- Start as early as possible, and contact the instructor if you get stuck.
- See the course outline for details about the course's grading policy and rules on collaboration.
- Submit your completed solutions to **Crowdmark**.

1. **Collapse of the Polynomial Hierarchy** [**6=4+2 marks**]
   (a) Show that if $\Sigma_k \supseteq \Pi_k$ or $\Sigma_k \subseteq \Pi_k$ then the polynomial hierarchy collapses.

Solution

First, if $\Sigma_k \subseteq \Pi_k$ then by taking the complements,

$$\Pi_k = co\Sigma_k \subseteq co\Pi_k = \Sigma_k,$$

from class. So either containment actually gives us equality: $\Sigma_k = \Pi_k$.
Consider an arbitrary language $L \in \mathsf{PH}$. Recall that PH is the union of all Σ_i, so $L \in \Sigma_i$ for some i. If $i \leq k$ then clearly $L \in \Sigma_i \subseteq \Sigma_k$. Otherwise, suppose A is an $\underbrace{\exists\forall\exists\forall\cdots}_{i}$-verifier for the language L.

Since $i > k$ and this means that $\underbrace{\exists\forall\exists\forall\cdots}_{i}$ ends in either $\underbrace{\exists\forall\cdots}_{k}$ or $\underbrace{\forall\exists\cdots}_{k}$ depending on parity. One corresponds to Σ_k and the other to Π_k, so the idea is to exchange the sequence of quantifiers.
More specifically, if we fix the first $i-k$ certificates, then there is a Σ_k or Π_k machine M which decides the last k. That is,

$$\begin{aligned} & A \text{ accepts input } x \text{ and certificates } c_1, \ldots, c_i \\ \iff\ & M \text{ accepts input } x\#c_1\#\cdots\#c_{i-k} \text{ and certificates } c_{i-k+1}, \ldots, c_i. \end{aligned}$$

Now exchange M for a Σ_k/Π_k machine M' (the opposite of what we had before) using our hypothesis that $\Sigma_k = \Pi_k$.

$$\begin{aligned} & M \text{ accepts input } x\#c_1\#\cdots\#c_{i-k} \text{ and certificates } c_{i-k+1}, \ldots, c_i \\ \iff\ & M' \text{ accepts input } x\#c_1\#\cdots\#c_{i-k} \text{ and certificates } c_{i-k+1}, \ldots, c_i. \end{aligned}$$

Now, because we changed certificate c_{i-k+1} from $\exists/\forall$ to $\forall/\exists$, it has the same quantifier as the previous certificate. We can therefore combine the two certificates $(c_{i-k}\#c_{i-k+1})$, so there exists a Σ_{i-1} machine A' such that

$$\begin{aligned} & A' \text{ accepts input } x \text{ and certificates } c_1, c_{i-k-1}, c_{i-k}\#c_{i-k+1}, \cdots, c_i \\ \iff\ & M' \text{ accepts input } x\#c_1\#\cdots\#c_{i-k} \text{ and certificates } c_{i-k+1}, \ldots, c_i. \end{aligned}$$

Chaining together the $\iff$ equivalences, A accepts the same inputs as A', and therefore L is in Σ_{i-1}. We can repeat to show that $L \subseteq \Sigma_k$, and since L was an arbitrary language in PH, we have $\mathsf{PH} \subseteq \Sigma_k$, i.e., the polynomial hierarchy collapses.

(b) Show that if $\Sigma_i = \Sigma_j$ or $\Pi_i = \Pi_j$ for $i < j$ then the polynomial hierarchy collapses.

Solution

If $\Sigma_i = \Sigma_j$ then $\Sigma_j = \Sigma_i \subseteq \Pi_{i+1} \subseteq \Pi_j$. If $\Pi_i = \Pi_j$ then similarly $\Pi_j \subseteq \Sigma_j$. Either way, part (a) finishes the problem.

