Algorithms & Data Structures II (course 1DL231)
Uppsala University – Autumn 2021
Report for Assignment 2 by Team 15

Jakob Nordgren Aljaz Kovac

29th November 2021

Part 1

Search-String Replacement

A Recursive Equation

Let i, j denote possible indices of characters in the strings u and r, respectively, and u_i is the ith character of u. If for example, $u = "abc"$, then $i \in \{1, 2, 3\}$ and $u_1 = "a"$ and so on. Then we have the following recursive equation:

$$OPT(i,j) = \begin{cases} R[-,-] & if\ i = 0,\ j = 0 \\ OPT(i-1,j) + R[u_i,-] & if\ i > 0, j = 0 \\ OPT(i,j-1) + R[-,r_j] & if\ i = 0, j > 0 \\ min \begin{cases} OPT(i-1,j) & +R[u_i,-] \\ OPT(i,j-1) & +R[-,r_j] \\ OPT(i-1,j-1) & +R[u_i,r_j] \end{cases} & otherwise \end{cases}$$

Intuitively, if we want to align two strings u and r, then for any pair of characters (u_i, r_j) there are three cases for aligning the substrings $u' = u_0...u_i$ and $r' = r_1...r_j$:

1. u' and r' are aligned by first aligning "u'_1 ... u'_{i-1}" with "$r'_1...r_j$", and then skipping the character u'_i by appending a "-" to r'_{j+1}.
2. u' and r' are aligned by first aligning "r'_1 ... r'_{j-1}" with "$u'_1...u_i$", and then skipping the character r'_j by appending a "-" to u'_{i+1}.
3. u' and r' are aligned by first aligning "u'_1 ... u'_{i-1}" with "$r'_1...r_{j-1}$", and then replacing the character u'_i with r'_j

The optimal solution at any step will therefore be either of the three cases. We now have defined the optimal solution in strictly smaller subproblems, so this solution does indeed have the optimal substructure property. Further, we can see that for computing $OPT(i, j)$, if $i > 0, j > 0$, then we must also compute $OPT(i-1, j)$, $OPT(i, j-1)$ and $OPT(i-1, j-1)$. But when computing $OPT(i-1, j)$ we will also compute $OPT(i-1, j-1)$, and when computing $OPT(i, j-1)$ we must also compute $OPT(i-1, j-1)$. It's clear that we have overlapping subproblems, so dynamic programming should be a suitable approach here.

B Bottom-Up Iteration and Top-Down Recursion

In part A, we defined an optimal solution $OPT(i.j)$ as a recursive equation. This can be implemented with the same time complexity as either bottom-up iteration or top-down recursion. We chose to use bottom-up iteration by populating a 2-dimensional table with solutions for increasingly larger subproblems. For example, if $u = "xyz"$ and $r = "abc"$, and $R[c_1, c_2] = 1$ for all $c_1, c_2 \in A \cup \{-\}$ then we compute the values of the table, starting from the bottom of the leftmost column:

c	3			
b	2	...		
a	1	2		
$-$	0	1		
	-	x	y	z

Intuitively, to compute the cost for the substrings corresponding to entries i, j, the algorithm looks at the neighboring entries $(i-1, j)$, $(i, j-1)$, $(i-1, j-1)$. Please refer to listing 1 for the implemented algorithm.

C Extending the algorithm

The extended algorithm is based on the idea of recursively traversing the matrix that was built using the algorithm described in parts A and B, and finding the path that was taken from the result (given in the last entry of the matrix M, so at position $M[length(u)][length(r)]$ where u and r are the two given strings and u is positioned on the row side, while r is positioned on the column side). We traverse the matrix, checking at every recursive call whether the current entry in the matrix originated from the entry to the left, below, or the entry diagonally left to it. If the entry originated from the left then we insert a dash into the substring r at the current position. If the entry originated from below then we insert a dash into the substring u at the current position. If the entry originated from the entry diagonal to it we simply skip that position. Please refer to listing 2 for the implemented algorithm.

