Problem

Show that for any function $f: \mathcal{N} \longrightarrow \mathcal{R}^{+}$, where $f(n) \geq n$, the space complexity class SPACE(f(n)) is the same whether you define the class by using the single tape TM model or the two-tape read-only input TM model.

Step-by-step solution

Step 1 of 3

$\mathrm{SPACE}(f(n))$: Let $f: N \to R^{+}$ be a function. The space completely class $\mathrm{SPACE}(f(n))$ is defined as follows:

$$\mathrm{SPACE}(f(n)) = \left\{ \begin{array}{l} L \mid L \text{ is a language decided by an } \mathrm{O}(f(n)) \text{ space} \\ \text{deterministic Turning machine} \end{array} \right\}$$

Now we have to prove that, class $\mathrm{SPACE}(f(n))$ is same whether we define z by using the single tape *TM* or two tape read-only input *TM* model.

Comment

Step 2 of 3

To prove this, we have to do the following two things

(i) We have to simulate single – tape *TM* on two tape read – only *TM* and

(ii) We have to simulate two tape read only *TM* on single – tape *TM*.

(i) **Simulation of single – tape *TM* on two tape read – only *TM:***

1. First we have to scan the read only tape.

2. Contents of the read only tape are copied to the work tape.

3. Remainder of the contents is simulated by treating the read l write work – tape as the input tape of a single – tape *TM*.

• Clearly only $\log n$ space is used to copy the contents.

• Since $f(n) \geq n$, we can write all of the input on are work – tape

In this way we simulated single-tape *TM* on two tape read only *TM*.

Comment

Step 3 of 3

(ii) **Simulation two tape read only *TM* on single – tape *TM:***

• There is no participation from single tape TM; we are working over the first input symbols, using remainder of the tape as our work area.

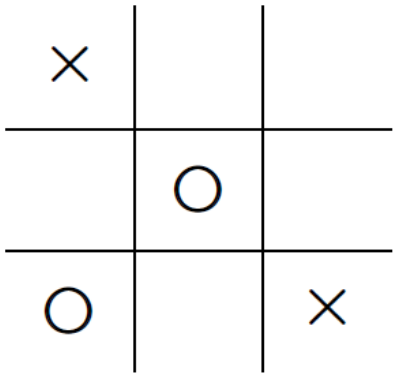
• For n symbols, add n amount of space. By increasing space we add constant to n.

So from (i) and (ii) simulations we proved that class $SPACE(f(n))$ is same whether we defined it by using the single tape *TM* or two – tape read _ only input *TM*.

Comment

Problem

Consider the following position in the standard tic-tac-toe game.



Let's say that it is the ×-player's turn to move next. Describe a winning strategy for this player. (Recall that a winning strategy isn't merely the best move to make in the current position. It also includes all the responses that this player must make in order to win, however the opponent moves.)

Step-by-step solution

Step 1 of 3

Tic- tac – toe game:

Game Constraints:

• This game is played by two players X and O, who take turns marking the spaces in a 3×3 grid.

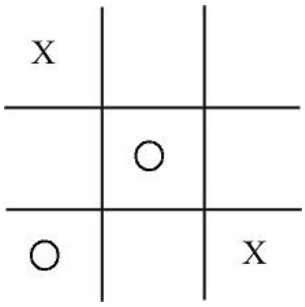
• Any X or O player goes first.

• **Winning the game:** who will first mark three places of vertical, horizontal or diagonal rows.

Comment

Step 2 of 3

Consider X moved first in following problem.



Winning strategy for the game:

The winning strategy of the game corresponds to quantified statement because games are closely related to quantifiers. These quantified statements help in understanding the complexity of the game.

Comment

Step 3 of 3

Consider the quantified Boolean formula in prenex normal form, $\phi = \exists x_1 \forall x_2 \exists x_3 ... Q x_k [\psi]$. Here, *Q* is either $\forall$ or $\exists$. Two players called *X* and *O* selects the gr in the tic-tac-toe in their respective turn. Here, $x_1, x_2, x_3, \cdots$ are the grids in the tic-tac-toe. At this position of the ga e, three grids decide the winner.

