Computational Complexity — tutorial 5'

Completeness cntd.

Here, we'll solve Problem **4** from last week's tutorial session in detail (there's also some discussion about the problem, so if you wrote down the solution, it would be quite a bit shorter). You may want to understand this proof and try to modify it in order to prove that some problems about deterministic and non-deterministic automata are PSpace-complete.

4. (homework 2017) *Let* pattern *be a word consisting of letters and question marks. A pattern p matches a word w if p can be obtained by taking an infix of w and replacing some of its characters by question marks. For instance,* `ab?d?` *matches* `abcde`, `ababadddd`, *but not* `abcd` *or* `abaaaadx`. *Consider the following problem: given an alphabet Σ and two sets P and N of patterns, decide whether there exists a word $w \in \Sigma^*$ matching all patterns in P and no pattern in N. Prove that this problem is PSpace-complete.*

The proof of completeness, as always, consists of two parts: proving that the problem is in PSpace, and proving that the problem is PSpace-hard.

Part 1: The problem is in PSpace.

We need to find a polynomial space algorithm solving our problem. It's not as easy as it seems since it might turn out that there exists a word as in the statement of the problem, but this word is exponentially long on the size of the input (see Exercise at the end of this document). We'll, however, apply a trick which is sometimes useful when designing polynomial space algorithms: we'll design a *non-deterministic* polynomial space algorithm which „guesses" the characters of a solution one by one (if such a solution exists). Since we cannot afford to store the whole solution in memory, we'll store some information about the prefix of our solution in some succinct way; this information should allow us to decide if the word we guess is a solution to the instance of the problem. If we do this correctly (in polynomial space), we'll prove that our problem is in NPSpace (non-deterministic polynomial space). By Savitch's theorem, we have NPSpace = PSpace, so our problem is in PSpace as well.

We shall now describe how to store information about a (potentially very long) prefix of the solution in a succinct way. For each pattern p (either in P or in N), we can easily create a non-deterministic finite automaton (NFA) with $|p|+1$ states recognizing all words matching this pattern. Now, we want to determine if there exists a word w such that:

- w is accepted by all NFAs for patterns in P, and
- w is rejected by all NFAs for patterns in N (equivalently, w is accepted by complements of these NFAs).

Complementing an NFA requires the determinization of the automaton, which might produce an exponentially large automaton. Hence, it's infeasible to store the complements of NFAs in our memory. However, we can get away with a simple trick: after having consumed a prefix of the solution, for each NFA we want to maintain a subset of the states our NFA might currently be in. This way, after we read a prefix of w, for each NFA (corresponding to a pattern p) we only keep:

- The description of this NFA, which has a polynomial size $O(|p| \cdot |\Sigma|)$, and
- The set of states the NFA can be possibly in after reading this prefix of w (this requires $|p|+1$ bits, one for each state of the NFA).

Adding a character to the prefix of w requires us to update the set of state for each NFA, which can be easily done in polynomial space (even in polynomial time). As soon as for some prefix of w, we can reach an accepting state of every NFA for patterns in P, and we cannot reach any accepting state in any NFA for patterns in N, we accept. Hence, we have a non-deterministic polynomial space solution to the problem. A deterministic polynomial space solution follows from Savitch's theorem.

Still, there's one problem: our non-deterministic machine may not terminate! I know two ways of resolving the problem:

- We know that the information about any prefix of any word can be stored in $S := \sum_{p \in P \cup N}(|p|+1)$ bits. Hence, if there is a word solving our problem, then the shortest such word has length at most 2^S. We can implement an S-bit binary counter which assures us that the solution „guessed" non-deterministically doesn't grow too much.
- We can analyze the proof of Savitch's theorem and understand that it always produces a terminating Turing machine. So even if the original non-deterministic Turing machine might sometimes not terminate, the determinization of it will always stop.

Part 2: The problem is PSpace-hard.

We'll prove this by a reduction from the following PSpace-complete problem:

> PSpace Halting Problem
> Input: deterministic Turing Machine $\mathcal{M}$, its input w, integer k in unary
> Output: does $\mathcal{M}(w)$ accept so that the memory usage of $\mathcal{M}$ does not exceed k?

In this description, all additional details will be grayed out.

Given $\mathcal{M}$, w, k, we'll create (in polynomial time) a set of patterns encoding accepting runs of $\mathcal{M}$ which don't use more than k memory cells. The encoding will have the following format:

`#initial configuration#configuration after 1 step#after 2 steps#...#accepting configuration#`

where each configuration is a word of length k encoding the contents of the tape of $\mathcal{M}$, position of the head of the machine and the current state of the machine. Formally, if the alphabet of TM is Σ, and the set of states is Q, then each character of the description is either $c \in \Sigma$, or $\binom{c}{q}$ for $c \in \Sigma$, $q \in Q$. The initial configuration of $\mathcal{M}$ is $\binom{w_1}{q_0} w_2 w_3 \dots w_{|w|} \underbrace{\texttt{BBBBBB...B}}_{k-|w| \text{ blanks}}$, where q_0 is the initial state of $\mathcal{M}$, and `B` is a blank.

We can assume that $\mathcal{M}$ terminates at the first position on the tape, having cleared the tape (we can add a couple of additional states to $\mathcal{M}$ which will ensure this). Hence, the accepting configuration of $\mathcal{M}$ is unique: $\binom{\texttt{B}}{q_{acc}}\texttt{B}^{k-1}$ for q_{acc}—the accepting state.

We now proceed to describe the patterns:

- We ensure that the sought word starts with the initial configuration; that is, it contains the initial configuration as a substring, but it doesn't contain it anywhere after the first character of the solution. Formally, we add $\#\binom{w_1}{q_0}w_2w_3\ldots w_{|w|}\mathtt{B}^{k-|w|}\#$ to P, and $\mathtt{?}\#\binom{w_1}{q_0}w_2w_3\ldots w_{|w|}\mathtt{B}^{k-|w|}\#$ to N.

