to the operation having no cost. Each position on the first row M_{1j} and column M_{i1} is then filled in accordance to the resemblance matrix R and taking to account the already accumulated score for the positions calculated earlier (for M_{1j}, the value to the right and for M_{i1}, the value above). With the base cases filled in, the matrix is completed using a nested for loop (in time $O(|u| \cdot |r|)$). The loop iterates through all remaining positions and fills each one by comparing the value to its left, above, and diagonally above to the left. The minimum value is chosen and the R-score corresponding to the two characters compared is added. This step of the algorithm is represented in the recursive formula below.

$$M_{ij} = min \begin{cases} M[i-1][j-1] + R[u_{i-1}][r_{j-1}] \\ M[i-1][j] + R[u_{i-1}][-] \\ M[i][j-1] + R[-][r_{j-1}] \end{cases}$$

With the full matrix filled, the bottom right value is collected and returned. This value corresponds to the minimal distance score of aligning the two strings. An example of a matrix M can be seen in Table 1.

As presented above, the dynamic programming approach of this algorithm will provide a matrix with the dimensions corresponding to the inputted strings (with addition to one row and column of dashes). We can derive from this structure that the algorithm indeed has an optimal substructure. The full scale problem to present the cost of aligning two stings of full size can be derived in steps answering sub problems of the cost of only aligning parts of each string. These sub problems are all represented in the matrix. By answering each sub problem with a bottom-up approach, we can use the answers already obtained from previous sub problems to decrease the computational cost since there is a dependence between the problems.

To reiterate, the score presented at M_{ij} corresponds to the cost of aligning the first i-1 characters of string u and the first j-1 characters of string r. If we then increase the observed values to either u, r, or both, we may use the already obtained information as a sub problem only adding potential cost. Hence, the algorithm has optimal substructure.

	-	D	I	N	A	M	C	K
-	0	1	2	3	4	5	6	7
D	1	0	1	2	3	4	5	6
Y	2	1	1	2	3	4	5	6
N	3	2	2	1	2	3	4	5
A	4	3	3	2	1	2	3	4
M	5	4	4	3	2	1	2	3
I	6	5	5	4	3	2	2	3
C	7	6	6	5	4	3	2	3

Table 1: A visualization of a matrix M computing the alignment cost of the two strings $DINAMCK$ and $DYNAMIC$. The matrix R implemented here states that the cost of all miss alignments is 1 and an alignment is 0. The bottom right value (colored blue) represents the minimal cost of alignment. The light gray cells represents the base cases.

```
    dp_matrix = [
        [None for i in range(len(r) + 1)] for j in range(len(u) + 1)
    ]

    dp_matrix = [
        [None for i in range(len(r) + 1)] for j in range(len(u) + 1)
    ]

    # Fill dp_matrix in bottom up manner
    # matrix fill nested for-loops, one running through 0,1,2,...,u and the other /
        through 0,1,2,...,r
    # The loops will terminate after (u+1)*(r+1) iterations, under O(|u*r|) time.
    for i in range(len(u) + 1):
        for j in range(len(r) + 1):
            #initialization cases
```

```
15          # [0][0] = 0
16          if i == 0 and j == 0:
17              dp_matrix[i][j] = 0
18
19          # Filling first row dp[0][column]
20          elif i == 0 and j != 0:
21              dp_matrix[i][j] = dp_matrix[i][j - 1] + R['-'][r[j - 1]]
22
23          # Filling first row dp[row][0]
24          elif i != 0 and j == 0:
25              dp_matrix[i][j] = dp_matrix[i - 1][j] + R[u[i - 1]]['-']
26
27          # Other cases, compare the value above, below and upper left diagonally.
28          # Add potential penalty (score) and assert the smallest value into the /
                position.
29          else:
30              dp_matrix[i][j] = min(dp_matrix[i - 1][j - 1] + R[u[i - 1]][r[j - 1]],
31                                    dp_matrix[i - 1][j] + R[u[i - 1]]['-'],
32                                    dp_matrix[i][j - 1] + R['-'][r[j - 1]])
33
34      return dp_matrix[len(u)][len(r)] # Return Score
```

Listing 1: Obtaining the minimal score of alignmnet using a dynamic programming implementation influenced by the Lavenshtein Distance algorithm.