The player *X* selects the grids that are bound to $\exists$ quantifier and player *O* selects the grids that are bound to $\forall$ quantifier. In this problem, player *X* starts the game. So, the formula starts with $\exists$ quantifier. The player *X* will win the game if ψ is TRUE.

The formula is as follows:

$$\phi = \exists x_1 \forall x_2 \exists x_3 [(x_1 \vee x_2) \wedge (x_2 \vee x_3) \wedge (\overline{x_2} \vee \overline{x_3})]$$

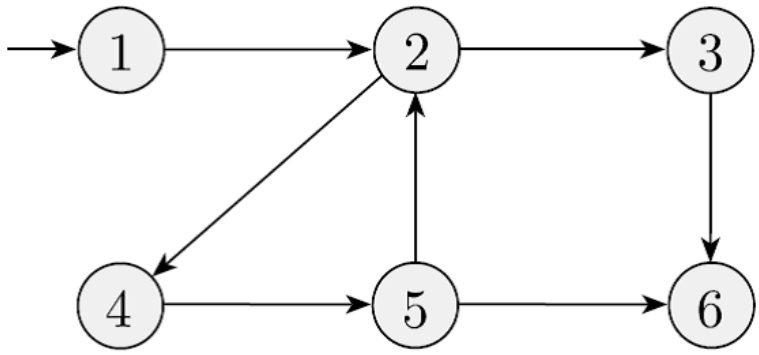
In the above formula, x_1 denotes the grid selected by player *X*, x_2 denotes the grid selected by player *O* and x_3 denotes the grid selected by player *X*. The player *X* always win the game if the third grid in the first row is selected (Consider it as 1 i.e., $x_1 = 1$)) and then selecting x_3 to be the negation of whatever player *O* selects.

The formula will become true, if the third grid in the first row is selected by player *X* then player *O* selects either third grid in the second row or the second grid in the first row or the first grid in the second row or the second grid in the third row. The player *X* can win the game if the second grid in the first row or the third grid in the second row is selected. Depending on the player *O*'s move, player *X* will select any of these grids and wins the game.

Comment

Problem

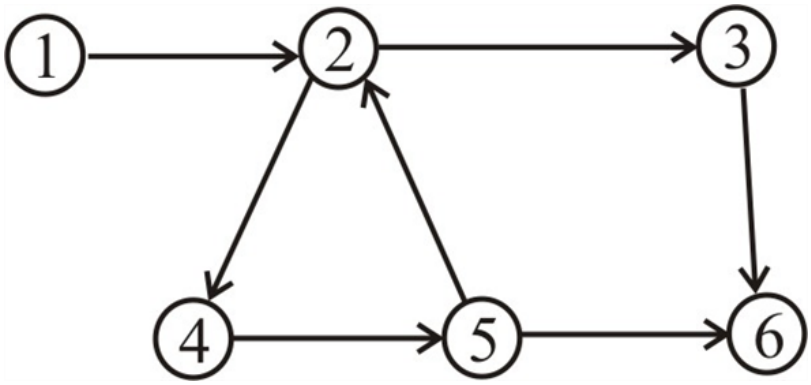
Consider the following generalized geography game wherein the start node is the one with the arrow pointing in from nowhere. Does Player I have a winning strategy? Does Player II? Give reasons for your answers.



Step-by-step solution

Step 1 of 3

Given generalized geography game is



Comment

Step 2 of 3

Geography is a game contains nodes connecting. When one player begins at one node, another player continues with connecting the same nodes. If who will get stuck first i.e., there is no connection of node that player will lost the game.

For given problem, say that player I is the one who moves first and player II second.

