Computational complexity homeworks

Michał Łazowik

1 Series 1

1.1 Problem 1.1

We are going to simulate moving left, right, and no movement by using multi-step subroutines. We assume a TM model with a pin at the beginning of the tape.

To do that we first extend the alphabet, so that for each letter a we also have a' and a''.

After each of the steps we would do on a standard TM the following invariant will hold: let's mark head position as i, all letters in the range $[0, i-2]$ are in the form a', and the letter at the position $i-1$ is in the form a''. All the other letters are the original ones.

We simulate movement as follows:

1.1.1 No movement

We make no-oparation steps until we find a letter in the form a'' (we know that the machine will go back to the beginning at some finite step). Then we move one step more and so we land at the desired position. Invariant is already satisfied.

1.1.2 Right

We make no-oparation steps until we find a letter in the form a''. We change it to a', move one step more and change a that is at this position to a''. Then we move one step more and so we land at the desired position with invariant satisfied.

1.1.3 Left

First we make noop steps until we land at the start of the tape. Then, as long as we see a' we replace them with a''. We then land at the a'' that used to be the only one and change it to a. This steps give us the tape with characters of type a'' at $[0, i-1]$ and a at all the other positions (i is the desired head position after we finish this subroutine).

Now we perform "passes". In each pass we check if there is more than one a'' type characters. If there are, we do a next pass and we change the first a'' into a'. When there is only ony a'' type character the invariant is satisfied for our new i and we are already at our desired position (the first character of type a on the tape).

1.1.4 Summary

After any movement subroutine the head points to a character of type a, so all the other operations are perforemed as on the regular TM.

This proves that our type of machine can recognize any language that the regular TM can. We can also simulate our model using the regular multi-tape (single-tape and multi-tape are equivalent) one with a separate tape for the step counter, which means that all languages recognized by our model can be also recognized by the regular one. This proves the statement from the task.

1.2 Problem 1.2

For each pair of nodes we will check if the distance between them equals k. To iterate through verices we keep two counters, the first one iterates v, the second u.

If we root the tree in the node 1 the given representation gives us an easy way to find parent of each node with no additional space usage.

To check the distance between given two nodes u and v:

1.2.1 Calculating depths

First we calculate depth (in number of edges, not their values) of both u and v. To calculate depth of a node we use a separate tape as a counter. Until we hit the root we find the parent of the current node and increment the counter.

1.2.2 Checking the (weighted) distance

We add a new distance counter that will count up to $2k$. If at any point it becomes more than k or the weight we try to add is more than k we move to the next pair of nodes.

Given the depths we start with the node that is deeper in the tree. We again traverse the tree up, but this time in each step we decrement the depth counter and add the weight of the edge to the parent to the distance counter.

When the depth matches the one of the other node we start doing the same procedure for both nodes in turns. This way both nodes will land on the lowest common ancestor of u and v. If the distance counter value equals k we accept, otherwise we move to the next pair of vertices.

Before moving to the next node pair we clear all the counters apart from the first two that indicate which nodes we are looking at.

1.2.3 Summary

All the counters used in the solution have to store at most $max(n, 2k)$, and there is a constant number of them. Thus, each counter uses $O(log(max(n, k))$ space while the input size is $\Omega(n * logn + k)$, so the counter size does not exceed $log(len(input))$. This proves that the solution is in L.

1.3 Additional problem

What if k is given in binary? Instead of having a simple distance counter in section 1.2.2 we can compare with k bit by bit. To get the least significant bit of the path length we do one pass in which we only count bits on the least significant position of each weight. If there is uneven number of them the result will have 1 at this position, otherwise 0. We can compare this to the least significant bit of k. We then divide the number of bits by 2 (this is the carry) and do another pass in which we add bits from the next position, compare, and so on.

This bit counter's value will never be more than the number of bits in the input (in fact it will be much less, as we look only at some of the numbers), so the space used for this counter will be $O(log(len(input)))$. All the other counters do not depend on the size of representation of k.