C Returning Alignment

When the matrix is completed, analogously to the matrix trace-back from **Assignment 1**, a reverse walk of the matrix will produce the re-aligned strings of the inputs. Starting from the bottom right corner the matrix is traversed by one of three actions: moving left, moving up or moving diagonally. A traversal to the left implies an added dash, -, to the second input string and a continuation of the first input string *(corresponding to a removal)*. A traversal upward corresponds to an insertion, adding a dash to the first input string. A diagonal travels means at that index both characters are considered. The traversal is determined through identifying which traversal (up, left or diagonal) has the minimum landing index character pair cost and matrix cost.

The traversal is complete when the first index, [0][0], is obtained. If the traversal lands on the 0:th row or column, the remaining traversals will go straight toward index [0][0], repeating either left or up movements. As the matrix is traversed backwards, after the traversal is complete, both strings need to reversed.

The time complexity of the matrix traversal is $O(|u| \cdot |r|)$, and will not alter the time complexity, which we will argue for in the following section.

```
1   u_edited_flipped = ""
2   r_edited_flipped = ""
3   #score = dp_matrix[len(u)][len(r)]
4   u_index = len(u)
5   r_index = len(r)
6   #logger = logging.getLogger("ACTIONS")
7   #while loop terminating when u_index == 0 and r_index == 0
8   #In every if / else statement either r_index or u_index is decreased
9   #when one of the variables is zero, the other one is decreased, until /
        termination
10  #time complexity O(|len(r)*len(u)|)
11  while u_index != 0 or r_index != 0:
12      if u_index == 0:
```

```
            #go up
            r_edited_flipped += r[r_index-1]
            u_edited_flipped += '-'
            r_index -= 1
        elif r_index == 0:
            #go_left
            r_edited_flipped += '-'
            u_edited_flipped += u[u_index-1]
            u_index -= 1
        else:
            diag_score = dp_matrix[u_index-1][r_index-1] + R[u[u_index - 2]][r[r_index - 2]]
            left_score = dp_matrix[u_index][r_index-1] + R[u[u_index - 1]][r[r_index - 2]]
            up_score = dp_matrix[u_index-1][r_index] + R[u[u_index - 2]][r[r_index - 1]]

            scores = [diag_score,left_score,up_score]
            action = scores.index(min(scores))

            if action == 0:
                #diag
                u_edited_flipped += u[u_index-1]
                r_edited_flipped += r[r_index-1]
                r_index -= 1
                u_index -= 1
            elif action == 1:
                #left
                r_edited_flipped += r[r_index-1]
                u_edited_flipped += '-'
                r_index -= 1
            else:
                #up
                r_edited_flipped += '-'
                u_edited_flipped += u[u_index-1]
                u_index -= 1

        #logger.debug(f"u = {u_edited_flipped[::-1]}, r = {r_edited_flipped[::-1]}, \n action = {action}, u index = {u_index}, r index = {r_index} ")
    u_out = u_edited_flipped[::-1]
    r_out = r_edited_flipped[::-1]
    return (dp_matrix[len(u)][len(r)], u_out, r_out)
```

Listing 2: The string alignment through the dynamic programming matrix created in previous sections.

D Time Complexity Analysis

The most time consuming step in this dynamic programming algorithm implementation is the cost of filling out the dynamic programming matrix, which is done in $\mathcal{O}(|u| \cdot |r|)$ corresponding to the dimensions of the matrix, as the operations filling the matrix don't further increase the time complexity upper bound. As described in the section above, the alignment string return is bounded by $\mathcal{O}(|u| \cdot |r|)$. The score of string alignment is obtained in constant time $\mathcal{O}(1)$. In sum, due to the cost of filling the score matrix being the most time consuming part of the algorithm, the overall time complexity will be $\mathcal{O}(|u| \cdot |r|)$.

Part 2

Recomputing a Minimum Spanning Tree

A Introduction

Given a connected, weighted, undirected graph $G = (V, E)$ with non-negative edge weights and a Minimum Spanning Tree (MST), $T = (V, E')$, of G, updating an edge-weight, from $w(e)$ to $\widehat{w}(e)$ in G, requires updating the MST T to ensure the minimum spanning properties. Depending on which edge is updated, the MST update will take different time complexity. In this part, we will argue for the time complexity of the four possible edge-weight updates listed below. We will also provide an implementation of the fourth possible edge-weight update.

