Uppsala Universitet

Algorithms and Data Structures II (1DL231), 2017

Assignment 2

Författare:
Team 13
Daniel Ågstrand
Linnea Andersson

UPPSALA
UNIVERSITET

7. december 2017

Problem 1: Search string replacement

This problem has as goal to create a program that will calculate the minimum cost of changing one string to a specific target string. The program should take in a string, a target string and a matrix that has the information about the cost of changing one character to an other and then it should return the minimum cost to get the target string from your start string. For example: $min\_difference(string, targetstring, cost - matrix) -> 3$

Another function that returns the correct alignment and the minimum cost should also be implemented. This function should take in the same input as the function above, one string, one target string and a matrix with change costs. $min\_difference\_align(string, targetstring, cost - matrix) -> 3, alignstring$

a.) Recursive equation

The recursive equation becomes:

$$md(u, r, R) = \begin{cases} \sum_{i=1}^{|u|} R[u[i]]['-'] & |r| = 0 \\ \sum_{j=1}^{|r|} R['-'][r[j]] & |u| = 0 \\ \min(md(u, r-1, R) + R['-'][r[j]], md(u-1, r, R) + R[u[i]]['-'], & \\ md(u-1, r-1, R) + R[u[i]][r[j]]) & otherwise \end{cases}$$

In the recursive equation one can see that for every iteration we have to check three possible alterations to the strings, namely insertion into u, deletion in u or substituting a character in u for one in r. The call to $md(u-1, r-1, R)$ signifies the minimum difference of changing the all but the last character in u into all but the last character in r. If we first call $md(u, r-1, R)$ and in the next recursion call $md(u-1, r, R)$, we get a recursion call to $md(u-1, r-1, R)$. Thus we have overlapping subproblems and can use a dynamic programming matrix to avoid doing the same recursion call twice.

b.) Code implementation for minimum cost

```
1  def min_difference(u,r,R):
2      """
3      Sig:      string, string, int[0../A/, 0../A/] ==> int
4      Pre:      None
5      Post:     Return the minimum change cost of u and r
6      Example: Let R be the resemblance matrix where every
7                    change and skip costs 1
8               min_difference("dinamck","dynamic",R) ==> 3
9      """
10     global lenu
11     global lenr
12     if len(u) < len(r):
13         for i in range(len(r)-len(u)):
14             #Variant: (len(r)-len(u)) - i
15             u = '-' + u
16     if len(r) < len(u):
17         for i in range(len(u)-len(r)):
18         #Variant: (len(u)-len(r)) - i
19             r = '-' + r
20     lenu = len(u)
21     lenr = len(r)
22     u = list(u)
23     r = list(r)
24     # Initialize the dynamic programming matrix, A
25     # Type: Int[0..(lenr+1)][0..(lenu+1)]
26     global A
27     A = [[0 for i in range(lenr+1)] for j in range(lenu+1)]
28     # Initialize backtrace matrix, V
29     # Type: string[0..(lenr+1)][0..(lenu+1)]
30     global V
31     V = [[0 for i in range(lenr+1)] for j in range(lenu+1)]
32     A[lenu][lenr] = min_difference_aux(u,lenu,r,lenr,R)
33     return A[lenu][lenr]
```

Listing 1: Code for minimum difference

```
def min_difference_aux(u,lenu,r,lenr,R):
    A[0][0] = 0
    for i in range(1,lenu+1):
        #Variant: (lenu+1) - i
        A[i][0] = A[i-1][0] + R[u[i-1]]['-']
        V[i][0] = 'up'
    for j in range(1,lenr+1):
        #Variant: (lenr+1) - j
        A[0][j] = A[0][j-1] + R['-'][r[j-1]]
        V[0][j] = 'left'
    for i in range(1,lenu+1):
        #Variant: (lenu+1) - i
        for j in range(1,lenr+1):
                #Variant: (lenr+1) - j
                A[i][j] = min(A[i-1][j] + R[u[i-1]]['-'], A[i][j-1] + R
                    ['-'][r[j-1]], A[i-1][j-1] + R[u[i-1]][r[j-1]])

                if A[i-1][j-1] + R[u[i-1]][r[j-1]] <= min(A[i-1][j] + R
                    [u[i-1]]['-'], A[i][j-1] + R['-'][r[j-1]]):
                    V[i][j] = 'diag'

                elif A[i-1][j] + R[u[i-1]]['-'] <= min(A[i][j-1] + R['-
                    '][r[j-1]], A[i-1][j-1] + R[u[i-1]][r[j-1]]):
                    V[i][j] = 'up'

                else:
                    V[i][j] = 'left'

    return A[lenu][lenr]
```

