Student Information

Full Name : Beyazit Yalcinkaya
Id Number : 2172138

Answer 1

a.

No, it does not imply that A is a RL. We give a counter example. Suppose that $A = \{1^n 0^n \mid n \geq 0\}$ and $B = 1^*$. Clearly, A is a CFL and B is a RL. We can define a computable function (in fact, it is a bijection) $f : A \mapsto B$ as follows: for $w \in A$ and $f(w) \in B$, $|f(w)| = |w|/2$, that is each element of A is mapped to an element of B with same number of 1s. Clearly, f is a bijection. This counter example shows that if $A \leq_m B$, then A is not necessarily a RL.

b.

An example is A_{TM}. To show that $A_{TM} \leq_m \overline{A_{TM}}$, we define a computable function f that takes input of the form $\langle M, w\rangle$ and returns output of the form $\langle M', w'\rangle$, where $\langle M, w\rangle \in A_{TM}$ if and only if $\langle M', w'\rangle \in \overline{A_{TM}}$. The following machine F computes a reduction f.

F = "On input $\langle M, w\rangle$:
1. Construct the following machine M'.
2. M' = "On input x:
    1. Run M on x
    2. If M accepts, then `REJECT`
    3. If M rejects, then `ACCEPT`"
3. Return $\langle M', w\rangle$."

c.

To show the undecidability of W_{TM}, we will show that $A_{TM} \leq_m W_{TM}$ by defining a computable function f that takes input of the form $\langle M, w\rangle$ and returns output of the form $\langle M'\rangle$, where $\langle M, w\rangle \in A_{TM}$ if and only if $\langle M'\rangle \in W_{TM}$. The following machine F computes a reduction f.

F = "On input $\langle M, w\rangle$:
1. Construct the following machine M'.
2. M' = "On input x:
    1. If x is not the empty string, then `REJECT`.
    2. If x is the empty string, then run M on w such that before writing an a to the tape always write b first and then override that b with an a.
    3. If M accepts, then write a and move right three times and `ACCEPT`.
    4. If M rejects, then `REJECT`."
3. Return $\langle M'\rangle$."

d.

ALL_{TM} is a language of TM descriptions. It satisfies two conditions of Rice's Theorem: (i) it is non-trivial since there are TMs that do not accept Σ^* and (ii) if two TMs recognize the same language, either both have descriptions in ALL_{TM} or neither do. Hence, by Rice's theorem, we conclude that ALL_{TM} is undecidable.

e.

1. If A is decidable, then $A \leq_m 0^*1^*$. For this part of the proof, we will define a computable function $f : A \mapsto 0^*1^*$.
$$f(x) = \begin{cases} 01 & \text{if } x \in A \\ 10 & \text{if } x \notin A \end{cases}$$
2. If $A \leq_m 0^*1^*$, then A is decidable. Notice that, 0^*1^* is trivially decidable since it is a RL and there exists a DFA deciding it. If $A \leq_m 0^*1^*$, then there exists a computable function $f : A \mapsto 0^*1^*$. We can construct a TM deciding A as follow: using f convert input string to a string in 0^*1^*, run DFA for 0^*1^* on this string, if it accepts accept; if it rejects reject.

Answer 2

$$L = \{\langle M, w\rangle \mid M \text{ is a TM that moves its head left when its head is on the left-most tape cell when started on } w\}$$

L is a language of TM descriptions. It satisfies two conditions of Rice's Theorem: (i) it is non-trivial since there are TMs that do not move their heads left when their heads are on the left-most tape cell when started on w and (ii) if two TMs recognize the same language, either both have descriptions in L or neither do. Hence, by Rice's theorem, we conclude that L is undecidable.

Answer 3

First, we encode the problem mathematically. A card c_i from a given sequence of cards is defined as an ordered pair $c_i = (C_0^i, C_1^i)$, where C_0^i and C_1^i are columns of the card and for convenience, we defined $c_i = (C_0^i, C_1^i) = (C_{\neg 1}^i, C_{\neg 0}^i)$. A column C is defined as a set of not-punched indexes $\{f_1, f_2, \ldots, f_m\}$ where hole positions are indexed from 1 to N so $f_i \in [1, N]$ for $i = 1, 2, \ldots, m$ for some m. A given sequence of cards $\langle c_1, c_2, \ldots, c_k\rangle$ is a solution if and only if an ordered pair of sequences $(S_1 = \langle C_{x_1}^1, C_{x_2}^2, \ldots, C_{x_k}^k\rangle, S_2 = \langle C_{\neg x_1}^1, C_{\neg x_2}^2, \ldots, C_{\neg x_k}^k\rangle)$, where $x_i \in \{0, 1\}$ for $i = 1, 2, \ldots, k$, satisfying Eqn. (1) exists. Notice that, a set does not have repetition of elements and taking union of several sets return gain a set without repetitive elements.

$$\left|\bigcup_{i=1}^{k} C_{x_i}^i\right| = \left|\bigcup_{i=1}^{k} C_{\neg x_i}^i\right| = N \tag{1}$$

To prove that $PUZZLE$ is NP-complete (1) we show $PUZZLE$ is in NP and (2) we prove that all languages in NP are polynomial time reducible to $PUZZLE$, i.e., $PUZZLE$ is NP-hard, by giving a polynomial time mapping reduction from $3SAT$ to $PUZZLE$.