1. $e \notin E'$ and $\widehat{w}(e) > w(e)$
2. $e \notin E'$ and $\widehat{w}(e) < w(e)$
3. $e \in E'$ and $\widehat{w}(e) > w(e)$
4. $e \in E'$ and $\widehat{w}(e) < w(e)$

B General and Example Reasoning

The general reasoning for the four different edge-weight increments will be supplemented by an example graph, presented in Figure 1. The MST T is highlighted with thick, black edges.

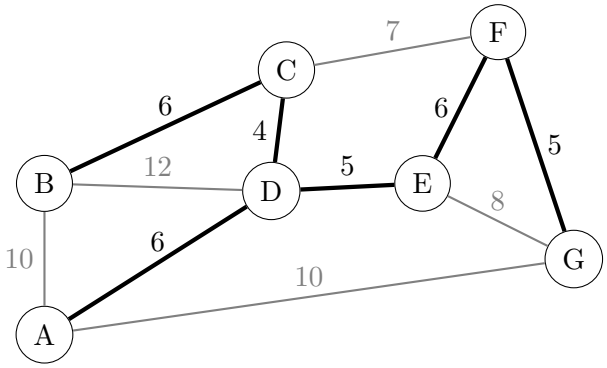

Figure 1: Minimum spanning tree in an example graph.

When the edge weight is updated according to items 1. and 3., we argue that updating T takes time $\mathcal{O}(1)$ with the reasoning as follows. Updating an edge not in the MST to a larger value (1.) will not test the minimum spanning properties of T. This is because an edge already too large to be a part of the MST does not pose a challenge to any of the edges in E'; on the contrary, it is less likely to be part of the MST. The same reasoning follows with item 3.; an edge already part of the MST further secures it's position in E' with a reduced edge-weight. In Figure 2, items 1. and 3. are proven using an example, where the edge-weight between E and F is reduced to 1 and the edge-weight between B and D is incremented to 36. The MST does not need to be altered for the properties to still hold. Therefore, items 1. and 3. require a MST update of time complexity $\mathcal{O}(1)$.

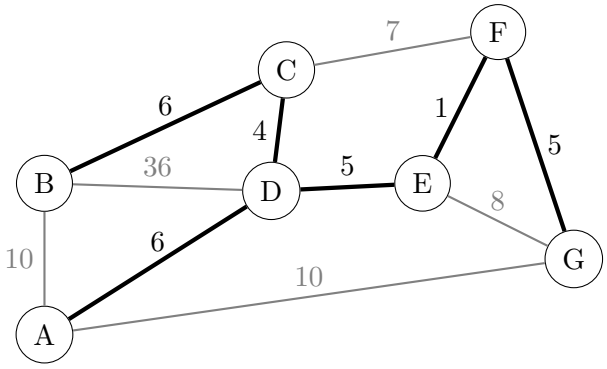

Figure 2: Minimum spanning tree preserved with edge-weight updates according to items 1. and 3.

For item 2., a more time-complex update is required. When an edge not in E' is decreased, the minimum spanning properties are not definite. The updated edge outside of the MST is the only candidate for being part of the MST, as the others have the same, already tested, values. Per the definition of MSTs, adding an edge will create a cycle in T. By omitting any edge from the cycle in T, a new tree with minimum spanning properties can be produced. Therefore, the edge with updated weight $\widehat{w}(e)$ is added to the MST, and cycle, *or ring structure*, is created. As showed in **Assignment 1**, using a modified Depth-First Search (DFS) algorithm, a ring structure can be found in time complexity $\mathcal{O}(|V|)$. When all edges in the ring structure are identified, the edge with largest weight can be omitted, and the MST is updated. Omission is not more complex than the ring structure search, which means that the MST can be updated in time $\mathcal{O}(|V|)$.

An example of item 2. is presented in Figure 3, where the edge from A to G has a reduced edge-weight $\widehat{w}(e) = 4$. The figure shows the cycle $A - D - E - F - G - A$ in the graph, and the largest weight will be omitted. In this case either from A to D or from E to F. Both omissions obtain a correct MST.