Comment

Step 3 of 3

Player I has a wining strategy as follows:

• Player I starts at node I, the designated start node.

• Node 1 points only at node 2, so player I must select node 2.

• Now Node 2 points at nodes 3 and 4.

• So player II must choose one of these two choices. He chooses node 3.

• Then player I must select 6, which doesn't point to any node.

• So player II is stuck, and thus player I win.

Player II has a wining strategy as follows:

• Player II starts at node 1, the designated start node.

• Node 1 points only at node 2, so player I must select node 2.

• Node 2 point at nodes 3 and 4.

• So player II must choose one of these two choices. He chooses node 4.

• Then player I must select node 5.

• Node 5 points at nodes 2 and 6.

• As node 2 already chosen, so player II must select node 6, which doesn't point to any node.

• So player I get stuck and thus player II wins.

Comment

Problem

Show that PSPACE is closed under the operations union, complementation, and star.

Step-by-step solution

Step 1 of 4

PSPACE is the class of languages that are decidable in polynomial space on a deterministic Turing machine i.e. $\text{PSPACE} = \bigcup_k \text{SPACE}\left(n^k\right)$.

Consider the two languages L_1 and L_2 that are decided by PSPACE Turing machines M_1 and M_2.

M_1 decides L_1 in deterministic time $O\left(n^k\right)$ and M_2 decides L_2 in deterministic time $O\left(n^L\right)$.

If any polynomial solvable in polynomial time, then it is solvable in polynomial space.

Comment

Step 2 of 4

UNION:

$M =$"on input w:

1. Run M_1 on w, if M_1 accepted then accept

2. Else run M_2 on w, if M_2 accepted then accept

3. Else reject"

Clearly, the longest branch in any computation tree on input w of length *n* is $O\left(n^{\max\{k,L\}}\right)$. Thus, M is a polynomial time deterministic decider for $L_1 \cup L_2$. If any polynomial solvable in polynomial time, then it is solvable in polynomial space. Therefore, PSPACE is closed under union.

Comment

Step 3 of 4

COMPLEMENTATION:

$M =$"on input w:

1. Run M_1 on w, if M_1 accepted then reject.

2. Else accept"

Clearly, the longest branch in any computation tree on input w of length n is $O\left(n^{\{k\}}\right)$. Thus, M is a polynomial time deterministic decider for $\overline{L_1}$. If any polynomial solvable in polynomial time, then it is solvable in polynomial space. Therefore, PSPACE is closed under complementation.

Comment

Step 4 of 4

STAR:

$M =$"on input w:

1. If $w = \varepsilon$ then accept

2. Deterministically select a number m such that $1 \le m \le |w|$

3. Deterministically split w into m pieces such that $w = w_1 w_2 ... w_m$.

4. For all *i,* $1 \le i \le m$: run M_1 on w_i, if M_1 rejected then reject.

5. Else (M_1 accepted all w_i, $1 \le i \le m$), accept".

The steps 1 and 2 takes $O(m)$ time. Step 3 also possible in polynomial time. In step 4, the for loop is run at most m times and every run takes almost $O(m^k)$. The total times is $O(m^{k+1})$. This means that M is a polynomial time decides for L_1^*. If any polynomial solvable in polynomial time, then it is solvable in polynomial space. Therefore, PSPACE is closed under star.

Therefore, PSPACE is closed under Union, Complementation and star.

Comment

Problem

Show that A_{DFA} ? L.

Step-by-step solution

Step 1 of 1

2287-8-7E AID: 1134 I 24/04/2012

RID: 48 I 18/05/2012

Class *L* : In deterministic Turing machine, L can be defined as class of languages complexity decidable in logarithmic space *i.e.* $L = SPACE(\log n)$

Now we need to prove that $A_{DFA} \in L$

$A_{DFA} = \{\langle B, w\rangle \mid B \text{ is a } DFA \text{ that accepts input string } w\}$

To prove $A_{DFA} \in L$ we need to construct a deterministic Turing machine (*TM*) to decide A_{DFA} in logarithmic space.