D Time complexity analysis

The algorithm for determining the minimum difference of two given strings will perform a fixed amount of $\mathcal{O}(1)$ operations for each combination of (u', r'), where u' and r' are any possible prefixes of the strings u and r. Therefore its time complexity is $\mathcal{O}(|u| * |r|)$. Determining the positioning for the minimum difference will take at most $|u| + |r|$ steps because the algorithm needs to traverse the matrix R from $R[|u|][|r|]$ down to $R[0][0]$, which in the worst case equals the length of the two sides of the matrix, or $|u| + |r|$. This operation is linear, and of a lower degree than the operation that populates the matrix. Therefore the time complexity for the extended algorithm is the same as for the algorithm that populates the matrix:
$\mathcal{O}(|u| * |r| + |u| + |r|) = \mathcal{O}(|u| * |r|)$

Part 2
Recomputing a Minimum Spanning Tree

A Description of algorithm

A.1 $e \notin E'$ and $\hat{w}(e) > w(e)$

The updated edge is not in the MST E' and the new value $\hat{w}(e)$ is greater than the previous value $w(e)$. Then E' is still the MST so

$$update_{MST}(G, T, e, w) = T$$

which is $\mathcal{O}(1)$.

A.2 $e \notin E'$ and $\hat{w}(e) < w(e)$

The updated edge is not in the MST E' and the new value $\hat{w}(e)$ is smaller than the previous value $w(e)$. Then e could be part of the MST. To update the MST, create a new tree

$T' = (V, E' \cup \{e\})$. Since a MST contains the minimal set of edges that connect all vertices, T' will now contain a cycle. By the cycle property of minimum spanning trees, we know that the most expensive edge $\bar{e}$ in this cycle cannot be a part of the MST. The algorithm therefore is:

1. Define a new tree $T' = (V, E' \cup \{e\})$. By definition, we are given V, E' and e, respectively so this step is $\mathcal{O}(1)$.
2. Do a depth-first search(DFS) on T', if a node $v \in V$ is visited twice in any path, we have found a cycle. Store all edges in that cycle. A DFS will iterate over all edges in the graph and perform some amount of constant work for each edge so time complexity is $\mathcal{O}(|E|)$
3. Let $\bar{e}$ be the edge of greatest weight in the cycle
4. The new MST is now $T'' = (V, (E' \cup \{e\}) \setminus \{\bar{e}\})$

Step 1 and 4 is done in constant time, since we are given V, E' and e in step 1 and also have already computed $\bar{e}$ in step 4. Step 2 is bounded by the complexity of a DFS, where we will do some constant amount of work for each edge in the MST so this is $\mathcal{O}(|E'|)$. Step 3 involves examining the edges of the cycle, but the number of edges here is of course never greater than $|E'|$. Now, by definition of a MST, it only has 1 edge for each vertex in the MST so total time complexity for the case is $\mathcal{O}(|V|)$.

A.3 $e \in E'$ and $\hat{w}(e) < w(e)$

The updated edge is in the MST E' and the new value $\hat{w}(e)$ is smaller than the previous value $w(e)$. Then E' is still the MST so

$$update_{MST}(G, T, e, w) = T$$

which, again, is $\mathcal{O}(1)$.

A.4 $e \in E'$ and $\hat{w}(e) > w(e)$

The updated edge is in the MST E' and the new value $\hat{w}(e)$ is greater than the previous value $w(e)$. Then E' may or may not still be the MST. Remove $e = (u, v)$ from E', so that the the graph now consist of two disjoint sets, say $T_1 = (V_1, E'_1)$ and $T_2 = (V_2, E'_2)$. Calculate which nodes belong to each of the component by doing a DFS from u and v respectively. Now go through all edges in the graph, and find the minimal weighted edge

$$\begin{aligned} \bar{e} = (\bar{u}, \bar{v}) \in (E \setminus E')\ &s.t. \\ &\bar{u} \in V_1 \\ \wedge\ &\bar{v} \in V_2 \\ \wedge\ &\forall \hat{e} = (\hat{u} \in V_1, \hat{v} \in V_2) : \hat{w}(\bar{e}) \leq \hat{w}(\hat{e}). \end{aligned}$$

That is, the minimal weighted edge that connects T_1 and T_2. The new MST is then

$$T'' = (V, (E' \setminus \{e\}) \cup \{\bar{e}\})$$

Here, we iterate through all edges in the graph so time complexity is $\mathcal{O}(|E|)$

