COMP3600/6466 — Algorithms

Convenor: Hanna Kurniawati (hanna.kurniawati@anu.edu.au)

Solution Example for Assignment 3

Part A

1. (a) 1.

   (b) We prove that $|\mathcal{T}(G)| = 1$ by contradiction. Suppose $|\mathcal{T}(G)| = 2$ and suppose the two MSTs of G are $T_1(V_1, E_1)$ and $T_2(V_2, E_2)$. Let $e \in E$ be an edge of G that has the smallest weight and lies in E_1 xor E_2 (i.e., in either one of the MSTs but not in both). Wlog., suppose $e \in E_1$. If we add e to E_2, then we will have a cycle (let's denote the cycle as C) in T_2, and so to ensure T_2 remains a tree, we need to remove an edge from C. Now, there must at least be one edge in C that is not in E_1 because otherwise, T_1 would have a cycle and therefore not a tree. Let's take one such edge (i.e., in C but not in E_1) and denote it as f. Since e is the edge with the smallest weight that lies exactly in one of T_1 or T_2, $w(e) \leq w(f)$. Since the edges of G have unique weight, $w(e) < w(f)$. Therefore, T_2 cannot have the minimum total weight, and therefore cannot be an MST. This is a contradiction, and proves that the MST of a graph with unique edge weight is unique.

2. (a)

Algorithm 1 PalindromeCheck(String S)

```
1: n = length(S)
2: RS = ""
3: for i = 1 to n do
4:     RS.charAtIndex(n-i+1) = S.charAtIndex(i)
5: [C, B] = LCS-Length(S, Reverse(S))
6: Return n − C[n][n]
```

LCS-Length is the longest common subsequence as defined in p.394 of [CLRS]. Line 2–5 is reversing the input string. We expand it here to help analysis in (c) and (d). In (b), we'll take Reverse(S) to be a function that takes a string as its input and outputs its reverse (e.g., Reverse("abc") will return "cba").

(b) We prove by construction.

First, notice that the LCS of a string S and its reverse is essentially the longest (possibly non-consecutive) sub-palindrome of S. Let's denote this longest sub-palindrome as P and let l = length(P). Suppose Id is the index that associates each character in P with its index in S. For example, when $S = abcabc$, $P = bcb$ and $Id[1] = S[2]$, $Id[2] = S[3]$, $Id[3] = S[5]$.

Now, let's form an array S' of strings of size $2l + 1$, where $S'[2i] = P[i]$ for $i \in [1, l]$ and all other element of S' is initialised to an empty string. We will replace some of these empty strings with substrings of S, such that the concatenation of the elements of S' forms the shortest palindrome that can be generated by adding characters to S. To this end, we perform the following operations:

i. Iterate over the index $i \in [1, l-1]$. Whenever $(Id[i+1] - Id[i]) > 1$, set substring $S_i = [S[Id[i] + 1], \cdots, S[Id[i+1] - 1]]$. Then, replace the empty string in $S'[2i+1]$ with S_i and the empty string in $S'[m - (2i+1)]$, where $m = 2(l+1)$, with Reverse(S_i).

ii. If $Id[1] > 1$, then set substring S_b to be the concatenation of the substring $[S[1], \cdots S[Id[1] - 1]]$ and the substring $[S[Id[l] + 1], \cdots, S[n]]$, where n = length(S). Then, replace the empty string in $S'[1]$ with S_b and the empty string in $S'[2l+1]$ with Reverse(S_b).

The two operations above inserts a total of $2(n - l)$ characters to S' because each character not already in the longest sub-palindrome P, will be inserted twice (loosely speaking, at the first half and later half of the string. However, $(n - l)$ of these characters are copied from S directly, which means the number of new characters inserted to S to form the palindrome is only $(n - l)$. Notice that if we remove even a single character from being inserted, we will either miss inserting the original character of S to S' or remove the character placed at the other half of the string. Either removal will cause the resulting string to no longer

be a palindrome formed by adding characters to S. Therefore, the minimum number of characters to be inserted to S, so that S becomes a palindrome is $(n - l)$, where l can be computed by running LCS-Length between S and its reverse.

(c) Line 1–2 requires constant time, line 3–4 requires $\Theta(n)$ time, line 5 requires $\Theta(n^2)$ based on LCS-Length implementation in p.394 of [CLRS], and line 6 requires constant time. Summing up these time complexity results in a total time complexity of $\Theta(n^2)$ for PalindromeCheck, where n is the length of the string.

(d) Here, we will assuming each character to occupy a single memory unit. This means, S requires $\Theta(n)$ memory, RS requires $\Theta(n)$ memory, n requires a single memory unit, each C and B requires $\Theta(n^2)$ memory. The total memory complexity of the PalindromeCheck is then $\Theta(n^2)$, where n is the length of the input string.

3. (a) Perfect binary tree. Since all leaves of a perfect binary tree lies at the same level and since the length of a Huffman encoding is equal to the path length from root to the leaf node, when the Huffman tree forms a perfect binary tree, the encoding of all the characters would have the same length (i.e., the same as the level of the leaves), which means the encoding would be the same as if we use fixed-length encoding.

(b) Mr T is wrong. AVL tree is a self-balancing binary search tree, which aims to set the differences between the height of the left subtree and the right subtree of any node in the tree differs by only 1. However, Huffman encoding is beneficial because we are allowed to have extremely different encoding length for different characters. This means, when Huffman encoding is represented using an AVL tree (similar to how a Huffman tree represents Huffman encoding), the requirements of AVL tree can sometimes contradict with the desired properties of Huffman encoding.
Note that it is true the levels of the leaves in an AVL tree may differ by much more than 1. However, the balance requirement of an AVL tree, impose certain conditions that we might need to violate to make an efficient Huffman encoding. This contradiction in requirements made AVL tree to not be the most efficient representation of the Huffman code.