Algorithms & Data Structures II (course 1DL231)
Uppsala University – Autumn 2017
Report for Assignment 2 by Team 60

Mona Mohamed Elamin Nevine Gouda

5th December 2017

1 Search String Replacement

Given two strings u and r, of possibly different lengths over the alphabet letters, where neither string can contain "-", and a resemblance matrix R, based on the Manhattan's distance for the qwerty keyboard, the minimum difference between both sub-strings should be calculated. Differences here can be either changing a character in the u string to become like the string r, or skipping a character in either u or r.

1.1 Recursive Equation

This problem can be approached recursively as stated in Equations 1 and 2. And based on that; we can easily conclude that the problem is broken down to smaller problems. However many of these smaller problems are redundant and are solved more than once, thus implementing the algorithm using dynamic programming for this problem is much more suffice.

$$min\_diff(u_i, r_j, R) = \begin{cases} R['-']['-'] & \text{if } |u| = |r| = 0 \\ min\_diff(u_{\text{i-1}}, r_j, R) + R[u[i-1]]['-'] & \text{if } |r| = 0 \\ min\_diff(u_i, r_{\text{j-1}}, R) + R['-'][r[j-1]] & \text{if } |u| = 0 \\ min\_other() & \text{Otherwise} \end{cases} \tag{1}$$

$$min\_other()^1 = \textbf{min} \begin{cases} min\_diff(u_{\text{i-1}}, r_j, R) + R[u[i-1]]['-'], \\ min\_diff(u_i, r_{\text{j-1}}, R) + R['-'][r[j-1]], \\ min\_diff(u_{\text{i-1}}, r_{\text{j-1}}, R) + R[u[i-1]][r[j-1]] \end{cases} \tag{2}$$

1.2 Dynamic Programming Implementation

In this section we attempted to solve the Search String Replacement problem using dynamic programming. Where our attempt can be found in Listing 1.

1.3 Difference's Align Implementation

While in this section we attempted to return an updated version of both strings u and r, with - in their appropriate positions and thus both returned strings should be of equal length. Which in return shows the differences that should be made in u to become r. Our implementation can be found in Listing 3.2.

[1] This is separated to another function only for readability.

1.4 Complexity

1. Time estimated to fill the difference array with 0s::

$$\begin{aligned} T &= (|u|+1)*(|r|+1) \\ &= |u|*|r|+|u|+|r|+1 \end{aligned} \tag{3}$$

2. Time estimated to update the values of the difference array:

$$\begin{aligned} T &= (|u|+1)*(|r|+1) \\ &= |u|*|r|+|u|+|r|+1 \end{aligned} \tag{4}$$

3. Time estimated to align the two strings:

$$T = |u|+|r| \tag{5}$$

4. Therefore the total time for the algorithm:

$$\begin{aligned} T &= 2*|u|*|r|+3*(|u|+|r|)+2 \\ &<= 2*|u|*|r| \\ &= O(|u|*|r|) \end{aligned} \tag{6}$$

2 Ring Detection in Graphs

2.1 Ring implementation

Given an un-directed graph, a path $< v_0, v_1, ...v_k >$ can be considered as a ring, or a cycle, if the start vertex v_0 and end vertex v_k are the same vertex. Thus the algorithm listed in Listing 3 attempts to detect if a ring/cycle exist in a given Graph and returns True if one exists, or returns False otherwise.
The algorithm does this using DFS (Depth First Search) which is used to visit all nodes. when the node visited, it is marked in the visited array. through these nodes visits the cycle is determined if the current node has been visited but by another path.

2.2 Ring's Extended implementation

Whereas if we want to retrieve the path for said cycle then we need to extend the code to return the path if it exits, otherwise it should return False and an empty list, indicating that no ring exists in the given graph. Our attempted algorithm can be found in Listings 4, 5.
To implement it we have a node_parent list which holds the path from the root to the current node's parent. Where the algorithm then incrementally updates the node's path, which will propagate to its children. And a ring will be detected using the previous implementation. Then we return the path from the root till the node that detected the ring. Then finally from this path we extract the ring by sub-string the returned path. That is simply done by searching for this pattern: $< v_0, v_1, v_2, ...v_0 >$.