Let M be the *TM* that decides A_{DFA} in log – space.

The construction of *TM* M is as follows:

$M =$"on input $\langle B, w\rangle$, where B is a *DFA* and w is string:

1. Simulate B on w by keeping track of B's current state and its current head location and updating them appropriately.

2. If the simulation ends in an accept state, accept

3. else if the simulation end in non accepting that, reject"

The SPACE required to carry out this simulation is $O(\log n)$, since n items of values storing its input by M. Thus we constructed *TM* M to decide A_{DFA} in log – space therefore $A_{DFA} \in L$.

Comment

Problem

Show that any PSPACE-hard language is also NP-hard.

Step-by-step solution

Step 1 of 2

PSPACE –completeness: A language B is PSPACE – complete if it satisfies two conditions

1. B exists in PSPACE, and

2. Every A is PSPACE is polynomial time reducible to B.

If B satisfies condition 2, we say that it is PSPACE – hard

Comment

Step 2 of 2

NP – Completeness : A language B is *NP* – complete if it satisfies two conditions.

1. B is in *NP*, and

2. Every A is *NP* is polynomial time reducible to B

If B satisfies condition 2, we say that it is NP – hard.

Now we have to show that any PSPACE – hard language is NP – hard.

We know that NP is subset of PSPACE.

Therefore any string outside of PSPACE is also outside of NP.

In any problem from NP will reduced to PSPACE _ hard language.

• We know that $\text{"SAT} = \{ <\Phi> | \text{ is a satisfiable Boolean formula}\}."$

• Also we know that $\text{"TQBF} = \{ <\Phi> | \text{ is a true fully quantified Boolean formula}\}."$

• We know that $SAT \in \text{NP-complete}$

• Since any SAT problem can be reduced to a TQBF problem by simply adding “there exist x_n” to the front of SAT expression for each variable x_n.

So SAT problem can be solved using TQBF algorithm

• But *TQBF* problem is reduced to any *PSPACE* – hard problem. Because as we know that *TQBF* is *PSPACE* –complete.

• Thus *SAT* is reducing to *PSPACE* – hard, that means *NP* – hard problem is reduced to *PSPACE* – hard.

Thus any *PSPACE* – hard is also *NP* – hard.

Comment

Problem

Show that NL is closed under the operations union, concatenation, and star.

Step-by-step solution

Step 1 of 4

Class *NL(Non-deterministic Logarithmic space):*

NL is the class of languages that are decidable in logarithmic space on a non deterministic Turing machine. i.e., $NL = NSPACE(\log n)$

let L_1 and L_2 be the languages that are decided by *NL* - machines M_1 and M_2.

Now we want to show that

• There is a nondeterministic decides $M_\cup$ such that $L(M_\cup) = L_1 \cup L_2$

• There is a nondeterministic decider $M_\cap$ such that $L(M_\cap) = L_1 \cap L_2$.

• There is a nondeterministic decider $M*$ such that $L(M_*) = L_1^*$

Now these 3 machines $M_\cup, M_\cap, M_*$ are for 3 different operations.

Comment

Step 2 of 4

(i)

$M_\cup =$"on input w:

1. Run M_1 on w, If M_1 accepted then accept

2. Else run M_2 on w, if M_2 accepted then accept

3. Else reject"

The machine $M_\cup$ will accept the input w if either M_1 or M_2 accept w.

If both M_1 and M_2 reject the input w then $M_\cup$ will reject w.

Comment

Step 3 of 4

(ii) **Intersection:**

$M_\cap =$"on input w:

1. Run M_1 on w, if M_1 rejected then reject.

2. Else run M_2 on w, if M_2 rejected then reject.

3. Else accept.

The machine $M_\cap$ will accept the input w if both M_1 and M_2 reject w then $M_\cap$ will reject w. If any one of M_1 and M_2 reject w then $M_\cap$ will reject w.