2. **Complete problems for big classes [12=4+4+4 marks]**
We introduced a definition of $\mathcal{C}$-complete languages for any class of languages $\mathcal{C}$ that contains P. In this question, we will apply that to some powerful classes (which all contain P, of course).

(a) Recall $\mathsf{E} = \mathsf{TIME}(2^{O(n)})$, or "linear exponential time". Prove that there exists an E-complete problem.

Solution

Let $A^{\mathsf{E}}_{\mathrm{TM}}$ below be the candidate problem.

$$A^{\mathsf{E}}_{\mathrm{TM}} = \{\langle M\rangle\langle t\rangle x \mid M \text{ accepts } x \text{ in } t \text{ steps}\}$$

Note that $\langle t\rangle$ is a *binary* representation rather than, say, 1^t.
First, we argue $A^{\mathsf{E}}_{\mathrm{TM}} \in \mathsf{E}$. Let T be a Turing machine which, on input $\langle M\rangle\langle t\rangle x$ simulates M for t steps on the input x. Let T reject if the simulation runs for more than t steps or if M rejects x, and otherwise accept. Clearly this correctly decides the language $A^{\mathsf{E}}_{\mathrm{TM}}$, so all that remains is running time. Even with a slow universal simulator, the running time is $\mathcal{O}(t^2)$. We assume the description of t has length $\Theta(\log t)$ and thus, for an input of length n, $t^2 = (2^{\mathcal{O}(n)})^2 = 2^{\mathcal{O}(n)}$. We conclude that $A^{\mathsf{E}}_{\mathrm{TM}} \in \mathsf{E}$.

Next, we argue $A^{\mathsf{E}}_{\mathrm{TM}}$ is E-hard. Consider an arbitrary language $L \in \mathsf{E}$ and the Turing machine T which decides it in linear exponential time. That is, there exists a constant k such that T halts on an input of length n in at most 2^{kn} steps, for all $n \geq 1$. Let h be the function

$$h(x) = \langle T\rangle\langle 2^{kn}\rangle x,$$

and note that h is easily computable in polynomial time.

(b) Recall the complexity class $\mathsf{ELEMENTARY}$.

$$\mathsf{ELEMENTARY} = \bigcup_{k\geq 1} \mathsf{TIME}\left(\underbrace{2^{2^{2^{\cdot^{\cdot^{\cdot^{n}}}}}}}_{k}\right)$$

Prove that there does not exist an $\mathsf{ELEMENTARY}$-complete problem.

Solution

Suppose L is an $\mathsf{ELEMENTARY}$-complete problem. Since $L \in$

ELEMENTARY, it belongs to

$$\mathcal{C}_k = \mathsf{TIME}\left(\underbrace{2^{2^{2^{\cdot^{\cdot^{\cdot^{n}}}}}}}_{k}\right)$$

for some fixed k. Since L is ELEMENTARY-hard, for any $K \in$ ELEMENTARY, there is a reduction $K \leq_{\mathsf{P}} L$. That means there is a polynomial-time computable function h from inputs for K to inputs for L. The polynomial runtime guarantees

$$|h(x)| \leq \mathsf{poly}(|x|) < 2^{|x|},$$

for all but finitely many $|x|$. It follows that $K \in \mathcal{C}_{k+1}$, since given input x we construct $h(x)$ and test whether it is in L in time

$$\underbrace{2^{2^{2^{\cdot^{\cdot^{\cdot^{|h(x)|}}}}}}}_{k} \leq \underbrace{2^{2^{2^{\cdot^{\cdot^{\cdot^{2^{|x|}}}}}}}}_{k} = \underbrace{2^{2^{2^{\cdot^{\cdot^{\cdot^{|x|}}}}}}}_{k+1},$$

and therefore $K \in \mathcal{C}_{k+1}$. Note that since K was arbitrary, we have ELEMENTARY $\subseteq \mathcal{C}_{k+1}$. However, $\mathcal{C}_{k+1} \subsetneq \mathcal{C}_{k+2}$ by the time hierarchy theorem—the running times are clearly separated by much more than a log factor, and it is straightforward to construct these functions. We conclude that original assumption is false and a ELEMENTARY-complete problem does not exist.

