Algorithms & Data Structures II (course 1DL231)
Uppsala University – Autumn 2023
Report for Assignment 2

Yueyuan Huang

23rd November 2023

Part 1
Search-String Replacement

A The Recursive Equation

Given an alphabet $\mathcal{A}$, a resemblance matrix $\mathcal{R}$ and two strings s and r having lengths n and m accordingly, define $F(\mathcal{A}, \mathcal{R}, i, j)$ as the difference between $s[: i]$ and $r[: j]$, i.e., the first i characters of s and the first j characters of r.

The recursive equation for $F(\mathcal{A}, \mathcal{R}, i, j)$ can be written as

$$F(\mathcal{A}, \mathcal{R}, i, j) = \begin{cases} 0 & \text{if } i \ == \ 0 \ and \ j \ == \ 0, \\ F(\mathcal{A}, \mathcal{R}, i-1, j) + R[s[i-1]]['-'] & \text{if } i \ > \ 0 \ and \ j \ == \ 0, \\ F(\mathcal{A}, \mathcal{R}, i, j-1) + R['-'][r[j-1]] & \text{if } i \ == \ 0 \ and \ j \ > \ 0, \\ min \ \{F(\mathcal{A}, \mathcal{R}, i-1, j) + R[s[i-1]]['-'], & \\ F(\mathcal{A}, \mathcal{R}, i, j-1) + R['-'][r[j-1]], & \\ F(\mathcal{A}, \mathcal{R}, i-1, j-1) + R[s[i-1]][r[j-1]]\} & \text{otherwise.} \end{cases}$$

From the recursive equation, we can say the problem of computing the minimum difference has the optimal substructure property because all of $F(\mathcal{A}, \mathcal{R}, i-1, j)$, $F(\mathcal{A}, \mathcal{R}, i, j-1)$ and $F(\mathcal{A}, \mathcal{R}, i-1, j-1)$ are parts of $F(\mathcal{A}, \mathcal{R}, i, j)$. It also has overlapping subproblems since $F(\mathcal{A}, \mathcal{R}, i, j)$ depends on $F(\mathcal{A}, \mathcal{R}, i-1, j)$, $F(\mathcal{A}, \mathcal{R}, i, j-1)$ and $F(\mathcal{A}, \mathcal{R}, i-1, j-1)$, then $F(\mathcal{A}, \mathcal{R}, i-1, j)$ will depend on $F(\mathcal{A}, \mathcal{R}, i-1, j-1)$ again.

B The DP Algorithm

As shown in 1. The bottom-up approach is chosen: We iterate through a dp matrix ($dp[i][j]$ stores the value of $F(\mathcal{A}, \mathcal{R}, i, j)$) from up-left to down-right, updating the current dp value according to the recursive equation A.

C The Extended Algorithm

The algorithm shown in 2 extends 1 returning also a specific positioning for the minimum difference.

```python
def min_difference(s: str, r: str, R: Dict[str, Dict[str, int]]) -> int:
    '''
    Pre: For all characters c in s 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
    '''
    n, m = len(s), len(r)
    dp = [[0]*(m+1) for _ in range(n+1)]
    for i in range(n):
       dp[i+1][0] = dp[i][0]+R[s[i]]['-']
    for j in range(m):
       dp[0][j+1] = dp[0][j]+R['-'][r[j]]
    for i in range(n):
       for j in range(m):
          dp[i+1][j+1] = min(dp[i][j+1]+R[s[i]]['-'], dp[i+1][j]+R['-'][r[j]], /
              dp[i][j]+R[s[i]][r[j]])
    return dp[-1][-1]
```

Listing 1: Python function $min\_difference(s, r, R)$.

D Time Complexity

The time complexity of the extended algorithm 2 should be $\mathcal{O}(|s| \cdot |r|)$ because we traverse through the state space (whose size is $(|s|+1) \cdot (|r|+1)$) only once, and we spend constant time dealing with a single state (i, j) (updating $dp[i][j]$, $sp[i][j]$ and $rp[i][j]$) during the traversal.

Part 2

Recomputing a Minimum Spanning Tree

A General Discussion

Given gragh $G = (V, E)$ and its mst $T = (V, E')$, consider four cases of updating the weight of a particular edge $e \in E$ from $w(e)$ to $\hat{w}(e)$:

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

   In this case we do not need to do any modification to T, because as e was not chosen before the update, it is definitely not going to be chosen after its weight being increased. This procedure costs $\mathcal{O}(1)$.