Comment

Step 4 of 4

(iii) **Star:**

Machine M_* has more complex algorithm as follows.

$M_* =$ "on input w:

1. P_1 and P_2 be two input positions. P_1 is initialized to 0 and P_2 is initialized to the position immediately preceding the first input symbol.

2. If there is no input symbol after P_2 then accept w.

3. Move P_2 forward to a non-deterministically selected input position.

4. Simulate M_1 non-deterministically on the substring of w from position following P_1 to the position P_2.

5. If M_1 is entered into accept state then copy P_2 to P_1 and go to stage 2

So M_* will decide L_1^*,

In this way ***NL* is closed under union intersection and star**.

Comment

Problem

Let EQ_{REX} = { $\langle R, S \rangle$ | R and S are equivalent regular expressions}. Show that EQ_{REX} ? PSPACE.

Step-by-step solution

Step 1 of 2

Language is $EQ_{REX} = \{(R,S) \mid R \text{ and S are equivalent regular expression}\}$

To show: $EQ_{REX} \in PSPACE$

PSPACE: PSPACE is deterministic Turing machine that contains the class of languages that are decidable in polynomial space on a deterministic Turing machine that is:

$$PSPACE = \bigcup_k SPACE(n^k)$$

For any language A, it is known that:

$$\overline{A} \in NPSPACE$$
$$\Rightarrow \overline{A} \in PSPACE$$
$$\Rightarrow A \in PSPACE$$

Thus, if it is shown that $\overline{EQ_{REX}} \in PSPACE$ then that implies that $EQ_{REX} \in PSPACE$

It is known that NPSPACE is non-deterministic Turing machine that contains the class of languages which are decidable in polynomial space.

Comment

Step 2 of 2

Let M be the non-deterministic Turing machine that decides $\overline{EQ_{REX}}$ in a polynomial space as follows:

M= “On input (R, S) where R, S are equivalent regular expressions.” the following points are followed:

• Construct non-deterministic finite automata $N_x = (Q_x, \Sigma, \delta_x, q_x, A_x)$ such that $L(N_x) = L(X)$ for $X \in \{R,S\}$.

• Let $m_X = \{q_X\}$.

• Repeat $2^{\max} X \in \{R,S\}^{|Q_x|}$ times.

• If $m_k \cap A_s = \phi \Leftrightarrow m_s \cap A_s \neq \phi, accept$.

• Pick any $a \in \Sigma$ and change m_x to $\cup_{q \in m_s} \delta_x(q,a)$ for $X \in \{R,S\}$.

• Reject

Hence, non-deterministic Turing machine M decides $\overline{EQ_{REX}}$ in polynomial space

Therefore, $\overline{EQ_{REX}} \in NPSPACE$ and hence $\overline{EQ_{REX}} \in PSPACE$

Comment

Problem

A *ladder* is a sequence of strings $s_1, s_2, \ldots, s_k$, wherein every string differs from the preceding one by exactly one character. For example, the following is a ladder of English words, starting with "head" and ending with "free":

head, hear, near, fear, bear, beer, deer, deed, feed, feet, fret, free.

Let $LADDER_{\text{DFA}} = \{\langle M, s, t\rangle \mid M \text{ is a DFA and } L(M) \text{ contains a ladder of strings, starting with } s \text{ and ending with } t\}$. Show that $LADDER_{\text{DFA}}$ is in PSPACE.

Step-by-step solution

Step 1 of 3

$LADDER_{DFA} \in PSPACE$ $LADDER_{DFA}$= {(M, s, t) I M is a DFA and L(M) contains a ladder of strings, starting with s and ending with t}

The ladder is a sequence of strings where every string differs from preceding one in exactly one character.