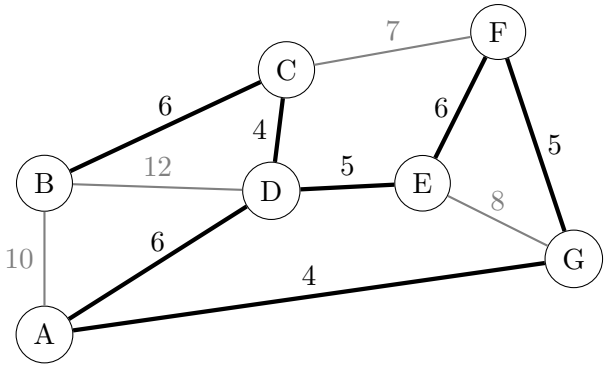

Figure 3: Minimum spanning tree with added edge from A to G. A cycle is created in the graph and the edge with largest weight in the cycle will be omitted to restore the MST property.

The fourth, and final, item 4. motivates a MST update of time $\mathcal{O}(|E|)$. When an edge in E' is incremented to a larger weight-value, it is no longer certain that the edge belongs in the MST of G. Therefore, the edge should be removed from E', creating two subsets from the cut starting from the with at least one connecting edge *(the removed one)*. The two subsetes have minimal properties, as their edges are not altered. To obtain a complete MST, the minimal weight connecting edge between the subsets needs to be identified. Using the edges E' in T,

the subsets can be identified by a modified DFS algorithm traceback from the vertices in the removed edge of E'. Then, all edges E in G are considered. If the vertices in spanning the edge are from different subsets, the edge is a connecting edge. The connecting edge with lowest weight is the edge fulfilling all MST properties when added to the MST. This algorithm will be described in a more technical manner in the following section.

An example of item 4. is presented in Figure 4, where the edge from D to E has a incremented edge-weight $\widehat{w}(e)$ to value 8. The identified connected edges are shown in cyan. In this example, the edge from C to F is added to the MST, creating a new and full MST.

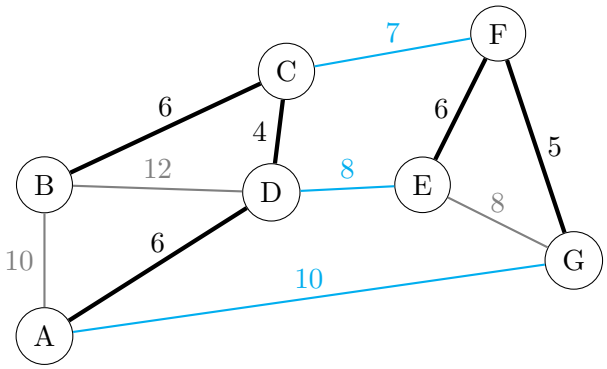

Figure 4: Minimum spanning tree with incremented weight if edge from D to E. Two cuts of the graph are present, $D-A-C-B$ and $E-F-G$. The connecting edges are colored in cyan. The new MST will contain the edge from C to F.

C Code Implementation

The code implementation of item 4. naturally reflects the reasoning above. When the edge-weight from vertex u to v is incremented, it is updated in E and removed from E'. Then, the two cuts are identified. In code, this is accomplished by a modified DFS-algoorithm, identifying connected vertices through computing edges in a Last In, First Out stack (LIFO). When the stack is empty, all vertices in the cut are identified. The two cuts' vertices are identified in a dictionary data-frame through strings 'U' and 'V'. The two loops in the code run on $\mathcal{O}(|E'|)$ time.

```
    (u, v) = e
    assert(e in G and e in T and weight > G.weight(u, v))
    #update edge weight in G and remove edge from T.
    G.set_weight(u,v,weight)
    T.remove_edge(u,v)
    #identify two cuts in one dictionary
    cuts = dict()
    parent = dict() #intialize parent dictionary
    parent[u] = None
    parent[v] = None
    #run through the two cuts, depth-first walking using a stack, starting from u /
        and v for both stacks
    #As the two while loops, containing pop and append operations, depend on an /
        empty stack to terminate
    #it could theoretically never terminate. However, if the input tree T is a MST /
        of graph G,
    #according to MST properties, there is a finite amount of edges E' which span /
        the tree.
```

```
15    #The two loops will combined run through all edges (using te built-in neighbor /
          function)
16    #With the parent dictionary and MST properties, there will never be a cycle in /
          the edges.
17    #Therefore, the two while loops in 100-122 will have time complexity O(|E'|),
18    #tighter upper bound than O(|E|)
19    #cut 'U'
20    stack = [u]
21    while len(stack) > 0:
22       node = stack.pop()
23       cuts[node] = 'U'
24       neighbors = T.neighbors(node)
25       for neighbor in neighbors:
26          if(parent[node] != neighbor): #if neighbor isn't parent
27             parent[neighbor] = node
28             stack.append(neighbor)
29    #cut 'V'
30    stack = [v]
31    while len(stack) > 0:
32       node = stack.pop()
33       cuts[node] = 'V'
34       neighbors = T.neighbors(node)
35       for neighbor in neighbors:
36          if(parent[node] != neighbor): #if neighbor isn't parent
37             parent[neighbor] = node
38             stack.append(neighbor)
```

