**15-455: UCT** **K. Sutner**

Assignment 7 Due: **March 30, 2025, 24:00**.

## 1. Immerman-Szelepsényi (20)

**Background**

The proof of the Immerman-Szelepsényi theorem depends critically on path guessing and the inductive, nondeterministic computation of the cardinality $|R_\ell|$ of the $\ell$th layer of reachable points, given $|R_{\ell-1}|$.

**Task**

A. Let $P_n$ be the directed path graph of length $n$, and $P'_n$ the modified version of $P_n$ where the last edge is flipped, see below for $n = 6$.


Explain why the Immerman-Szelepsényi algorithm properly handles non-reachability for $s = 1$ and $t = n$ in both $P_n$ and $P'_n$.

B. Show that for any reasonable space complexity $s(n) \geq \log n$ we have NSPACE$(s(n))$ = co-NSPACE.

**Comment** To preserve TA sanity, use the notation from lecture for part (A).

## 2. Lexicographic SAT (20)

**Background**

A satisfiable Boolean formula usually has multiple satisfying assignments, so it is natural to try to find the lexicographically smallest one. This leads to the function problem LEXSAT:

| | |
|---|---|
| Problem: | **Least** SAT (LEXSAT) |
| Instance: | A Boolean formula $\varphi(x_1, x_2, \ldots, x_m)$. |
| Solution: | The lexicographically least satisfying assignment, if it exists; No otherwise. |

It is reasonable to suspect that $\mathsf{LEXSAT} \in \mathbb{P}^{\mathsf{SAT}}$: if we can check satisfiability, we can build the least satisfying assignment with polynomial overhead.

**Task**

A. Prove that there is a polynomial time machine with oracle SAT that solves LEXSAT.

B. Come up with a decision version of LEXSAT, and some sort of hardness result.

**Comment**

A perfect answer for part (B) is tricky, just make sure your problem is harder than SAT, assuming the usual separation results.

## 3. Node Deletion (20)

**Background**

Recall the node deletion problem mentioned in class.

| | |
|---|---|
| Problem: | **Node Deletion** |
| Instance: | Two graphs $G$ and $H$, a number $k$. |
| Question: | Can one delete at most $k$ vertices from $G$ to obtain a graph that does not contain $H$ as a subgraph? |

Node Deletion appears to be outside of $\mathbb{NP}$: we can guess the nodes to be deleted, but the verification that the remaining graph $G'$ does not contain $H$ does not seem to be manageable in polynomial time ($H$ is part of the input, not fixed).

As always, we use "Node Deletion" as a shorthand for the formal language $L \subseteq \mathbf{2}^\star$ that describes the yes-instances.

**Task**

A. Write Node Deletion as a $\Sigma_2$ Boolean formula.

B. Show that Node Deletion is in $\mathbb{NP}^{\mathbb{NP}}$ by explaining how the corresponding oracle Turing machine works.

C. Explain how an alternating Turing machine would solve Node Deletion in polynomial time.

**Comment** For these question, don't get lost in the weeds, e.g., it's OK to say "there is a polynomial size Boolean formula" that does this and that.

## 4. Overhead-Free LBA (40)

**Background**

An LBA is a (possibly nondeterministic) one-tape Turing machine which is not allowed to use no more space than what is initially occupied by the input string $x \in \Sigma^\star$. Alas, we allow the Turing machine to use a tape alphabet $\Gamma$ larger than $\Sigma$, which can be used to erase or mark symbols, compress the input, open a second track, and on so on.

This type of alphabet manipulation coexists uneasily with physical realizability, it is preferable to have an alphabet that is fixed once and for all. To model this situation, we use **2** as input alphabet, plus an endmarker #. No other symbols are allowed. The initial tape inscription has the form

$$\#x_1x_2\ldots x_{n-1}x_n\#$$

where $x_i \in \Sigma$ and the endmarkers cannot be overwritten or moved. The head is positioned at, say, $x_1$. So a configuration consists of a word $w \in \Sigma^n$, plus a state and a head position. This type of machine is called an overhead-free LBA.

It is known that overhead-free LBAs cannot accept all CSLs but do accept all context-free languages, and some non-context-free ones. A full proof is too hard, but here a two manageable examples.

**Task**

1. Construct an overhead-free LBA that recognizes palindromes over the binary alphabet.
2. How does the running time of your machine compare to a standard one-tape Turing machine?
3. Construct an overhead-free LBA that recognizes all strings of the form $0^n1^n0^n$, again over the binary alphabet.