2.3 Complexity

1. Time to fill the visited nodes:
$$T = |V| \tag{7}$$

2. Time to fill the node_parents:
$$T = |V| \tag{8}$$

3. Where for ring_detection_extended_Function: the maximum number of calls to the recursion function is the number of nodes. That is because it checks all the nodes to be visited. In addition, all the other operations in this functions takes constant time. Therefore the the time for this function:
$$T = |V| + c \tag{9}$$

4. Therefore the total time for the algorithm:
$$\begin{aligned} T &= 3|V| + c \\ &= O(|V|) \end{aligned} \tag{10}$$

3 Recomputing a Minimum Spanning Tree

3.1 Updating Minimum Spanning Tree Algorithms

Given a connected, weighted, un-directed graph $G = (V, E)$ with non-negative edge weights, and a minimum spanning tree $T = (V, E')$ of G. If we want to update the weight of an edge e, where $e \in E$, then we have four possible cases, as stated below, and each case is followed by an efficient algorithm on updating the Minimum Spanning Tree.

1. $e \notin E'$ and $\hat{w}(e) > w(e)$:
The algorithm should do nothing. Why? Because since the edge is not in the spanning tree with its old value then the new larger value will not be included ,Because if we try to add the edge e to the Spanning Tree T so that it creates a sub-graph with exactly 1 cycle, we can conclude that the added edge will be the longest edge in that cycle, whether with the new weight $\hat{w}(e)$ or even the old weight $w(e)$. Thus we can conclude that increasing the weight to an already useless edge to the tree should not affect the Minimum Spanning Tree T in any way. Therefore the Minimum Spanning Tree T after the edge's update should be the same as before the update.Therefore we can conclude that it takes Constant time.

2. $e \notin E'$ and $\hat{w}(e) < w(e)$:
Since the edge is not a part of the Minimum Spanning Tree but the edge's weight is decreased, we can conclude that with this update, there may be another Minimum Spanning Tree T with a lower overall weight that contains the updated edge e.
Therefore the algorithm should do the following:
- Let $e = (u, v)$ where e is the edge to be updated between the vertices u and v.
- It will add that edge to the Minimum Spanning Tree T, which will in return create a cycle in the Tree. And since we have a cycle in a Minimum Spanning Tree we can note the following rule:

> "For any cycle C in the graph, if the weight of an edge e of C is larger than the individual weights of all other edges of C, then this edge cannot belong to a MST."[2]

[2]Obtained from: https://en.wikipedia.org/wiki/Minimum_spanning_tree#Cycle_property

- Based on the cycle property previously mentioned, the algorithm should find the edge with the highest weight in the cycle created and remove it using the Depth-First Search Algorithm. Which in return remove the cycle created. And the edge removed can either be the edge we initially inserted which would mean that the Tree before and after updating would be exactly the same. Or it could be another edge, which would mean that updating the weight actually resulted in a better Minimum Spanning Tree T. And since in the worst case, the ring is formed with all the nodes, we will have compute the minimum edge between them, and since this is a tree not a graph, thus we conclude that it depends on the number of vertices and has time Complexity of $O(|V|)$.

3. $e \in E'$ and $\hat{w}(e) < w(e)$:
   The algorithm should do nothing. Why? Because since the edge is already in the tree this mean it is one of the minimum value in the graph therefore if its new smaller value will be in the tree without change ,Because if we try to add the edge e to the Spanning Tree T so that it creates a sub-graph with exactly 1 cycle, we can conclude that the added edge will be the longest edge in that cycle, whether with the new weight $\hat{w}(e)$ or even the old weight $w(e)$. Thus we can conclude that decreasing the weight to an already minimum edge to the tree should remain the edge within the Tree T. Therefore the Minimum Spanning Tree T after the edge's update should be the same as before the update. Therefore we can conclude that it takes Constant time.

4. $e \in E'$ and $\hat{w}(e) > w(e)$:
   Since the edge is already a part of the Minimum Spanning Tree but the edge's weight is increased, we can conclude that there may be another Minimum Spanning Tree T with a lower overall weight that doesn't contain the updated edge e.
   Therefore the algorithm should do the following:
   - Let $e = (u, v)$ where e is the edge to be updated between the vertices u and v.
   - Using Depth-First Search, it will split the vertices in the Tree into two sub-trees T_u and T_v, Where T_u is a sub-tree that holds all the vertices that are reachable from vertex u in T without going through v.
   While T_v is also another sub-tree that holds all the vertices that are reachable from vertex v in T but without going through u.
   - Where $T_u \cap T_v = \varnothing$ and $T_u \cup T_v = V$.
   - Thus the algorithm should go through the edges $\notin T_u$ and $\notin T_v$ but $\in E$. And find all the connecting edges that have one vertex $\in T_u$ and the other $\in T_v$. Then find the smallest weight edge $e_{\min}$. Where If the weight of the newly selected smallest edge $e_{\min}$ is less than that of the initial edge e, then replace edge e in initial T with the new edge $e_{\min}$ and remove edge e from the Minimum Spanning Tree. Where this will take time Complexity of $O(|E|)$.