Listing 3: Identifying the two cuts after updated edge is removed from T, labelling them either as U or V.

When the vertices are grouped in the dictionary data frame, the connecting edges between the cuts need to be identified. All edges of G, E, are considered. If the edge connects vertices from different cuts, then it is a connecting edge. As we search for the minimal spanning tree, only the edge with minimum weight is saved. In code, this is accomplished by a for-loop through all edges. When all edges have been considered, the minimum connecting edge will be updated in T.

```
1
2     #walk through all edges in G O(|E|)
3     min_connecting_edge_weight = None
4     min_connecting_edge = None
5     #For-loop iterating through all edges in O(|E|) time.
6     #No actions in the loop will increase the upper bound.
7     for edge in G.edges:
8        (a,b) = edge
9        if cuts[a] != cuts[b]: #if nodes are in different cuts
10          weight = G.weight(a,b) #connecting edge
11          #connecting_edges[weight] = edge
12          if min_connecting_edge_weight is None:
13             min_connecting_edge_weight = weight
```

Listing 4: Identifying the two cuts after updated edge is removed from T, labelling them either as U or V.

Visible in the code implementation is that the recomputation of the MST will have time complexity $\mathcal{O}(|E|)$. The two while-loops will together run in time $\mathcal{O}(|E'|)$, as they loop through all edges (except for the deleted edge) in E'. The for-loop runs in $\mathcal{O}(|E|)$, which is a looser upper-bound than $\mathcal{O}(|E'|)$. As there are no more loops or computations to consider, the whole algorithm will run on time $\mathcal{O}(|E|)$.