To prove: $LADDER_{DFA}$ is in PSPACE

It is known that PSPACE=NPSPACE so, if it is shown that $LADDER_{DFA} \in NPSPACE$ then it implies that $LADDER_{DFA} \in PSPACE$

It is known that NPSPACE is non-deterministic Turing Machine that clears the class of languages which can be decided in polynomial space.

Comment

Step 2 of 3

Follow the following steps:

• If $|s| \neq |t|$, then reject.

• Otherwise, the graph G of size vertices are indexed by strings in $2^{|s|}$ and there is a directed edge from w_1 to w_2 vary in precisely one character and $w_1, w_2 \in L(M)$

• $(M, s, t) \in LADDER_{DFA}$ if and only if there is a path from s to t in G.

• It can be checked in NPSPACE by guessing the path and at each step storing only the name of the current vertex.

• To guess the path at the vertex w_1 non deterministically select a new vertex w_2 that differs from w_1 in precisely one character and verify that M accepts w_2.

• The machine M always halts by keeping a counter and incrementing it with each guess and rejecting when the counter hits $|\Sigma|^{|s|}$

Comment

Step 3 of 3

Thus, the non-deterministic Turing machine is constructed to decide $LADDER_{DFA}$

Therefore, $LADDER_{DFA} \in NPSPACE$ so, $LADDER_{DFA} \in PSPACE$

Comment

Problem

The Japanese game *go-moku* is played by two players, “X” and “O,” on a 19 × 19 grid. Players take turns placing markers, and the first player to achieve five of her markers consecutively in a row, column, or diagonal is the winner. Consider this game generalized to an n × n board. Let

$$GM = \{\langle B\rangle \mid B \text{ is a position in generalized go-moku, where player “X” has a winning strategy}\}.$$

By a *position* we mean a board with markers placed on it, such as may occur in the middle of a play of the game, together with an indication of which player moves next. Show that *GM* ? PSPACE.

Step-by-step solution

Step 1 of 3

PSPACE: PSPACE is deterministic Turing machine that contains the class of languages that are decidable in polynomial space on a deterministic Turing machine *i.e.,* $\mathrm{PSPACE} = \bigcup_k SPACE\left(n^k\right)$

To generalize *go – moku* game on $n \times n$ board is given as

$GM = \{\langle B\rangle \mid B \text{is a position in generalized } go\text{-}moku, \text{where player "X" has a winning strategy}\}$

• This game is play by 2 players "X" and "O" Now we have to prove that $GM \in PSPACE$.

• Let us assume that B is written as a grid of "X" and "O" are empty, so the length of the input is $O\left(n^2\right)$

• Now let us define a recursive algorithm to solve $GM(B)$, which accepts if there is a winning strategy for player X starting at position B.

Comment

Step 2 of 3

Recursive Algorithm of *GM* :

$GM(B)$:

(1) $\text{potential } X \text{ moves:}$ All spaces of i in position B without marker on them

(i) Put an X marker on space i, next changing the position to B'. If there are $5X's$ in a row (it a best move) then accept. If the board is now full, and no one has won, reject.

(ii) $\text{potential } O \text{ moves:}$ all spaces j in position B' without _ markers on them.

(a) Put an O marker on space j, next changing the position to B'. if there are $5\,O's$ in a row(it is also best move) or the board in full and no one has won, loop to the next *i* (go to step(i)); putting an X on i is obviously a bad move.

(b) Otherwise, run $\mathrm{GM}(\mathrm{B}'')$

• If it accepts, loop to next j (goto step (b)).

• If it rejects, loop to the next i (goto step (i))

(iii) If all j cause $\mathrm{GM}(\mathrm{B}'')$ to accept, i is a good X move, since it covers all possible O moves, so accept.

Comment

Step 3 of 3

(2) If no *i* step (i) causes accept, reject, there are no good moves from this position, so reject.

• We can loop through all moves i, j at each step, since we can just reuse the space.

• As w just need to store configurations B', B'', which takes only $O(n^2)$ space.

• And our recursion is only $O(n^2)$ since there are at most n^2 moves.