B Implementation

We have implemented the algorithm for the last case (A.4), shown in listing 3. The solution is based on a bottom-up iterative implementation of the algorithm described in the previous section. The depth-first search is however implemented as a top-down recursive nested function.

```python
# Solution to Task B:
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: Return the minimal cost of changing string u into string r according to
          the given resemblance matrix
    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']

    # List comprehension loop Variant: len(r)-len(u)-(iteration_number)
    # The u string goes on the x-axis, r goes on the y-axis
    dp_matrix = [
        [None for _ in range(len(r)+1)] for _ in range(len(u)+1)
        ]
    dp_matrix[0][0] = R['-']['-']
    for x in range(1,len(u)+1):
        # Variant: len(u)-x
        dp_matrix[x][0] = dp_matrix[x-1][0] + R[u[x-1]]['-']
    for y in range(1,len(r)+1):
        # Variant: len(r)-y
        dp_matrix[0][y] = dp_matrix[0][y-1] + R['-'][r[y-1]]


    # Populate the matrix
    for x in range(1, len(u)+1):
        # Variant: len(u)-x
        for y in range(1, len(r)+1):
            # Variant: len(r)-y
            dp_matrix[x][y] = min(
                dp_matrix[x-1][y] + R[u[x-1]]['-'],
                dp_matrix[x][y-1] + R['-'][r[y-1]],
                dp_matrix[x-1][y-1] + R[u[x-1]][r[y-1]])

    # Read the result
    result = dp_matrix[len(u)][len(r)]
    return result
```
Listing 1: The algorithm that returns the minimum difference between two strings

```python
def align(x, y, u, r):
   '''
   Sig: int, int, str, str -> Tuple[str, str]
   Pre: dp_matrix[i][j] contains the minimum difference for strings u[i-1] and /
       r[j-1]
         On the first(non-recursive) call, x=len(u) andn y=len(r)
   Post: Returns strings r and u with inserted '-' characters, indicating the /
       minimal
         cost alignment of the strings.
   Ex: (Let R map all changes and skips to value 1)
      align(len("dinamic"), len("dynamck"), "dynamic", "dinamck") -> /
          "dinam-ck", "dynamic-"

   Variant: x+y
   Invariant: At each call the tails of each string, u[x:] and r[y:], are /
       aligned
   '''
   if x == 0 and y == 0:
      return u, r

   leftVal = dp_matrix[x-1][y] + R[u[x-1]]['-']
   underVal = dp_matrix[x][y-1] + R['-'][r[y-1]]
   diagonalVal = dp_matrix[x-1][y-1] + R[u[x-1]][r[y-1]]

   if leftVal < underVal and leftVal < diagonalVal:
      return align(x-1, y, u, r[:y] + '-' + r[y:])
   elif underVal < diagonalVal:
      return align(x, y-1, u[:x] + '-' + u[x:], r)
   else:
      return align(x-1, y-1, u, r)

alignedU, alignedR = align(len(u), len(r), u, r)

return result, alignedU, alignedR
```

Listing 2: The algorithm that returns the minimal cost of changing string u into string r according to the given resemblance matrix, as well as the aligned strings

```
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: 'T' is the minimal spanning tree of 'G' before updating edge 'e' with /
       weight 'weight'
   Post: 'T' is the minimal spanning tree of 'G' after updating edge 'e' with /
       weight 'weight'
   Ex: TestCase 4 below
   """
   (u, v) = e
   assert(e in G and e in T and weight > G.weight(u, v))

   def dfs(v, visited):
      # Simple DFS that adds reachable nodes to a set
      # Recursion Variant: (Number of vertices in connected component of v) - /
          len(visited)
      visited.add(v)
      for neighbor in T.neighbors(v):
         # Loop Variant: len(T.neighbors(v)) - (index of 'neighbor' in /
             T.neighbors(v))
         if neighbor not in visited:
            dfs(neighbor, visited)

   # Update G with the new weight, and remove the edge from the MST
   G.set_weight(u,v,weight)
   T.remove_edge(u,v)

   # t1 and t2 are the disjoint connected components
   t1 = set()
   t2 = set()

   # Variables to keep track of candidates for the new minimal edge
   minU = None
   minV = None
   minWeight = float('INF')

   # Populate sets
   dfs(u, t1)
   dfs(v, t2)

   for gu, gv in G.edges:
      # Loop Variant: (nubmer of edges in G) - (iteration number)
      if (gu in t1 and gv in t2) or (gu in t2 and gv in t1):
         if G.weight(gu,gv) < minWeight:
               minU, minV = gu, gv
               minWeight = G.weight(gu, gv)
   T.add_edge(minU, minV, minWeight)
```

Listing 3: The algorithm that recomputes the minimum spanning tree for the following case: the updated edge is in the MST E' and the new value $\hat{w}(\bar{e})$ is greater than the previous value $w(e)$