Assignment 4

Andrew Hobden (V00788452)

November 19, 2014

1 Dynamic Programming

Observing the following:

$$s = [1, 2, 3, 4, 5, 6, 7, 8, 9], t = [1, 2, 8, 9] \rightarrow r = [1, 2, 8, 9]$$

$$s = [1, 2, 4, 3, 5], t = [1, 2, 4, 3, 6] \rightarrow r = [1, 2, 4]$$

$$s = [4, 5, 6, 1, 2], t = [1, 2, 3, 4, 5, 6, 7, 8, 9] \rightarrow r = [4, 5, 6]$$

Defining $LCIS(s_1...s_n, t_1...t_m, last)$ recursively:

- If $s_n = t_m$ and $s_n > last$, then $LCIS = max\{LCIS(s_1...s_{n-1}, t_1...t_{m-1}, s_n)+ 1, LCIS(s_1...s_{n-1}, t_1...t_m, last), LCIS(s_1...s_n, t_1...t_{m-1}, last)\}$
- If $s_n \neq t_m$, then $LCIS = max\{LCIS(s_1...s_{n-1}, t_1...t_m, last), LCIS(s_1...s_n, t_1...t_{m-1}, last)\}$

It's possible to solve this problem using two arrays, L which stores the length of the LCIS ending at a given element i, and P which stores the previous element of the LCIS. With these it's possible to reconstruct the LCIS.

For each element of s and t at indexes i and j respectively:

- Initialize $cur = 0$, $last = -1$.
- If $s_i = t_j$ and the size of the resulting LCIS ($cur+1$) is larger then previous found sequences (via checking L_j), then $C_j = cur + 1$, and $P_j = last$.
- If $t_j < s_i$ and $cur < C_j$, then $cur = C_j$ and $last = j$.

```
function increasing_subsequence(s: int[], t: int[]) {
    size = max(s, t)
    (largest, smallest) =
        if s.len() == size
            (s, t)
        } else {
            (t, s)
        }
    c = int[]
    p = int[]

    // Build arrays
    for i in range(0, smallest.len()) {
        (cur, last) = (0, -1)
        for j in range(0, largest.len()) {
            if smallest[i] == largest[j] & cur+1 > c[j] {
                c[j] = cur + 1
                p[j] = last
            }
            if smallest[i] > largest[j] & cur < c[j] {
                cur = c[j]
                last = j
            }
        }
    }

    // Find the end of the sequence
    (length, index) = (0, 0)
    for i in range(0, size) {
        if c[i] > length {
            length = c[i]
            index = i
        }
    }

    // Find the sequence
    sequence = []
    while index != -1 {
        seq.push_front(largest[index])
        index = p[index]
    }
    return sequence
}
```

2 Space Complexity

When using the LCS algorithm discussed in class, the $O(mn)$ space complexity arises from having to construct the table of past results.

		C	H	I	M	P	A	N	Z	E	E
	0	0	0	0	0	0	0	0	0	0	0
H	0	0	1	1	1	1	1	1	1	1	1
U	0	0	1	1	1	1	1	1	1	1	1
M	0	0	1	1	2	2	2	2	2	2	2
A	0	0	1	1	2	2	3	3	3	3	3
N	0	0	1	1	2	2	3	4	4	4	4

It can be observed that values always increase as you move between $row_1...row_n$ and column $col_1...col_m$. There are two cases of where the algorithm selected values from to consider:

- $\max\{\uparrow, \leftarrow\}$ That is, $(v_{r,c} = v_{r,c-1} || v_{r,c} = v_{r-1,c})$ where (r, c) are row, column in all cases except where the algorithm would have choosen the $\nwarrow$ case instead. This only depends on the previous row and column.
- $\nwarrow$ In this case the algorithm only depends on the previous iteration's result.

As such, it's possible to utilize two arrays in the case where only the length is required. Some array *cur* which contains the current row's result, and some array *prev* to store the last row's result.