• Total space needed is $O(n^4)$, which is a polynomial in the input length, since we assumed the input had length $O(n^2)$.

• Clearly no game of generalized *go – moku* on $n \times n$ board can have more than $n \times n$ moves.

• So possible configurations following from *B* needs only polynomial spac*e.*

• Thus $GM \in PSPACE$

Comment

Problem

Show that if every NP-hard language is also PSPACE-hard, then PSPACE = NP.

Step-by-step solution

Step 1 of 1

Given statement,

Every *NP* – hard language is also $PSPACE-\text{hard}$. That means NP $\subseteq$PSPACE.

We have to show that

$PSPACE = NP$.

NP – complete: "A language *B* is *NP*- complete if it satisfies two conditions.

1. *B* is in *NP*, and

2. Every *A* in *NP* is polynomial time reducible to *B*"

If *B* satisfies condition 2, we say that it is *NP*- hard.

• From the hypothesis, Every *NP*- hard language is also *PSPACE*- hard.

• By the definition, *NP* – hard contains all of the *NP* – complete problem.

• So every *NP* – complete language is also *PSPACE*-hard. We know that *SAT* is *PSPACE*- hard.

• For any language *A* is *PSPACE,* A reduces to *SAT*.

• Assume that *A* is in *NP*.

Create a Turing Machine (TM), *M* as follows

$M =$"On input *x*

1. compute $f(x)$. The poly – time nondeterministic algorithm between *A* and *SAT*.

2. If $f(x)$ is satisfible, accept *x*.

3. Else reject."

Claim *M* decides *A* since *x* is in *A* iff $f(x)$ is in *SAT*. Also *M* is an *NP* machine since computing *SAT* is in *NP*.

Thus, language *A* is in *PSPACE* then *A* is in *NP*

So, $PSPACE = NP$.

Comment

Problem

Show that *TQBF* restricted to formulas where the part following the quantifiers is in conjunctive normal form is still PSPACE-complete.

Step-by-step solution

Step 1 of 3

TQBF: *TQBF* problem is to determine whether a fully quantified Boolean formula is true or false.

$TQBF = \{\langle\phi\rangle \mid \phi$ is a true fully quantified Boolean formula$\}$

Show that *TQBF* restricted to formulas where the part following quantifiers is in conjunctive normal form (cnf) is $PSPACE$- complete.

That is $cnf - TQBF = \{\overline{Q}\phi \in TQBF \mid \phi \text{ is in } cnf\}$ is *PSPACE*- complete.

Comment

Step 2 of 3

PSPACE- complete:

A language *B* is PSPACE- complete if it satisfies two conditions:

1. *B* is in *PSPACE,* and

2. Every *A* in *PSPACE* is polynomial time reducible to *B*.

If *B* merely satisfies condition 2, we say that it is *PSPACE*-hard.

1. $cnf - TQBF \in PSPACE$: we know that $TQBF \in PSPACE$

As a subset of *TQBF* characterized by a simple syntactic test, $cnf - TQBF$ is obviously still in *PSPACE*.

2. $cnf - TQBF \in PSPACE - \text{hard}$:

• We show $PSPACE - \text{hardness}$ by proving $TQBF \leq_p cnf - TQBF$

• Given a *TQBF* instance $\overline{Q}\phi$ where $\overline{Q}$ is a sequence of quantifiers and ϕ is a Boolean formula, we construct in polynomial time an equivalent $cnf - TQBF$ instance $\overline{Q}\overline{E}\psi$ where $\overline{E}$ is a sequence of existential quantifiers concerning the fresh proposition in ψ but not in ϕ.

• Here we use the technique for transforming the *SAT* instance ϕ in to an equi – satisfiable *CSAT* instance ψ.

• $\phi F \phi$ if and only if there exist an extension π' of π that make ψ true.