(c) Recall that RE is the set of languages recognizable by a Turing machine. Prove that there exists an RE-complete problem.

Solution

Recall from class $\mathsf{HALT}_{\mathrm{TM}}$.

$$\mathsf{HALT}_{\mathrm{TM}} = \{\langle M\rangle x \mid M \text{ halts on } x\}$$

Clearly this problem is in RE—you just simulate M on x. The simulation halts on x if and only if M halts on x.

Now we argue $\mathsf{HALT}_{\mathrm{TM}}$ is RE-complete. Let M be an arbitrary Turing machine. We reduce $\mathcal{L}(M) \leq_{\mathsf{P}} \mathsf{HALT}_{\mathrm{TM}}$ by the computable function h which maps

$$h(x) = \langle M\rangle x,$$

prepending $\langle M\rangle$ to the input. Clearly h is computable and polynomial time, and so

$$\begin{aligned} h(x) \in \mathsf{HALT}_{TM} &\iff \langle M\rangle x \in \mathsf{HALT}_{TM} \\ &\iff M \text{ halts on } x \\ &\iff x \in \mathcal{L}(M), \end{aligned}$$

finishes the reduction.

3. **Circuits and DFAs [8 marks]**
The set of regular languages, REG, is contained in LIN because we can recognize any regular language with its DFA in linear time. For any regular language L, show that you can construct a family of Boolean circuits deciding L of size $\mathcal{O}(n)$ and depth $\mathcal{O}(\log n)$.

Solution

Fix a DFA $\mathcal{A} = (\Sigma, Q, \delta, q_0, F)$. Let $\delta^*(q, x)$ denote the state we reach if we start in state $q \in Q$ and read input $x \in \Sigma^*$.
We claim that for any given DFA $\mathcal{A}$, there exists a family of Boolean circuits of size $\mathcal{O}(n)$ and depth $\mathcal{O}(\log n)$ which, given $\langle x \rangle$ (for all $x \in \Sigma^*$), output a $|Q| \times |Q|$ Boolean matrix X where

$$X_{qq'} = \begin{cases} 1 & \text{if } q' = \delta^*(q, x), \\ 0 & \text{otherwise.} \end{cases}$$

We construct this family of circuits by induction on n. If $n = 0$ then the matrix is the identity matrix, and the circuit simply needs to output a fixed value—this is no problem. If $n = 1$, then the matrix depends on the input symbol $\langle x_1 \rangle$, but there are only a constant number of values, so a constant-size circuit to compute it exists.
When $n \geq 2$, we apply divide and conquer. The input $x = x_1 \cdots x_n$ divides into $y = x_1 \cdots x_{\lfloor x/2 \rfloor}$ and $z = x_{\lfloor n/2 \rfloor + 1} \cdots x_n$. Since $|y|, |z| < |x|$, the induction hypothesis gives us circuits of linear size and log depth to compute Boolean matrices Y and Z for the two halves.
We want to output the matrix X. On input $x = yz$, the first half takes us to some state $r = \delta^*(q, y)$, and the second half takes us from r to $\delta^*(r, z)$. It follows that the formula

$$X_{qs} = \bigvee_{r \in Q} (Y_{qr} \wedge Z_{rs}) \tag{1}$$

computes entries of the matrix from the entries of the two subproblems.
What is the size/depth of this circuit? Imagine the recursion tree. Every split divides the problem in half, and thus there are at most $\lceil \log n \rceil$ splits. The circuit when $n = 1$ occurs at each leaf and has some constant size S and constant depth D. The circuit computing (1) has some constant size $\leq S'$ and constant depth $\leq D'$. The total size is the size of all leaves (nS) and the size of all internal nodes ($(n-1)S'$), which is $\mathcal{O}(n)$. The total depth is the number of levels ($\mathcal{O}(\log n)$) times the depth per level ($\max\{D, D'\} = \mathcal{O}(1)$) which is $\mathcal{O}(\log n)$ overall.