- We analogously ensure that the sought word has the accepting configuration as a suffix. We add $\#\binom{\mathtt{B}}{q_{acc}}\mathtt{B}^{k-1}\#$ to P, and $\#\binom{\mathtt{B}}{q_{acc}}\mathtt{B}^{k-1}\#\mathtt{?}$ to N.

- We ensure that the hashes (#) are at distance $k+1$ from each other. For each $1 \leqslant i \leqslant k$, we add a pattern forbidding two hashes from appearing at distance i (we add $\#?^i\#$ to N for each i). Then, for each c in the alphabet of the constructed word different than #, we forbid c from appearing at distance exactly $k+1$ characters from any # (we add $\#?^k c$ to N for each $c \in \Sigma \cup \{\binom{s}{q} \mid s \in \Sigma,\ q \in Q\}$).

- Now, let's denote two adjacent configurations as $a_1a_2\ldots a_k$, $a'_1a'_2\ldots a'_k$ (where a_i is either a character or a character together with a state). Since we consider deterministic Turing machines, we can convince ourselves that each a'_i only depends on the values of a_{i-1}, a_i, and a_{i+1} since the head of the machine can only move one cell at a time. Formally, there exists a function $f : \mathsf{Cell}^3 \to \mathsf{Cell}$ such that $a'_i = f(a_{i-1}, a_i, a_{i+1})$ for each $i \in \{1,\ldots,k\}$, where for simplicity we set $a_0 = a_{k+1} = \mathtt{B}$.

  Now, we want to say a bunch of statements of the form „given the contents of the cells a_{i-1}, a_i, and a_{i+1} in the previous configuration, the contents of a'_i in the following configuration cannot be equal to x" (for each $i \in \{1,2,\ldots,k\}$, each $a_{i-1}, a_i, a_{i+1} \in \mathsf{Cell}$, and $x \in \mathsf{Cell}, x \neq f(a_{i-1}, a_i, a_{i+1})$). We can do this by adding patterns of the following form to N:

  $$\#?^{i-2}a_{i-1}a_ia_{i+1}?^{k-i-1}\#?^{i-1}x?^{k-i}\#.$$

  (There are a few special cases for $i = 1$ and $i = k$ since a_{i-1} or a_{i+1} don't exist, but this isn't much of a problem.)

  In total, we add $O(k \cdot |\mathsf{Cell}|^4)$ patterns of length $2k+3$ (which is still a polynomial on k and the size of the Turing machine).

These requirements are enough to encode terminating runs of Turing machines as finite words (and only them). Hence, the reduction is complete. This reduction can be performed easily in polynomial time, so the problem is PSpace-hard.

Since the problem is in PSpace (Part 1) and PSpace-hard (Part 2), the problem is PSpace-complete.

Exercise. Try to modify the proof above to prove that the following problems are PSpace-complete:

(a) DFA Intersection Non-emptiness
Input: n deterministic finite automatons $A_1, A_2, \ldots, A_n$ over the same alphabet Σ
Output: is there a word $w \in \Sigma^*$ such that $w \in L(A_1) \cap L(A_2) \cap \cdots \cap L(A_n)$?

(b) NFA Universality
Input: a non-deterministic finite automaton A over an alphabet Σ
Output: is $L(A) = \Sigma^*$?

Hint: the proofs stay mostly the same. Understand that the „problem belongs to PSpace" part works analogously as before (maybe it's even a bit simpler than before). In the reduction part, try writing (one or many) automatons matching each of the four conditions: the word starts with the initial configuration; it ends with the accepting configuration; the hashes are at distance $k+1$ from each other; each cell in each following configuration depends only on the values of this cell and two neighboring cells in the previous configuration (assuming that the hashes are already at distance $k+1$ from each other). Note that directly translating the patterns in the proof into DFAs might not always work; some care/ingenuity is needed.

Exercise. Prove that for each $n \in \mathbb{N}$, there exists a set of positive patterns P and a set of negative patterns N, so that the total size of the patterns is polynomial in n, and the only word matching all patterns in P and no patterns in N is the following string of length $(n+1)2^n + 1$:

$$\#\underbrace{\texttt{0000..000}}_{\texttt{0}^n}\texttt{\#0000..001\#0000...010\#0000..011\#0000..100\#...\#1111..110\#}\underbrace{\texttt{1111..111}}_{\texttt{1}^n}\#$$

Hint.

- Write two patterns (one positive, one negative) requiring that the word starts with $\#\texttt{0}^n\#$ — that is, it contains this pattern somewhere, but it cannot contain it after the first character of the word.
- Do similarly for $\#\texttt{1}^n\#$ at the end of the word.
- Disallow two `#`'s from being in distance n or smaller from each other.
- For each `#`, require that the character exactly $n+1$ positions later (if such character should exist) is `#`. Now we have that all `#`'s are evenly spaced, partitioning the word into binary blocks of length n.
- Write incrementation rules:
  - If in a block, k-th digit is `0` and all later digits are `1`, then in the following block, k-th digit must be `1` (=it cannot be `0`). (This can be done using n negative rules.)
  - If in a block, k-th digit is `1` and all later digits are `1`, then in the following block, k-th digit must be `0`.
  - Similarly, if in a block, k-th digit is `0/1`, and there exists some later digit in that block equal to `0`, then in the following block, k-th digit is unchanged. (This can be done using $O(n^2)$ negative rules.)
- In total, you'll write $O(n^2)$ rules of total length $O(n^3)$ for which the shortest matching word has length $O(2^n \cdot n)$.