Listing 2: Code for minimum difference help function

c.) Code implementation for minimum cost and alignment

```
def min_difference_align(u,r,R):
    """
    Sig:     string, string, int[0../A/, 0../A/] ==> int, string, string
    Pre:     None
    Post:    Return the minimum change cost of u and r and the correct
        alignment
    Example: Let R be the resemblance matrix where every change and
        skip costs 1
             min_min_difference_align("dinamck","dynamic",R) ==>
                                        3, "dinam-ck", "dynamic-"
    """
    diff = min_difference(u,r,R)
    if len(u) < len(r):
        for i in range(len(r)-len(u)):
            #Variant: (len(r)-len(u)) - i
            u = '-' + u
    if len(r) < len(u):
        for i in range(len(u)-len(r)):
            #Variant: (len(u)-len(r)) - i
            r = '-' + r
    i = lenu-1
    j = lenr-1
    u = list(u)
    r = list(r)
    while j != 0 or i != 0:
        #Variant: lenu-1,lenr-1 - i,j
        if i == 0:
            u.insert(i,'-')
            j -= 1

        elif j == 0:
            r.insert(j,'-')
            i -= 1

        elif V[i+1][j+1] == 'diag':
            if R[u[i]][r[j]] > R['-'][r[j]] + R[u[i]]['-']:
                r.insert(j+1,'-')
                u.insert(i,'-')
            j -= 1
            i -= 1

        elif V[i+1][j+1] == 'up':
            r.insert(j+1,'-')
            i -= 1
```

```
43
44          else:
45              u.insert(i+1,'-')
46              j -= 1
47
48      i = 0
49      while u[i] == '-' and r[i] == '-':
50          #Variant: u[i] == '-' and r[i] == '-' - i
51          u = u[1:]
52          r = r[1:]
53          i += 1
54      u = ''.join(u)
55      r = ''.join(r)
56
57      return (diff,u,r)
```

Listing 3: Code for minimum cost and alignment

d.) Complexity of the program

```
1       if len(u) < len(r):
2           for i in range(len(r)-len(u)):
3               #Variant: (len(r)-len(u)) - i
4               u = '-' + u
5       if len(r) < len(u):
6           for i in range(len(u)-len(r)):
7           #Variant: (len(u)-len(r)) - i
8               r = '-' + r
```

This part has complexity $O(|r|)$ or $O(|u|)$. In our calculation we will use the notation $O(|r|)\ ||\ O(|u|)$ for this.

```
1       for i in range(1,lenu+1):
2           #Variant: (lenu+1) - i
3           A[i][0] = A[i-1][0] + R[u[i-1]]['-']
4           V[i][0] = 'up'
5       for j in range(1,lenr+1):
6           #Variant: (lenr+1) - j
7           A[0][j] = A[0][j-1] + R['-'][r[j-1]]
8           V[0][j] = 'left'
```

This part has complexity $O(|r|) + O(|u|)$

```
1       for i in range(1,lenu+1):
2           #Variant: (lenu+1) - i
3           for j in range(1,lenr+1):
4                   #Variant: (lenr+1) - j
```

This has complexity $O(|r| * |u|)$

```
1      if len(u) < len(r):
2          for i in range(len(r)-len(u)):
3              #Variant: (len(r)-len(u)) - i
4              u = '-' + u
5      if len(r) < len(u):
6          for i in range(len(u)-len(r)):
7              #Variant: (len(u)-len(r)) - i
8              r = '-' + r
```

And this part has complexity $O(|r|)$ or $O(|u|)$.

By adding all the different complexities, you will get the complexity of *min_difference*. So the complexity is $(O(|r|) \mid\mid O(|u|)) + O(|r|) + O(|u|) + O(|r| * |u|)$, which can be simplified to $O(|r| * |u|)$.

min_difference_align has a function call to *min_difference* in it. So we add the complexity of *min_difference* to the complexity of modifying the strings. The complexity of modification is:

```
1      while j != 0 or i != 0:
2          #Variant: lenu-1,lenr-1 - i,j
3          if i == 0:
4              u.insert(i,'-')
5              j -= 1
6
7          elif j == 0:
8              r.insert(j,'-')
9              i -= 1
10
11         elif V[i+1][j+1] == 'diag':
12             if R[u[i]][r[j]] > R['-'][r[j]] + R[u[i]]['-']:
13                 r.insert(j+1,'-')
14                 u.insert(i,'-')
15             j -= 1
16             i -= 1
17
18         elif V[i+1][j+1] == 'up':
19             r.insert(j+1,'-')
20             i -= 1
21
22         else:
23             u.insert(i+1,'-')
24             j -= 1
```

This part has complexity $O(|r|) \mid\mid O(|u|)$.

```
1        while u[i] == '-' and r[i] == '-':
2            #Variant: u[i] == '-' and r[i] == '-' - i
3            u = u[1:]
4            r = r[1:]
5            i += 1
```

And this part has complexity $O(|r|)$ || $O(|u|)$).