4. **Cellular automata [10=1+5+4 marks]**
Let's test the strong Church-Turing thesis by considering a new model of computation, cellular automata (CA), and showing that this model is polynomially equivalent to

Turing machines.[1]

Definition 1. A *one-dimensional cellular automaton* is a triple $\mathcal{A} = (\Gamma, b, \phi)$ where

- a finite set of states, Γ,
- a *blank state* $b \in \Gamma$,
- an *update rule* $\phi\colon \Gamma^3 \to \Gamma$,

and subject to the condition $\phi(b, b, b) = b$. A *configuration* is a function $c\colon \mathbb{Z} \to \Gamma$ such that $c(n) = b$ for all but finitely many values of $n \in \mathbb{Z}$.

We start a computation by initializing consecutive cells with the input string (much like the Turing machine, but without a tape head to worry about). Then the computation proceeds in discrete steps $c_0, c_1, c_2, \ldots$ where the configuration updates deterministically as follows:

$$c_{t+1}(n) := \phi(c_t(n-1), c_t(n), c_t(n+1)) \quad \text{for all } n \in \mathbb{Z}.$$

A *description* $\langle c \rangle$ of a configuration c is an interval $c(i)c(i+1)\cdots c(j)$ of the function which contains all non-b states occurring in c. For this question, we say the computation halts if it reaches a *stable configuration* that does not change from one step to the next.

(a) Argue that for any cellular automaton $\mathcal{A}$ as defined above, there is a Turing machine $M_{\mathcal{A}}$ which takes a configuration $\langle c_0 \rangle$ and natural number $t \in \mathbb{N}$ as input, and simulates the CA by writing $\langle c_t \rangle$ on the tape and halting. Moreover, show that $M_{\mathcal{A}}$ halts in time polynomial in the size of c_0 (number of non-blank cells) and t.

Solution

Suppose we have $\langle t \rangle$ and a configuration written on the tape. In a loop, we

i. If $t = 0$, erase the counter and halt.
ii. Otherwise, decrement t.
iii. Scan across the tape, remembering the two previous symbols $c_i(j-2)c_i(j-1)$ (initially bb at the left edge of the configuration) with the finite state, so that when we read $c_i(j)$, we can apply the rule to figure out $c_{i+1}(j-1)$ and write it the tape.
iv. When we reach the right edge of the configuration, scan back to the the counter, and repeat from the start.

(b) In the other direction, show the following. For any Turing machine M, there exists a CA $\mathcal{A}_M$ such that for any input $x \in \{0,1\}^*$, the CA reaches the all-blank configuration from x if and only if M accepts x. Moreover, if it takes M t steps to halt then the CA reaches the all-blank configuration in $\mathcal{O}(t)$ timesteps.

Hint: It might be helpful to recall a result from the midterm: For any Turing machine M, there exists a Turing machine M' which accepts/rejects/loops precisely when M does, but also cleans up its tape if it halts.

[1] If you have seen cellular automata before, you may have encountered definitions which were, e.g., limited to binary states, fixed to two dimensions, or where cells are updated based on counts of their neighbours. The definition here is not like these examples, so read it carefully.

Correction: Per Piazza, you are allowed to add a short string on either side of the input bit string in the initialization.

Solution

Per the hint, we first modify the TM to clean up its tape before it halts. Let the states for the CA include

- the tape alphabet Γ of the TM, which includes $\{\texttt{0}, \texttt{1}\}$ and the TM blank symbol, $\square$,
- $Q \times \Gamma$ to represent the tape head, and
- a new delimiter \$ to mark the left edge of the input.

We'll reuse the blank symbol from the TM, i.e., $b = \square$. Recall that the initial configuration is a string $\$x$ (for $x \in \Gamma \backslash \{\square\}$) starting at cell 0, with all other cells initialized to $\square$.