3.2 Updating Minimum Spanning Tree Implementation

Since it was stated in the assignment that we should implement at least one algorithm for non-constant time algorithms. Therefore we decided tom implement cases 1,2 and 3 only. Where our implementation is based Based on the algorithms described in the previous section 3.1. These implementations can be founds in Listings 6 and 7.

4 Honour Declaration

We hereby certify that this report and all its uploaded attachments were produced solely by our team, except where explicitly stated otherwise and clearly referenced. We hereby state 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.

```
1  import unittest
2  # Sample matrix provided by us:
3  from string import ascii_lowercase
4
5
6  # Solution to part b:
7  def min_difference(u, r, R):
8      """
9      Sig: string, string, int[0..|A|, 0..|A|] ==> int
10     Pre: Both strings should only be lowercased alphabetical letters, from a to z.
11     Post: Returns the number of differences between both strings
12     Example: Let R be the resemblance matrix where every change and skip costs 1
13             min_difference("dinamck","dynamic",R) ==> 3
14     """
15     # To get the resemblance between two letters, use code like this:
16     # difference = R['a']['b']
17     u = u.lower()
18     r = r.lower()
19     # Variant: i, j
20     # InVariant: len(r), len(u)
21     D = [[0 for i in range(len(r) + 1)] for j in range(len(u) + 1)]
22     D[0][0] = R['-']['-']
23
24     # Variant: i, D[i][0]
25     # InVariant: len(u), D[i-1][1:]
26     #initialize the base cases for the first column
27     for i in range(1,len(u)+1):
28         D[i][0] = D[i-1][0]+R[u[i-1]]['-']
29
30     # Variant: j, D[0][j]
31     # InVariant: len(r)
32     #initialize the base cases for the first raw
33     for j in range(1, len(r)+1):
34         D[0][j] = D[0][j-1] + R['-'][r[j-1]]
35
36     # Variant: i
37     # InVariant: len(u)
38     for i in range(1, len(u)+1):
39
40         # Variant: j
41         # InVariant: len(r), D[i][j]
42         for j in range(1, len(r)+1):
43             D[i][j] = min((D[i-1][j] + R[u[i-1]]['-']), (D[i][j-1]+ R['-'][r[j-1]] ), (D[i-1][j-1] + /
                R[u[i-1]][r[j-1]]))
44
45     return D[len(u)][len(r)]
```

Listing 1: Minimum Difference - Implementation

```
# Solution to part c:
def min_difference_align(u, r, R):
    """
    Sig: string, string, int[0..|A|, 0..|A|] ==> int, string, string
    Pre: Both strings should only be lowercased alphabetical letters, from a to z.
    Post: Returns the number of differences between both strings and the alginment for both strings, /
        with "-" for each difference.
    Example: Let R be the resemblance matrix where every change and skip costs 1
            min_difference_align("dinamck","dynamic",R) ==>
                                    3, "dinam-ck", "dynamic-"
    """
    u = u.lower()
    r = r.lower()
    # Variant: i, j
    # InVariant: len(r), len(u)
    D = [[0 for i in range(len(r) + 1)] for j in range(len(u) + 1)]
    D[0][0] = R['-']['-']

    # Variant: i, D[i][0]
    # InVariant: len(u), D[i-1][1:]
    # initialize the base cases for the first column
    for i in range(1, len(u)+1):
        D[i][0] = D[i-1][0] + R[u[i-1]]['-']

    # Variant: j, D[0][j]
    # InVariant: len(u), D[1:][j-1]
    # initialize the base cases for the first row
    for j in range(1, len(r)+1):
        D[0][j] = D[0][j-1] + R['-'][r[j-1]]

    # Variant: i
    # InVariant: len(u)
    for i in range(1, len(u)+1):
        # Variant: j, D[i][j]
        # InVariant: len(r)
        for j in range(1, len(r)+1):
            D[i][j] = min((D[i-1][j] + R[u[i-1]]['-']), (D[i][j-1]+ R['-'][r[j-1]] ), (D[i-1][j-1] + /
                R[u[i-1]][r[j-1]]))

    diff = D[len(u)][len(r)]
    u_new = u
    r_new = r
    i = len(u)
    j = len(r)
    while i != 0 or j != 0:
        if i>0 and j> 0 and D[i-1][j-1] + R[u[i-1]][r[j-1]] == D[i][j]:
            i = i-1
            j = j-1
        elif i>0 and D[i-1][j]+R[u[i-1]]['-'] == D[i][j]:
            r_new = r_new[:j] + "-" + r_new[j:]
            i= i-1
        elif j>0 and D[i][j-1]+R['-'][r[j-1]] == D[i][j]:
            u_new = u_new[:i]+"-"+u_new[i:]
            j = j-1
    return diff, u_new, r_new
```