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

   If we simply add e to T, a cycle will be formed. So we can perform a DFS on the original T to get the path P between the two nodes of the updated edge e. The cycle will then be $C = \{P, e\}$. After that we search for edge with the maximum weight in the loop, $e^* = argmax_{e' \in C}\ w(e')$, and remove it from the mst. Now T is guaranteed to be an optimal mst. The DFS takes $\mathcal{O}(|V|)$ time and finding the heaviest edge on P costs $\mathcal{O}(|V|)$ in worst case, so the overall procedure takes $\mathcal{O}(|V|)$ time.

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

```python
def min_difference_align(s: str, r: str,
                         R: Dict[str, Dict[str, int]]) -> Tuple[int, str, str]:
    '''
    Pre: For all characters c in s 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_align('dinamck', 'dynamic', R) -->
                                  3, 'dinam-ck', 'dynamic-'
                               or 3, 'dinamck', 'dynamic'
    '''
    n, m = len(s), len(r)
    dp = [[0]*(m+1) for _ in range(n+1)]
    # positioning of s and r
    sp, rp = [['']*(m+1) for _ in range(n+1)], [['']*(m+1) for _ in range(n+1)]
    for i in range(n):
        dp[i+1][0] = dp[i][0]+R[s[i]]['-']
        sp[i+1][0] = sp[i][0]+s[i]
        rp[i+1][0] = rp[i][0]+'-'
    for j in range(m):
        dp[0][j+1] = dp[0][j]+R['-'][r[j]]
        sp[0][j+1] = sp[0][j]+'-'
        rp[0][j+1] = rp[0][j]+r[j]
    for i in range(n):
        for j in range(m):
            up = dp[i][j+1]+R[s[i]]['-']
            left = dp[i+1][j]+R['-'][r[j]]
            upleft = dp[i][j]+R[s[i]][r[j]]
            if up < left and up < upleft:
                dp[i+1][j+1] = up
                sp[i+1][j+1] = sp[i][j+1]+s[i]
                rp[i+1][j+1] = rp[i][j+1]+'-'
            elif left < upleft:
                dp[i+1][j+1] = left
                sp[i+1][j+1] = sp[i+1][j]+'-'
                rp[i+1][j+1] = rp[i+1][j]+r[j]
            else:
                dp[i+1][j+1] = upleft
                sp[i+1][j+1] = sp[i][j]+s[i]
                rp[i+1][j+1] = rp[i][j]+r[j]
    return dp[-1][-1], sp[-1][-1], rp[-1][-1]
```

Listing 2: Python function $min\_difference\_align(s, r, R)$.

This case is similar to case 1, in which we need to do nothing because e will remain in T after its weight begin decreased. This costs $\mathcal{O}(1)$.

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

   Let the two nodes of e be u and v. We first remove e from E', separating T into two parts. Then we do DFS from u and from v, adding nodes of the two parts into two containers (C_u and C_v) supporting $\mathcal{O}(1)$ lookup, e.g., hash sets. After that we iterate through all edges $e' \in E \setminus E'$ (including e) to find the edge connecting two parts (i.e. one node is in C_u and the other is in C_v) and having the least weight, which we will add into T to get an optimal mst. The DFS's costs $\mathcal{O}(|E'|)$, and the iteration costs $\mathcal{O}(|E|)$, so in total the procedure spends $\mathcal{O}(|E|)$ time.

B Simpler Cases

The first and third case's implementation is shown in 3 and 4.

```python
def update_MST_1(G: Graph, T: Graph, e: Tuple[str, str],
              weight: int) -> Union[Tuple[None, None],
                                Tuple[Tuple[str, str],
                                     Tuple[str, str]]]:
    '''
    Pre: G is an undirected weighted graph, T is a minimum spanning tree of G,
         e is an edge in G but not in T, and weight is greater than the current
         weight of e.
    Post: The new weight is updated in G, and T is a minimum spanning tree of
         the new graph G (remains unchanged).
    Ex: TestCase 1 below
    '''
    (u, v) = e
    assert (e in G and e not in T and weight > G.weight(u, v))
    G.set_weight(u, v, weight)
    return None, None
```

Listing 3: Python function $update\_mst\_1(G, T, e, weight)$.

```python
def update_MST_3(G: Graph, T: Graph, e: Tuple[str, str],
              weight: int) -> Union[Tuple[None, None],
                                Tuple[Tuple[str, str],
                                     Tuple[str, str]]]:
    '''
    Pre: G is an undirected weighted graph, T is a minimum spanning tree of G,
         e is an edge both in G and in T, and weight is less than the current
         weight of e.
    Post: The new weight is updated in G and in T, and T is a minimum spanning
         tree of the new graph G (remains unchanged).
    Ex: TestCase 3 below
    '''
    (u, v) = e
    assert (e in G and e in T and weight < G.weight(u, v))
    G.set_weight(u, v, weight)
    T.set_weight(u, v, weight)
    return None, None
```

Listing 4: Python function $update\_mst\_3(G, T, e, weight)$.