Part 3
Code Appendix

A Search-String Replacement

```
from typing import * # noqa
import unittest # noqa
from src.difference_data import data # noqa
import math # noqa
from collections import defaultdict # noqa
from string import ascii_lowercase # noqa
# If your solution needs a queue, then you can use this one:
from collections import deque # noqa
# If you need to log information during tests, execution, or both,
# then you can use this library:
# Basic example:
# logger = logging.getLogger("put name here")
# a = 5
# logger.debug(f"a = {a}")
import logging # noqa


def min_difference(u: str, r: str, R: Dict[str, Dict[str, int]]) -> int:
    """
    Sig: str, str, Dict[str, Dict[str, int]] -> int
    Pre: For all characters c in u and k in r,
         then R[c][k] exists, and R[k][c] exists.
    Post:
    Ex: Let R be the resemblance matrix where every change and skip
         costs 1
         min_difference("dinamck", "dynamic", R) --> 3
    """
    # To get the resemblance between two letters, use code like this:
    # difference = R['a']['b']
    dp_matrix = [
        [None for i in range(len(r) + 1)] for j in range(len(u) + 1)
    ]

    dp_matrix = [
        [None for i in range(len(r) + 1)] for j in range(len(u) + 1)
    ]

    # Fill dp_matrix in bottom up manner
    # matrix fill nested for-loops, one running through 0,1,2,...,u and the other /
        through 0,1,2,...,r
    # The loops will terminate after (u+1)*(r+1) iterations, under O(|u*r|) time.
    for i in range(len(u) + 1):
        for j in range(len(r) + 1):
            #initialization cases
            # [0][0] = 0
            if i == 0 and j == 0:
                dp_matrix[i][j] = 0

            # Filling first row dp[0][column]
            elif i == 0 and j != 0:
                dp_matrix[i][j] = dp_matrix[i][j - 1] + R['-'][r[j - 1]]

            # Filling first row dp[row][0]
```

```
53          elif i != 0 and j == 0:
54             dp_matrix[i][j] = dp_matrix[i - 1][j] + R[u[i - 1]]['-']
55
56          # Other cases, compare the value above, below and upper left diagonally.
57          # Add potential penalty (score) and assert the smallest value into the /
                position.
58          else:
59             dp_matrix[i][j] = min(dp_matrix[i - 1][j - 1] + R[u[i - 1]][r[j - 1]],
60                                   dp_matrix[i - 1][j] + R[u[i - 1]]['-'],
61                                   dp_matrix[i][j - 1] + R['-'][r[j - 1]])
62
63    return dp_matrix[len(u)][len(r)] # Return Score
64
65
66
67 # Solution to Task C:
68 def min_difference_align(u: str, r: str,
69                          R: Dict[str, Dict[str, int]]) -> Tuple[int, str, str]:
70    """
71    Sig: str, str, Dict[str, Dict[str, int]] -> Tuple[int, str, str]
72    Pre: For all characters c in u and k in r,
73         then R[c][k] exists, and R[k][c] exists.
74    Post:
75    Ex: Let R be the resemblance matrix where every change and skip
76         costs 1
77         min_difference_align("dinamck", "dynamic", R) -->
78                               3, "dinam-ck", "dynamic-"
79    """
80    dp_matrix = [
81       [None for i in range(len(r) + 1)] for j in range(len(u) + 1)
82    ]
83
84    # Fill dp_matrix in bottom up manner
85    # matrix fill nested for-loops, one running through 0,1,2,...,u and the other /
          through 0,1,2,...,r
86    # The loops will terminate after (u+1)*(r+1) iterations, under O(|u*r|) time.
87    for i in range(len(u) + 1):
88       for j in range(len(r) + 1):
89          #initialization cases
90          # [0][0] = 0
91          if i == 0 and j == 0:
92             dp_matrix[i][j] = 0
93
94          # Filling first row dp[0][column]
95          elif i == 0 and j != 0:
96             dp_matrix[i][j] = dp_matrix[i][j - 1] + R['-'][r[j - 1]]
97
98          # Filling first row dp[row][0]
99          elif i != 0 and j == 0:
100            dp_matrix[i][j] = dp_matrix[i - 1][j] + R[u[i - 1]]['-']
101
102         # Other cases, compare the value above, below and upper left diagonally.
103         # Add potential penalty (score) and assert the smallest value into the /
                position.
104         else:
105            dp_matrix[i][j] = min(dp_matrix[i - 1][j - 1] + R[u[i - 1]][r[j - 1]],
106                                  dp_matrix[i - 1][j] + R[u[i - 1]]['-'],
107                                  dp_matrix[i][j - 1] + R['-'][r[j - 1]])
108
109   #dp_matrix created. Now backward traceback.
```

```
    u_edited_flipped = ""
    r_edited_flipped = ""
    #score = dp_matrix[len(u)][len(r)]
    u_index = len(u)
    r_index = len(r)
    #logger = logging.getLogger("ACTIONS")
    #while loop terminating when u_index == 0 and r_index == 0
    #In every if / else statement either r_index or u_index is decreased
    #when one of the variables is zero, the other one is decreased, until /
        termination
    #time complexity O(|len(r)*len(u)|)
    while u_index != 0 or r_index != 0:
        if u_index == 0:
            #go up
            r_edited_flipped += r[r_index-1]
            u_edited_flipped += '-'
            r_index -= 1
        elif r_index == 0:
            #go_left
            r_edited_flipped += '-'
            u_edited_flipped += u[u_index-1]
            u_index -= 1
        else:
            diag_score = dp_matrix[u_index-1][r_index-1] + R[u[u_index - /
                2]][r[r_index - 2]]
            left_score = dp_matrix[u_index][r_index-1] + R[u[u_index - 1]][r[r_index /
                - 2]]
            up_score = dp_matrix[u_index-1][r_index] + R[u[u_index - 2]][r[r_index - /
                1]]

            scores = [diag_score,left_score,up_score]
            action = scores.index(min(scores))

            if action == 0:
                #diag
                u_edited_flipped += u[u_index-1]
                r_edited_flipped += r[r_index-1]
                r_index -= 1
                u_index -= 1
            elif action == 1:
                #left
                r_edited_flipped += r[r_index-1]
                u_edited_flipped += '-'
                r_index -= 1
            else:
                #up
                r_edited_flipped += '-'
                u_edited_flipped += u[u_index-1]
                u_index -= 1

        #logger.debug(f"u = {u_edited_flipped[::-1]}, r = {r_edited_flipped[::-1]}, /
            \n action = {action}, u index = {u_index}, r index = {r_index} ")
    u_out = u_edited_flipped[::-1]
    r_out = r_edited_flipped[::-1]
    return (dp_matrix[len(u)][len(r)], u_out, r_out)
```