		C	H	I	M	P	A	N	Z	E	E
	...	...	...	...	...	...	...	...	...	...	...
U	0	0	1	1	1	1	1	1	1	1	1
M	0	0	1	1	2	2	2	2	2	2	2
	...	...	...	...	...	...	...	...	...	...	...

The space complexity of this is $O(2max(n, m)) = O(max(n, m))$ which can be shown to be no greater than $O(n + m)$ in all cases.

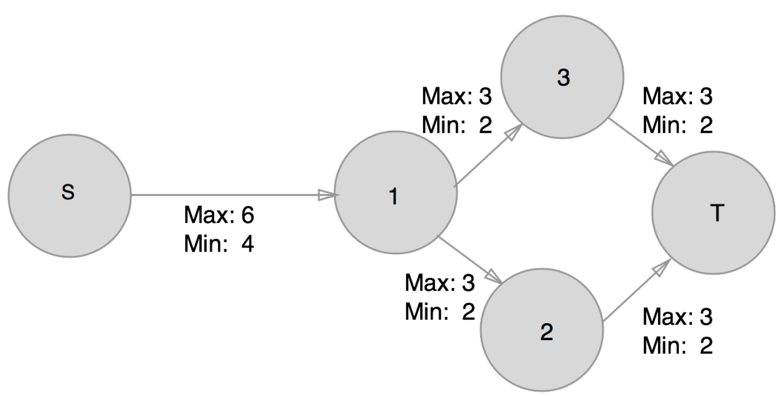
```
function lcs(s: int[], t: int[]) {
    size = max(s, t)
    (largest, smallest) =
        if s.len() == size
            (s, t)
        } else {
            (t, s)
        }
    cur = int[]  // Row i
    prev = int[] // Row i-1

    // Build arrays
```

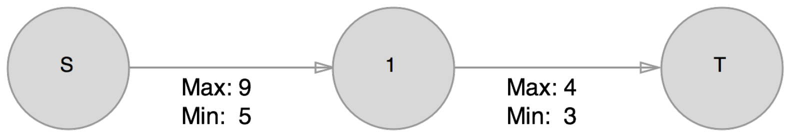
```
    for i in range(0, smallest.len()) {
        for j in range(0, largest.len()) {
            if smallest[i] == largest[j] {
                cur[j] = prev[j-i]
            } else if prev[j] > cur[j-1]
                cur[j] = prev[j]
            } else {
                cur[j] = cur[j-1]
            }
        }
        // Swap them (cur will get overwritten)
        (cur, prev) = (prev, cur)
    }
    // Last element is the longest subsequence.
    // Remember, we just swapped (prev, cur)
    // So prev[] contains the 'true' current row.
    return prev.getLast()
}
```

3 Max Flow with Min Capacities

Where might this variant of the Max Flow problem experience troubles? Observe the following scenarios:



In this case, it's possible to have a flow of 3 from $s \to 1 \to 3 \to t$ and a flow of 3 from $s \to 1 \to 2 \to t$. Alone, neither of these flows satisfies the *min* restriction of 4 of $s \to 1$, but together, they are valid.



In this case, it's not possible to have a flow at all, since the minmum flow from $S \rightarrow 1$ is 5, while the maximum of $1 \rightarrow T$ is 4.

In order to account for these two scenarios, as well as other subproblems that arise, we can *shift* the input graph's weights. By making $Capacity(S_{i,j}) = MaxCapacity(G_{i,j}) - MinCapacity(G_{i,j})$ for some$S$, then computing $FordFulkerson(S)$ and then *unshifting* the resulting flow network F such that $F_{i,j} = F_{i,j} + MinCapacity(G_{i,j})$ if and only if $F_{i,j} > 0$.

```
function maxFlowWithMins(G: int[][], s: int, t: int) {
    S: int[][]
    for vertex in G {
        for edge in vertex {
            S[vertex][edge] = edge.max - edge.min
            // min of S is 0 always.
        }
    }
    F = FordFulkerson(S, s, t)
    for vertex in F {
        for edge in vertex {
            if F[vertex][edge] != 0 {
                F[vertex][edge] += G[vertex][edge].min
            }
        }
    }
    return F
}
```