(1) To show that $PUZZLE$ is in NP, we describe a NTM M_{PUZZLE} deciding $PUZZLE$.

M_{PUZZLE} = "On input $\langle c_1, c_2, \ldots, c_k \rangle$, where each c_i is a card described as above:

1. Nondeterministically generate a bit string w such that $|w| = k$. Use w to generate an ordered pair of sequences $(S_1 = \langle C^1_{w_1}, C^2_{w_2}, \ldots, C^k_{w_k} \rangle, S_2 = \langle C^1_{\neg w_1}, C^2_{\neg w_2}, \ldots, C^k_{\neg w_k} \rangle)$.
2. Check if this ordered pair of sequences satisfies Eqn. (1).
3. If Eqn. (1) is satisfied, return `ACCEPT`; otherwise, return `REJECT`."

It takes $O(k)$ steps to randomly generate a bit string of length k in stage 1, checking Eqn. (1) in stage 2 takes again $O(k)$ step, and stage 3 is simply $O(1)$. Hence, M_{PUZZLE} operates in polynomial time. Since it is a nondeterministic TM, $PUZZLE$ is in NP. Notice that, we could have constructed a polynomial time verifier to $PUZZLE$ as well, i.e., just disregard stage 1 and the bit string as input an execute stage 2 and 3. It is just another way to show that $PUZZLE$ is in NP.

(2) We show that $3SAT \leq_P PUZZLE$ by giving an algorithm for mapping $3SAT$ to $PUZZLE$. The gadgets for variables are cards and the gadgets for clauses are the rows of holes on the cards. Given a formula with k variables and m clauses, construct k cards with m rows of hole positions on both columns. If a variable appears as positive in a clause, fill hole in the first column and corresponding row. If a variable appears as negative in a clause, fill hole in the second column and corresponding row. We also add and additional card to this construction. This card has its second column completely filled and its first column full of holes. Notice that this special card is never flipped.

Below, we give the algorithm formally. For convenience, we define followings: for a CNF formula ϕ and a sequence of cards $\langle c_1, c_2, \ldots, c_k \rangle$, $\mathsf{variables}$ is the set of variables, $\mathsf{clauses}$ is the set of clauses in ϕ, and cards is the set of cards in $\langle c_1, c_2, \ldots, c_k \rangle$. Let $|\mathsf{variables}| = |\mathsf{cards}| = k$ and $|\mathsf{clauses}| = m$, then for $1 \leq i \leq k$, $\mathsf{variables}[i]$ is the literal with index i and $\mathsf{cards}[i]$ is the card with index i and for $1 \leq j \leq m$, $\mathsf{clauses}[j]$ is the clause with index j. We also refer first column of card with index i as $\mathsf{cards}[i][0]$ and second column as $\mathsf{cards}[i][1]$.

A = "On input $\langle \phi \rangle$:

1. For $i = 1$ to k
2.   For $j = 1$ to m:
3.   If $\mathsf{variables}[i] \in \mathsf{clauses}[j]$:
4.   If $\mathsf{variables}[i]$ is a positive literal: $\mathsf{cards}[i][0] = \mathsf{cards}[i][0] \cup \{j\}$.
5.   Else: $\mathsf{cards}[i][1] = \mathsf{cards}[i][1] \cup \{j\}$.
6. $\mathsf{cards} = \mathsf{cards} \cup \{(\emptyset, \{1, 2, \ldots, m\})\}$.
7. Return cards."

Notice that, stage 6 adds the special card that has its second column completely filled and its first column full of holes. This special card is never flipped. Let's give an example for this mapping. Given $(x_1 \vee x_2) \wedge (\overline{x_1} \vee \overline{x_2})$, algorithm given above, generates two cards. The first one has its holes filled in the cells with indexes $[1, 1]$ and $[2, 2]$. The second card has its holes filled in the same cell positions as well. These cards are a solution when the first card is placed as it is and the second one is placed after flipping. This solution corresponds to assigning x_1 to be true and x_2 to be false which satisfies given formula.