Listing 2: Minimum Difference Align - Implementation

```
import unittest
import networkx as nx
"""IMPORTANT:
We're using networkx only to provide a reliable graph
object. Your solution may NOT rely on the networkx implementation of
any graph algorithms. You can use the node/edge creation functions to
create test data, and you can access node lists, edge lists, adjacency
lists, etc. DO NOT turn in a solution that uses a networkx
implementation of a graph traversal algorithm, as doing so will result
in a score of 0.
"""


def ring_detection(g, n, visited, parent):
    """
    Sig: graph G(node, edge), int, int[0..j-1], int ==> boolean
    Pre: n is a non-negative node, and the Graph consists of non negative integers nodes.
    Post: returns True if ring is detected in the graph.
    Example:
        ring_detection(g1, 0, [], -1) ==> False
        ring_detection(g1, 5, [], 4) ==> True
    """
    visited[n] = True
    #Variant: i, visited
    #InVariant: g
    for i in g.adjacency_list()[n]:
        if not visited[i]:
            if ring_detection(g, i, visited, n):
                return True
        else:
            if i != parent:
                return True
    return False


def ring(G):
    """
    Sig: graph G(node, edge) ==> boolean
    Pre: The graph consists of non negative integer nodes.
    Post: returns True if ring is detected in the graph.
    Example:
        ring(g1) ==> False
        ring(g2) ==> True
    """
    visited_nodes = [False for i in range(G.number_of_nodes())]
    # this if to insure that it goes for all nodes if the graph not connected
    # Variant: i, visited_nodes
    # InVariant: g
    for i in range(G.number_of_nodes()):
        if not visited_nodes[i]:
            if ring_detection(G, i, visited_nodes, -1):
                return True
    return False
```

Listing 3: Graph Ring Detection - Implementation

```
1
2  def ring_detection_extended(g, n, visited, parent, node_parent, ring, found):
3      """
4      Sig: graph G(node, edge), int, int[0..n-1], int, [0..m-1], int[0..j-1], boolean ==> boolean, /
           int[0..j-1]
5      Pre: The graph consists of non negative integer nodes.
6      Post: Returns True if ring is detected in the graph and a list that consists of the ring's node /
           values.
7            But returns False and an empty list if no ring exists in the input graph.
8      Example:
9          ring_detection_extended(g1, 2, [1], 1, [0, 1], [], False) ==> False, []
10         ring_detection_extended(g1, 1, [1, 2, 4], 4, [0, 1, 2, 4], [], True) ==> True, [1, 2, 4, 1]
11     """
12     visited[n] = True
13     if n != 0:
14         node_parent[n] += node_parent[parent]
15         node_parent[n].append(parent)
16     # Variant: i, visited
17     # InVariant: g
18     for i in g.adjacency_list()[n]:
19         if not visited[i]:
20             isring, ring_list,found = ring_detection_extended(g, i, visited, n, node_parent, ring, found)
21             if isring is True and found is False:
22                 ring = node_parent[i]
23                 ring.append(i)
24                 ring += ring_list
25                 found = True
26                 return True, ring, found
27             elif isring is True and found is True:
28                 return True, ring_list, found
29         else:
30             if i != parent:
31                 ring.append(i)
32                 return True, ring, found
33     if not found:
34         return False, [], found
```