Listing 5: Full code supplied in difference.py.

B Recomputing a MST

```
from typing import * # noqa
import unittest # noqa
from src.recompute_mst_data import data # noqa
from src.graph import Graph # noqa
# If your solution needs a queue, then you can use this one:
from collections import deque # noqa
# If you need to log information during tests, execution, or both,
# then you can use this library:
# Basic example:
# logger = logging.getLogger("put name here")
# a = 5
# logger.debug(f"a = {a}")
import logging # noqa

__all__ = ['update_MST_1', 'update_MST_2', 'update_MST_3', 'update_MST_4']

def update_MST_4(G: Graph, T: Graph, e: Tuple[str, str], weight: int):
    """
    Sig: Graph G(V, E), Graph T(V, E), edge e, int ->
    Pre:
    Post:
    Ex: TestCase 4 below
    """
    (u, v) = e
    assert(e in G and e in T and weight > G.weight(u, v))
    #update edge weight in G and remove edge from T.
    G.set_weight(u,v,weight)
    T.remove_edge(u,v)
    #identify two cuts in one dictionary
    cuts = dict()
    parent = dict() #intialize parent dictionary
    parent[u] = None
    parent[v] = None
    #run through the two cuts, depth-first walking using a stack, starting from u /
        and v for both stacks
    #As the two while loops, containing pop and append operations, depend on an /
        empty stack to terminate
    #it could theoretically never terminate. However, if the input tree T is a MST /
        of graph G,
    #according to MST properties, there is a finite amount of edges E' which span /
        the tree.
    #The two loops will combined run through all edges (using te built-in neighbor /
        function)
    #With the parent dictionary and MST properties, there will never be a cycle in /
        the edges.
    #Therefore, the two while loops in 100-122 will have time complexity O(|E'|),
    #tighter upper bound than O(|E|)
    #cut 'U'
    stack = [u]
    while len(stack) > 0:
        node = stack.pop()
        cuts[node] = 'U'
        neighbors = T.neighbors(node)
        for neighbor in neighbors:
            if(parent[node] != neighbor): #if neighbor isn't parent
                parent[neighbor] = node
                stack.append(neighbor)
    #cut 'V'
```

```
52    stack = [v]
53    while len(stack) > 0:
54       node = stack.pop()
55       cuts[node] = 'V'
56       neighbors = T.neighbors(node)
57       for neighbor in neighbors:
58          if(parent[node] != neighbor): #if neighbor isn't parent
59             parent[neighbor] = node
60             stack.append(neighbor)
61
62    #walk through all edges in G O(|E|)
63    min_connecting_edge_weight = None
64    min_connecting_edge = None
65    #For-loop iterating through all edges in O(|E|) time.
66    #No actions in the loop will increase the upper bound.
67    for edge in G.edges:
68       (a,b) = edge
69       if cuts[a] != cuts[b]: #if nodes are in different cuts
70          weight = G.weight(a,b) #connecting edge
71          #connecting_edges[weight] = edge
72          if min_connecting_edge_weight is None:
73             min_connecting_edge_weight = weight
74             min_connecting_edge = edge
75          elif weight < min_connecting_edge_weight:
76             min_connecting_edge_weight = weight
77             min_connecting_edge = edge
78
79    #min_connecting_edge is the new edge in the MST
80    (u,v) = min_connecting_edge
81    weight = min_connecting_edge_weight
82    T.add_edge(u, v, weight)
```

Listing 6: Full code supplied in recompute_mst.py.