C More Complicated Cases

The second and fourth case's implementation is shown in 5 and 6. The helper function dfs 7 is used.

D Real-world Application

A possible situation where the problem may occur is maintaining a logistics route network. Say a logistics company delivers package to all cities in a country. Consider the state's road network being an undirected weighted graph $G = (V, E)$ (nodes V are cities, edges E are roads, and

```python
def update_MST_2(G: Graph, T: Graph, e: Tuple[str, str],
                 weight: int) -> Union[Tuple[None, None],
                                       Tuple[Tuple[str, str],
                                             Tuple[str, str]]]:
    '''
    Pre: G is an undirected weighted graph, T is a minimum spanning tree of G,
         e is an edge in G but not in T, and weight is less than the current
         weight of e.
    Post: The new weight is updated in G, and T is a minimum spanning tree of
         the new graph G (updated if necessary).
    Ex: TestCase 2 below
    '''
    (u, v) = e
    assert (e in G and e not in T and weight < G.weight(u, v))
    G.set_weight(u, v, weight)
    prev = {} # for finding path
    def add2(d): # currying: add to dict
       def add2d(u, p):
          d[u] = p
       return add2d
    # find path from u to v in T
    dfs(T, u, '', add2(prev)) # assume '' is not a node in T
    ne = e # new edge to remove from T (with the greatest weight)
    while v != u:
       if G.weight(prev[v], v) > G.weight(*ne):
          ne = (prev[v], v)
       v = prev[v]
    if ne == e: return None, None # e has the greatest weight in the cycle
    T.remove_edge(*ne)
    T.add_edge(*e, G.weight(*e))
    return ne, e
```

Listing 5: Python function $update\_mst\_2(G, T, e, weight)$.

weight of an edge $w(e)$ is the time spent going from one end of the road e to another). Then the routes the company chooses (and operates daily) should be a minimum spanning tree T of G. As G will not stay constant because new highways may be build to replace the old ones and existing roads may be under construction, in which cases the weight of some edge in G will be changed. Therefore the company have to recompute T from time to time in order to cut its cost.

```
def update_MST_4(G: Graph, T: Graph, e: Tuple[str, str],
                 weight: int) -> Union[Tuple[None, None],
                                 Tuple[Tuple[str, str],
                                       Tuple[str, str]]]:
    '''
    Pre: G is an undirected weighted graph, T is a minimum spanning tree of G,
         e is an edge both in G and in T, and weight is greater than the
         current weight of e.
    Post: The new weight is updated in G and T, and T is a minimum spanning
         tree of the new graph G (updated if necessary).
    Ex: TestCase 4 below
    '''
    (u, v) = e
    assert (e in G and e in T and weight > G.weight(u, v))
    G.set_weight(u, v, weight)
    T.remove_edge(u, v) # remove e from T, spliting T into two trees
    cu, cv = set(), set()
    def add2(s): # currying: add to set
       def add2s(e, _):
          s.add(e)
       return add2s
    # add nodes of two trees into cu or cv (for fast lookup)
    dfs(T, u, '', add2(cu)) # assume '' is not a node in T
    dfs(T, v, '', add2(cv)) # same as above
    edges = set(G.edges) - set(T.edges) # edges in E \ E'
    ne = None # new edge to add into T (with the least weight)
    for edge in edges:
       (u, v) = edge
       if (u in cu and v in cv) or (u in cv and v in cu):
          if ne is None or G.weight(u, v) < G.weight(*ne):
             ne = edge
    T.add_edge(*ne, G.weight(*ne))
    if set(list(e)) == set(list(ne)): return None, None # e and ne is the same edge
    return e, ne
```

Listing 6: Python function $update\_mst\_4(G, T, e, weight)$.

```
def dfs(G: Graph, u: str, p: str, func: Callable[[str, str], None]) -> None:
    '''
    Pre: G is an undirected weighted graph, u and p are nodes in G, and func
         is a function that takes two nodes u and p as input.
    Post: The function func is applied to each pair of nodes (u, p) in G (p
         is the node visited just before u).
    Ex: dfs(G, u, '', print) will print all nodes in G with their parent.
    '''
    func(u, p)
    for v in G.neighbors(u):
       if v == p: continue
       dfs(G, v, u, func)
```

Listing 7: Python function $dfs(G, u, p, func)$.