Listing 4: Graph Ring Detection Extended - Implementation Continued

```
1  def ring_extended(G):
2      """
3      Sig: graph G(node, edge) ==> boolean, int[0..j−1]
4      Pre: The graph consists of non negative integer nodes.
5      Post: Returns True if ring is detected in the graph and a list that consists of the ring's node /
           values.
6            But returns False and an empty list if no ring exists in the input graph.
7      Example:
8          ring(g1) ==> False, []
9          ring(g2) ==> True, [1, 2, 4, 1]
10     """
11     # Variant: i, len(visited_nodes)
12     # InVariant: G
13     visited_nodes = [False for i in range(G.number_of_nodes())]
14     # Variant: i, len(node_parent)
15     # InVariant: G
16     node_parent = [[] for i in range(G.number_of_nodes())]
17     # this is to insure that it goes for all nodes if the graph not connected
18     # Variant: i
19     # InVariant: G
20     ring = []
21     for i in range(G.number_of_nodes()):
22         if not visited_nodes[i]:
23             isring, ring, found = ring_detection_extended(G, i, visited_nodes, −1, node_parent, ring, /
                   False)
24             if isring:
25                 x = ring[−1]
26                 index = 0
27                 for j in range(len(ring)−2, −1, −1):
28                     if ring[j] == x:
29                         index = j
30                         break
31                 return True, ring[index:]
32     return False, ring
```

Listing 5: Graph Ring Detection Extended - Implementation Continued

```
1  import unittest
2  import networkx as nx
3  import ring
4  def update_MST_1(G, T, e, w):
5      """
6      Sig: graph G(V,E), graph T(V, E), edge e, int ==>
7      Pre: None
8      Post: updates Graph G while Tree T remains the same
9      Example: TestCase 1
10     """
11     (u, v) = e
12     assert(e in G.edges() and e not in T.edges() and w > G[u][v]['weight'])
13     G[u][v]"weight"] = w
14
15
16
17 def update_MST_2(G, T, e, w):
18
19     """
20     Sig: graph G(V,E), graph T(V, E), edge e, int ==>
21     Pre: None
22     Post: updates Graph G with new weight, and Tree T could be updated or stays the same /
           depending on the weight value
23     Example: TestCase 2
24     """
25     mapper = {}
26     counter = 0
27     # Variant: i, len(mapper), counter
28     # InVariant: T
29     for i in T.nodes():
30         mapper[i] = counter
31         counter += 1
32     (u, v) = e
33     assert(e in G.edges() and e not in T.edges() and w < G[u][v]['weight'])
34     G[u][v]"weight"] = w
35     T.add_edge(u, v, weight=w)
36     H = nx.relabel_nodes(T, mapper)
37     isring, ring_list = ring.ring_extended(H)
38     max = 0
39     max_index = 0
40     # Variant: i,
41     # InVariant: len(ring_list)
42     for i in range(len(ring_list)-1):
43         edge_weight = H[ring_list[i]][ring_list[i+1]]['weight']
44         if edge_weight>max:
45             max = edge_weight
46             max_index = i
47     edge = ring_list[max_index:max_index+2]
48     new_u = mapper.keys()[mapper.values().index(edge[0])]
49     new_v = mapper.keys()[mapper.values().index(edge[1])]
50     T.remove_edge(new_u, new_v)
51     return T
```

Listing 6: Updating Minimum Spanning Tree - Implementation

```
1  def update_MST_3(G, T, e, w):
2      """
3      Sig: graph G(V,E), graph T(V, E), edge e, int ==>
4      Pre: None
5      Post: updates Graph G while Tree T remains the same
6      Example: TestCase 3
7      """
8      (u, v) = e
9      assert(e in G.edges() and e in T.edges() and w < G[u][v]['weight'])
10     G[u][v]["weight"] = w
11
12
13
14 def update_MST_4(G, T, e, w):
15     """
16     Sig: graph G(V,E), graph T(V, E), edge e, int ==>
17     Pre: None
18     Post: updates Graph G with new weight, and Tree T could be updated or stays the same /
           depending on the weight value
19     Example: TestCase 4
20     """
21     (u, v) = e
22     assert(e in G.edges() and e in T.edges() and w > G[u][v]['weight'])
23     G[u][v]["weight"] = w
24     T = nx.minimum_spanning_tree(G)
25     return T
```

Listing 7: Updating Minimum Spanning Tree - Implementation Continued