We define the update rule $\phi(x, y, z)$ of the CA is as follows:

i. If $x = \$$ and $y \in \Gamma$ then output (y, q_0).
ii. If $y = \$$ and $z \in \Gamma$ then output $\square$.
iii. If exactly one of x, y, z is from $(Q \backslash F) \times \Gamma$, compute the transition δ on that symbol and state, and the transition writes $w \in \Gamma$, goes to state $q' \in Q$, and moves in direction $d \in \{L, R\}$.
   - If $y = (q, y')$ then output w, since this symbol will be overwritten and the tape head will move elsewhere.
   - If $x = (q, x')$ and $d = R$ then output (q', y), since the tape head moves here and the tape symbol remains the same. If $d = L$ then output y instead, since the tape head moved the other direction.
   - If $z = (q, z')$ and $d = L$ then output (q', y), otherwise output z, similar to the previous case.
iv. If $x = \square$, $y = (q_{\text{halt}}, \square)$, $z = \square$, then output $\square$.
v. Excluding the cases above, maintain the previous value by outputting y.

Since the initial configuration does not include a tape head, the first two cases above create a tape head in state q_0 on the leftmost symbol of the input. Aside from the first step, which only constructs this tape head, we claim the CA updates the configuration at a rate of one CA step per TM step, leading to $\mathcal{O}(t)$ runtime.

We saw in class (most recently, during the Cook-Levin theorem) that only a small neighbourhood around the tape head changes from one step to the next. Case (iii) of the update rule implements this change. If the tape head is above, then the tape symbol will change and the head will move somewhere else. If the tape head is immediately to one side, then depending on the transition (which we have enough information available to compute) the tape head will move into this cell and join with the current tape symbol, or it will move the other direction, and this symbol remains unchanged.

If the TM halts, we know the entire tape is blank ($\square$) when it halts. This means the configuration is $\cdots\square(q_{\text{halt}}, \square)\square\cdots$, and case (iv) turns the last

symbol into $\square$.

(c) Design a cellular automaton which beats the single tape Turing machine by deciding

$$\mathsf{PALINDROME} = \{x \in \{0,1\}^* \mid x = x^R\},$$

in linear time. Specifically, if x is a palindrome then it should accept by halting in the all-blank configuration, and reject by halting in any other configuration. Justify the correctness and runtime of your CA.

Note: The Piazza correction applies here too.

Solution

The intention for this problem was for students design a bespoke CA for PALINDROME, not design a TM and go through part (b). This was just about the simplest problem I could imagine.

Let's have our CA be over states

$$\Gamma = \{0, 1, \blacktriangleleft, \blacktriangleright, \overline{0}, \overline{1}, \square\}.$$

We start in the configuration $\blacktriangleright x \blacktriangleleft$ where $x \in \{0,1\}^*$ is the input, and there are blanks everywhere else.

The rules are as follows. In the following rules, $*$ represents a symbol we don't care about (space, bit, $\blacktriangleleft$, whatever). The first four rules let a $\blacktriangleright$ move right, exchanging with bits as it goes.

$$\begin{array}{llll} \blacktriangleright 0 * & \to & \blacktriangleright & \qquad \blacktriangleright 1 * \to \blacktriangleright \\ * \blacktriangleright 0 & \to & 0 & \qquad * \blacktriangleright 1 \to 1 \end{array}$$

Next, we have mirrored rules for $\blacktriangleleft$, but now the bits $0, 1$ are replaced with "anti-bits" $\overline{0}$, $\overline{1}$.

$$\begin{array}{llll} a\, 0 \blacktriangleleft & \to & \blacktriangleleft & \qquad a\, 1 \blacktriangleleft \to \blacktriangleleft \\ a \blacktriangleright 0 & \to & \overline{0} & \qquad a \blacktriangleright 1 \to \overline{1} \end{array}$$