Now, we prove that this reduction works by showing ϕ is satisfiable if and only if cards is a solution. We start with a satisfying assignment. This assignment can be view as a bit string w, e.g., if x_1 is true then the first bit of w is 1. Then w can be used to refer which sides of cards to be used, i.e., we can construct an ordered pair of sequences $(S_1 = \langle C^1_{w_1}, C^2_{w_2}, \ldots, C^k_{w_k}\rangle, S_2 = \langle C^1_{\neg w_1}, C^2_{\neg w_2}, \ldots, C^k_{\neg w_k}\rangle)$. Since we add the special card that has its second column completely filled and its first column full of holes, the second column is always filled. By our construction, the first column filled when all clauses are true due to some positive literals being true or some negative literals being true, i.e., some variables being true or false. Hence, cards is a solution.

We continue cards being a solution. As showed above, using orientation of cards we can refer to a bit string w and use this bit string to refer an assignment for variables in ϕ. By construction, since all holes are filled by some cards, all clauses are satisfies by at least one variable. Then, since all clauses are satisfied the whole formula is also satisfied. Hence, ϕ is satisfied by this assignment.

Finally, we show that the reduction can be carried out in polynomial time. Loop starting from stage 1 executes $O(k)$ time. Loop starting from stage 1 executes $O(m)$ time. Stage 3 is $O(1)$ since each clause has at most 3 literals. Stages 4, 5, 6, and 7 are all $O(1)$ operations. Then, the total time complexity of the reduction is $O(km)$ which is polynomial. Hence, $PUZZLE$ is NP-complete.

Answer 4

a.

Below, we give a description of a TM M that decides $SPATH$ in polynomial time. Note that, for the sake of simplicity in the presentation, we define $MARK[n]$ to return the marking of node n.

M = "On input $\langle G, a, b, k\rangle$, where G is an undirected graph with nodes a and b and k is a natural number:

1. $MARK[a] = 0$.
2. Repeat the following until no additional nodes are marked:
3. Scan all the edges of G. If an edge (x, y) is found between a marked node x and an unmarked node y, $MARK[y] = MARK[x] + 1$.
4. Scan all the edges of G. If an edge (x, y) is found between an unmarked node x and a marked node y, $MARK[x] = MARK[y] + 1$.
5. If $MARK[b]$ is not defined, return `REJECT`.
6. If $MARK[b]$ is defined and $MARK[b]$ is more than k, return `REJECT`.
7. If $MARK[b]$ is defined and $MARK[b]$ is less than or equal to k, return `ACCEPT`."

We analyze this algorithm to show that it runs in polynomial time. Clearly, stages 1, 5, 6, and 7 are executed only once and they are $O(1)$ operations. Stages 3 and 4 are executed at most m times where m is the edges and they both are $O(m)$ operations since they require a scan of the edges in G. The overall complexity is $O(m^2)$ which is a polynomial in the size of G, i.e., $SPATH \in P$.

b.

We show that $UHAMPATH \leq_P LPATH$ by defining a TM F computing a function $f : UHAMPATH \mapsto LPATH$.

F = "On input $\langle G = (V, E), s, t\rangle$:

1. Return $\langle G = (V, E), s, t, |V|\rangle$."

F simply operates on $O(1)$ time, i.e., polynomial time. By using F, an input for $UHAMPATH$ can be converted to an inout of $LPATH$ in constant time. There for $LPATH$ is NP-complete.

Answer 5

First, we show that U is in NP by constructing a verifier V. Notice that, a certificate for $\langle M, x, \#^t \rangle$ is a branch of M accepting input x within t steps. Define a branch b as a sequence of configurations of M on x and $\langle M, x, \#^t \rangle \in U$ if and only if b is a valid sequence of configurations and b contains $t+1$ many configurations. A sequence of configurations is valid if it starts in the start state, ends in a halting state, all consecutive configurations follow one another by applying the transition function, and b is a sequence of configurations for the input x. b should contain $t+1$ configuration because it implies that t many steps are taken.

V = "On input $(\langle M, x, \#^t \rangle, b)$:

1. Check if b is a valid sequence of configurations
2. Check if b contains $t+1$ configurations.
3. If both checks are satisfied, return `ACCEPT`; else return `REJECT`."

Both stage 1 and 2 takes $O(t)$ time and stage 3 takes $O(1)$ time. Therefore, V is a polynomial time verifier for U. Then, we can conclude that U is in NP.

Second, we show that U is NP-hard. We know that for every L in NP, there exists a NTM accepting strings of L in polynomial time in the length of input. Then, we can do the following to map every language in NP to U. For any L in NP, there exists a NTM M such that $L(M) = L$. For any input $x \in L$, M executes in polynomial time on x, call this polynomial function p. Then, we can construct $W = \langle M, x, \#^p \rangle$. Obviously, $x \in L$ if and only if $W \in W$. Therefore, U is NP-hard. Then, we can conclude that U is NP-complete.

