Complexity - homework 2

Michał Kukuła

7th January 2019

1

Proof. Let's denote with Ω the set of all Turing machines with logarithmic working tape and additional stack. For a given pair (ω, w), where $\omega \in \Omega$ and $w \in \Sigma^*$, $|w| = n$, we will call a configuration each quadruple (q, i, j, m), where:

- $q \in Q(\omega)$ is a current state of machine ω.
- $i \in \{1, ..., n\}$ is a current position of ω on the input tape, where the word w is written.
- $j \in \{1, ..., log(n)\}$ is a current position of ω on the working tape.
- $m \in (\Sigma \cup \{\#\})^{log(n)}$ is a current state of memory - letters written on the working tape ($\#$ is blank).

Total number of configurations for given (ω, w) equals
$|Q(\omega)| \cdot |(\Sigma \cup \{\#\})^{log(n)}| \cdot n \cdot log(n) = qc^{log(n)} n log(n) = qn^{log(c)+1} log(n) = poly(n)$
for some constants q and c. Let's denote with C set of configurations.
Now, consider a function $\gamma : C \times (\Sigma \cup \{\#\}) \to C \times (\{\epsilon\} \cup \Sigma \cup \Sigma^2)$ which describes transitions between configurations, given top letter on a stack tape. We have three cases:

$$\gamma(c, a) = \begin{cases} (c', \epsilon) & (1) \\ (c', a) & (2) \\ (c', ab) & (3) \end{cases}$$

1. From configuration c, when top stack letter is a, we go to configuration c' and move left on stack.
2. From configuration c, when top stack letter is a, we go to configuration c' and do not move on stack.
3. From configuration c, when top stack letter is a, we go to configuration c', move right on stack and write b on the top of it.

We see that such construction describes a deterministic pushdown automata $pda(\omega, w)$, where:

- stack alphabet is a language of ω,
- input alphabet is unary.
- states are configurations, as described above.
- start state is a start configuration of ω - $(q_{start}, 1, 1, \#^{log(n)})$.
- start stack symbol is $\#$.
- accepting states are configurations of form (q_{acc}, i, j, m).
- transition function is γ described above.

Machine ω accepts w if and only if language generated by automata $pda(\omega, w)$ is not empty - all words 1^n correspond to ω's runs of length n. Machine halts in accepting state after n steps if and only if word a 1^n is accepted by $pda(\omega, n)$. So, in order to prove that a language recognized by ω is in P, we would do in $poly(n)$ time two steps:

1. for a given word w, construct $pda(\omega, w)$.
2. check if language recognized by this automata is non-empty.

The important observation is that $pda(\omega, n)$ is of $poly(n)$ size, so composing these two steps will result in a polynomial time algorithm, if both work in polynomial time.
Here is how we do both steps:

1. Consider the Turing machine which firstly writes to the memory the representation of ω, then calculates the transition function γ, given this representation and input word w, and writes all of the transitions to the memory. Size of the machine ω is constant, number of transitions and configurations is polynomial. Obviously, machine works in polynomial time.

2. Firstly, given a deterministic pushdown automata, we would like to construct a context-free grammar, which recognizes the same language. As we know from JAiO course, we can do it in polynomial time and in such way that the number of nonterminals in an output grammar is polynomial of number of states in automata [1].
   Now, after obtaining an equivalent context-free grammar, we do the following:

   (a) We remove all terminals from production rules. We will check if in such grammar the start symbol is nullable.
   (b) For ich production rule $X \to \epsilon$ denote that X is nullable.

[1] https://docs.google.com/document/d/1wzb162HaleLLWwUpE9Mur2EZkRJL72rVPV2dHo0O2kk/edit - W. Czerwiński notes to JAiO 2017 course - section "Równoważność automatów ze stosem i gramatyk" - in polish

(c) Iterate over all production rules. If for some rule $X- > Y_1...Y_k$ all of the nonterminals on the right side of the production rule are nullable, we denote that X is nullable. If X is a start symbol of a grammar, return true.

(d) If in the previous iteration of point a we have denoted at least one new nonterminal as nullable, do point a again. If not, we have found all nullable nonterminals, we stop the algorithm and return false.

In each iteration of point a (except the last one) we have denoted at least one new nonterminals as nullable, so we have performed at most as many iterations, as the number of nonterminal symbol in grammar. Thus, algorithm works in polynomial time. We have then checked in polynomial time if $pda(\omega, w)$ is not empty, qed.

■

2

Proof. We will use the hint and a fact from the lecture that the circuit for words of length n from the uniform sequence of circuits corresponding to $L \in P$ can be found in logarithmic space. What is important - using construction described on lecture, in such circuit each gate can have at most 3 inputs.
We will denote as *gen_circuit*() a procedure, which works in logarithmic space and puts on the top of a stack the representation of *nth* circuit corresponding to language L (of course, the length of the input word can be calculated in logarithmic memory). Let's denote by $q(i)$ quintuple $(i, in_1, in_2, in_3, op)$, which describes an *nth* gate from a circuit - n is a gate number, in_i are the numbers of input gates or values $0, 1$ if already determined and $op \in \{\texttt{or}, \texttt{and}, \texttt{neg}\}$ is an operation linked with gate n. In logarithmic memory, we can keep 4, or any constant number of such quintuples - size of each of them is logarithmic because number of gates in our circuit is polynomial.
Our Turing machine with logarithmic memory and additional stack will work as follows:

1. perform *gen_circuit*() - to put circuit on stack.

2. determine the output gate of the circuit, the input gates of it and operation by scanning the circuit top-down and writing in memory corresponding quintaple. Put $q(out)$ on stack and remove it from the memory.

3. Now, we will work recursively. Put the top quintaple from stack to memory and remove it from stack. Let's denote it by $q(top)$. For each of it's inputs in_i, which do not have determined values, we perform *gen_circuit*() procedure, scan the circuit to find quintuple $q(in_i)$ and put it to memory. Now, we check if value of any of quintuples from memory can be determined - we can do it if all 3 inputs of one of the quintuples correspond to gates where an input word is written. If so, we calculate the value and

replace corresponding input in $q(top)$ to a result value. Now, if some of the input values of $q(top)$ are still not calculated, we push (maybe) modified $q(top)$ back to stack, and then we push each $q(in_i)$, if not determined yet. We clean quintuples from memory and go back to point 3. If all inputs of $q(top)$ are calculated, we determine the value of it and update the current top-of-the-stack quintuple. If now this quintaple can be determined, we pop it from stack, put in memory, calculate and repeat. We do it as long as we can determine top stack quintuple value. After that, we clean memory and go to back point 3.

4. If in any moment we have calculated the value of $q(out)$, we finish the computations and depending on result go to accepting or rejecting state.

■