At some point, the $\blacktriangleright$ and $\blacktriangleleft$ meet in the middle. Depending on the parity of the length, they could meet in any of the following configurations. In any case, we delete them.

$$\begin{array}{llll} \blacktriangleright \blacktriangleleft * & \to & \square & \qquad * \blacktriangleright \blacktriangleleft \to \square \\ \blacktriangleright * \blacktriangleleft & \to & \square & \end{array}$$

When spaces are created, we want the anti-bits to move left into that space.

$$\begin{array}{llll} * \square \overline{0} & \to & \overline{0} & \qquad \square \overline{0} * \to \square \\ * \square \overline{1} & \to & \overline{1} & \qquad \square \overline{1} * \to \square \end{array}$$

To avoid complication, the blue bits will remain stationary. Where a bit and anti-bit of the same value meet, we let them "annihilate" into a $\square$.

$$\begin{array}{lcl@{\qquad\qquad}lcl} *\,0\,\overline{0} & \to & \square & 0\,\overline{0}\,* & \to & \square \\ *\,1\,\overline{1} & \to & \square & 1\,\overline{1}\,* & \to & \square. \end{array}$$

In all other cases, we preserve the middle symbol.
A typical accepting run goes something like this:

$$\begin{array}{ccccccccccc} \cdots & \square & \blacktriangleright & 0 & 1 & 1 & 0 & \blacktriangleleft & \square & \cdots \\ \cdots & \square & 0 & \blacktriangleright & 1 & 1 & \blacktriangleleft & \overline{0} & \square & \cdots \\ \cdots & \square & 0 & 1 & \blacktriangleright & \blacktriangleleft & \overline{1} & \overline{0} & \square & \cdots \\ \cdots & \square & 0 & 1 & \square & \square & \overline{1} & \overline{0} & \square & \cdots \\ \cdots & \square & 0 & 1 & \square & \overline{1} & \square & \overline{0} & \square & \cdots \\ \cdots & \square & 0 & 1 & \overline{1} & \square & \overline{0} & \square & \square & \cdots \\ \cdots & \square & 0 & \square & \square & \overline{0} & \square & \square & \square & \cdots \\ \cdots & \square & 0 & \square & \overline{0} & \square & \square & \square & \square & \cdots \\ \cdots & \square & \square & \square & \square & \square & \square & \square & \square & \cdots \end{array}$$

Whereas a rejecting run goes something like this:

$$\begin{array}{ccccccccc} \cdots & \square & \blacktriangleright & 0 & 1 & 1 & \blacktriangleleft & \square & \cdots \\ \cdots & \square & 0 & \blacktriangleright & 1 & \blacktriangleleft & \overline{1} & \square & \cdots \\ \cdots & \square & 0 & 1 & \square & \overline{1} & \overline{1} & \square & \cdots \\ \cdots & \square & 0 & 1 & \overline{1} & \square & \overline{1} & \square & \cdots \\ \cdots & \square & 0 & \square & \square & \overline{1} & \square & \square & \cdots \\ \cdots & \square & 0 & \square & \overline{1} & \square & \square & \square & \cdots \\ \cdots & \square & 0 & \overline{1} & \square & \square & \square & \square & \cdots \\ \cdots & \square & 0 & \overline{1} & \square & \square & \square & \square & \cdots \end{array}$$

At this point the automaton is stuck.
As you can see, the idea is that the triangles move to the middle (taking around $\frac{n}{2}$ steps), and cancel out. Along the way, the bits on the right half are converted to "anti-bits"; $0 \to \overline{0}$, $1 \to \overline{1}$. The anti-bits then travel right by exchanging with blanks, although it takes a further $\frac{n}{2}$ (approx.) steps until the rightmost anti-bit moves. Assuming the word is a palindrome, the bits and anti-bits annihilate in the middle, being replaced by blanks. The rightmost anti-bit takes about n steps to meet the leftmost bit, so in the case $x \in \mathsf{PALINDROME}$, it takes around $2n = \mathcal{O}(n)$ steps to halt.