So the full complexity of $min\_difference\_align$ is $3(O(|r|)$ || $O(|u|))+O(|r|)+O(|u|)+O(|r|*|u|)$, which can also be simplified to O(|r|*|u|).

Problem 2: Ring Detection in Graphs

This problem has two goals. The first goal is to create a program that will take in a graph and return a boolean, true or false. True if the graph contains a circle. The second goal is to make a program that take in a graph and returns a boolean and also a list with the nodes that builds the circle.

We solved this by implementing a version of depth first search. First, all nodes in the graph are colored white. When a node is discovered, it is colored red. When a node is finished, meaning it has no more neighbor nodes to explore, it is colored blue. Whenever a red node is discovered, it means we have been there before and thus we have a circle.

a.) Code implementation of the ring detector

```
1  def ring(G):
2      """
3      Sig:  graph G(node,edge) ==> boolean
4      Pre:  None
5      Post: Returns true if there is a ring in the graph
6      Example:
7          ring(g1) ==> False
8          ring(g2) ==> True
9      """
10     global nodes
11     global cycle
12     global ringlist
13     global done
14     # Initialize the return list for ring_extended
15     # Type: nodes[]
16     ringlist = []
17     cycle = False
18     done = False
19     nx.set_node_attributes(G, 'color','white')
20     nodes = nx.get_node_attributes(G, 'color')
21     for n in G:
22         #Variant: G - n
23         if nodes[n] is 'white':
24             nodes[n] = 'red'
25             DFS_visit(n,G,n)
26             if cycle is True:
27                 return True
28         nodes[n] = 'blue'
29     return False
```

Listing 4: Code for ring detector

```
def DFS_visit(u,G,previous):
    """
    Sig:  node u, graph G(node,edge), node previous ==> boolean
    Pre:  Node can't be none
    Post: Returns true if a node has been vistited before
    Example:
        DFS_visit(node u1, graph G1) ==> False
        DFS_visit(node u2, graph G2) ==> True
    """
    global cycle
    global done
    nodes[u] = 'red'
    for j in G.neighbors(u):
        #Variant: G.neighbors(u) - j
        if cycle is True:
            break
        if nodes[j] is 'white':
            nodes[j] = 'red'
            DFS_visit(j,G,u)
        elif nodes[j] is 'red' and j != previous:
            cycle = True
    if cycle is True and done is False:
        ringlist.append(u)
        if u in G.neighbors(ringlist[0]) and len(ringlist) > 2:
            done = True
            ringlist.append(ringlist[0])
    if cycle is False:
        nodes[u] = 'blue'
```

Listing 5: Code for depth first search algorithm

b.) Code implementation of the extended ring detector

```
1  def ring_extended(G):
2      """
3      Sig:  graph G(node,edge) ==> boolean, int[0..j-1]
4      Pre:  None
5      Post: Returns true and a list with the nodes in the ring if there
6            is a ring in the graph, otherwise false and an empty list
7      Example:
8          ring(g1) ==> False, []
9          ring(g2) ==>  True, [3,7,8,6,3]
10     """
11     if ring(G) is False:
12         return False, []
13     else:
14         return True, ringlist
```

Listing 6: Code for extended ring detector

c.) Complexity of the ring detector

The time complexity of a correct implemented depth first search has the time complexity of O(|V|+|E|), where |V| is the amount of nodes in the graph and |E| is the amount of the edges in the graph. This is because depth first search should find out the structure of the graph, including all edges between nodes. But because we are only interested in finding circles in the graph, we are not interested in finding all the edges. We only follow one path to each node, and therefore the complexity does not depend on the number of edges. We are visiting nodes until a circle is discovered, at which point we stop exploring nodes. If a circle is not found, the program will discover all nodes and then terminate. Since each node is visited a maximum of one time, our complexity is O(|V|).

Intellectual Property

We certify that our report and all its uploaded attachments were produced solely by our team, except where explicitly stated otherwise and clearly referenced, that each teammate can individually explain any part starting from the moment of submitting our report, and that our report and attachments are not and will not be freely accessible on a public repository.