Answer 6

Assuming that P=NP. Let $A \in$ P=NP such that $A \neq \emptyset$ and $A \neq \Sigma^*$. To show that A is NP-complete, first, we show that there exists a polynomial time verifier for A and second, every language in P=NP is polynomial time reducible to A. The first is trivial by our assumption, i.e., there even exists a polynomial time decider for A. For the second, we show that any language in P=NP is polynomial time reducible to A. Pick an arbitrary language $B \in$ P=NP, below, a TM F computing a polynomial time mapping reduction function $f : B \mapsto A$ is given. Notice that, since $A \neq \emptyset$ and $A \neq \Sigma^*$, there is a string $w_{in} \in A$ and a string $w_{out} \notin A$.

F = "On input w:

1. Run the decider for B on w.
2. If it accepts, return w_{in}; otherwise return w_{out}."

Since $B \in$ P=NP, stage 1 takes polynomial time and stage 2 is simply $O(1)$. Then, f is a polynomial time many-to-one mapping reduction from B to A. Therefore, A is NP-complete.

Bonus

1

a.

Below, we give a decider M_{TSTMTS} for $TSTMTS$. Notice that, M_{TSTMTS} always halts and it is indeed a decider because of the assumptions given in the problem.

M_{TSTMTS} = "On input $\langle T \rangle$:

For each possible string P:
    Run M^* on $\langle P, T \rangle$.
    If M^* accepts:
        `ACCEPT`.
    If M^* rejects:
        Run M^* on $\langle P, \neg T \rangle$.
        If M^* accepts:
            `REJECT`.
        If M^* rejects:
            Continue loop."

b.

Below, we give a decider M_{HALT} for the halting problem.

M_{HALT} = "On input $\langle M, w \rangle$:

Construct a statement T :"M halts on w"
Run M_{TSTMTS} on $\langle T \rangle$.
    If M_{TSTMTS} accepts:
        `ACCEPT`.
    If M^* rejects:
        `REJECT`."

c.

Since halting problem is proved to be undecidable, M_{HALT} does not exists and we end up with a contradiction. We discharged the initial assumption that every true statement T_i has a proof P_i. That is, there exists a true theorem that does not have a proof; hence, proving Gödel's Theorem.

2

This problem is proved in [1], here, we present the main idea behind the proof.

The first part of the proof provides an upper bound on the length of the sequence of configurations (referred as crossing sequence) as follows. Suppose that M is an s (≥ 2) state one-tape TM accepting a set L within time $f(n)$. Let $g(n)$ be defined by

$$g(n) = \begin{cases} \frac{nlogn}{f(n)} & \text{if } n \geq 2 \\ 1 & \text{if } n = 0, 1 \end{cases}$$

Then we have

$$\lim_{n \to \infty} g(n) = 0$$

and their is an upper bound c on the length of any crossing sequence of M for any input $x \in L$ with $|x| \geq 2$. By [2], there exists a DFA that accepts L.

Next part of the proof justifies the result by showing that there is no $x \in L$ with $|x| \geq 2$ such that M generates a crossing sequence of length larger than the upper bound c in accepting x.

Basically overall structure of the proof is follows. First, it is showed that for a language $L \in TIME(f(n))$, sequence of configurations is upper bounded by a constant, i.e., it is $O(1)$. Then, this result leads to the existence of a DFA deciding L which implies that L is a RL.

3

I think that P $\neq$ NP. For languages in NP, we have polynomial time verifiers. The exponential growth comes from the search for a certificate to pass to the verifier whereas it is not the case for the languages in P, i.e., it does not take exponential time to search. I think the main reason behind this difference is that for problems in P, either we can construct a solution with a polynomial time algorithm or we have a deeper understanding of the problem and of the search space. Then, we can prune big portions of the search space and try to find the solution in a polynomially big search space. On the other hand, for problems in NP, most of the time, we have to check every possible solution and it is basically not possible to come up with a smart way to do this search faster. The reason is that it is equally likely to find the solution in every possibility in the search space regardless of the depth of our understanding of the problem. Therefore, it seems that classes P and NP have different notions behind them. An example of this difference in their nature can be languages $SPATH$ and $LPATH$ from question 4. $SPATH$ declares an upper bound on the path length so it basically limits the search space and it is a more relaxed goal to achieve while looking for a solution. Such a relaxed goal can be achieved by only checking the edges of the graph once. However, on the other hand, $LPATH$ declares a lower bound on the path length which requires an exhaustive enumeration of all the possibilities before declaring a result. This difference between these two languages is a good example to illustrate the difference between classes P and NP.

References

[1] Kobayashi, Kojiro. "On the structure of one-tape nondeterministic Turing machine time hierarchy." Theoretical computer science 40 (1985): 175-193.

[2] Hennie, Fred C. "One-tape, off-line Turing machine computations." Information and Control 8.6 (1965): 553-578.