• This construction establishes that

$\pi \models \phi$ that is $\pi \models \overline{E}\psi$ and hence $\models \overline{Q}\phi$ iff $\models \overline{Q}\,\overline{E}\psi$.

Comment

Step 3 of 3

From (1) and (2) $cnf - TQBF$ is $PSPACE$ complete.

Thus, it is proved that the $TQBF$ restricted to formulas where the part following the quantifiers is conjunctive normal from, is still $PSPACE -$ complete.

Comment

Problem

Define A_{LBA} = { $\langle M, w \rangle$ | M is an LBA that accepts input w}. Show that A_{LBA} is PSPACE-complete.

线性有界自动机

Step-by-step solution

Step 1 of 4

$\underline{PSPACE-\text{complete}}$: A language *B* is *PSPACE* – complete if it satisfies two conditions.

1. *B* is in *PSPACE*, and

2. every *A* in *PAPACE* is polynomial time reducible to *B*.

If *B* satisfies condition 2, we say that *B* is *PSPACE*- hard

Comment

Step 2 of 4

Given language is

$A_{LBA} = \{\langle M, w \rangle | M \text{ is an } LBA \text{ that accepts input } w\}$

A linear bound automation (LBA) is one – tape, one – head *NTM*.

We need to show that A_{LBA} is $PSPACE-$complete.

That means A_{LBA} has to satisfy the 2 conditions of *PSPACE*-complete.

Comment

Step 3 of 4

(i) $\underline{A_{LBA} \in PSPACE}$:

To show $A_{LBA} \in PSPACE$, we need to construct a deterministic Turing machine that decides A_{LBA} in polynomial space.

Let *T* be the Turing machine (TM) that decides A_{LBA} in polynomial space.

T can be constructed as follows.

$T =$" on input $\langle M, w \rangle$

Where *M* is a $TM(Q, \Sigma, \Gamma, \delta, q_0, q_{accept}, q_{reject})$ and $w \in \Sigma^*$

1. Take a step count *S* and initialize *S* to O.

2. While $S < |Q| \cdot |w| \cdot |\Gamma|^{|w|}$

(i) Simulate *M* on *w*

(ii) If *M* runs out of $|w|-$bounded tape then reject.

(iii) accept if *M* could in this step.

(iv) increment *S*.

3. Reject."

This machine *T* runs in polynomial space since the extra counter assumes values at most exponential in the length of the input word.

Thus we constructed a *TM T* to decide A_{LBA} in polynomial space.

Therefore $A_{LBA} \subset PSPACE$

Comment

Step 4 of 4

(ii) A_{LBA} is $PSPACE-\text{hard}$:

Let $L \in PSPACE$

Let M be TM that decides L in space at most n^k

clearly $L \leq_P A_{LBA}$ by giving a reduction that maps w to $\left\langle M, wL_1^{|w|^{k-1}} \right\rangle$.

Thus every L in $PSPACE$ is polynomial time reducible to A_{LBA}.

Hence A_{LBA} is $PSPACE$-hard.

From (i) and (ii) A_{LBA} is $PSPACE-$complete

Comments (2)

Problem

The cat-and-mouse game is played by two players, “Cat” and “Mouse,” on an arbitrary undirected graph. At a given point, each player occupies a node of the graph. The players take turns moving to a node adjacent to the one that they currently occupy. A special node of the graph is called “Hole.” Cat wins if the two players ever occupy the same node. Mouse wins if it reaches the Hole before the preceding happens. The game is a draw if a situation repeats (i.e., the two players simultaneously occupy positions that they simultaneously occupied previously, and it is the same player’s turn to move).

$$HAPPY\text{-}CAT = \{\langle G, c, m, h\rangle \mid G, c, m, h \text{ are respectively a graph, and positions of the Cat, Mouse, and Hole, such that Cat has a winning strategy if Cat moves first}\}.$$

Show that *HAPPY-CAT* is in P. (Hint: The solution is not complicated and doesn’t depend on subtle details in the way the game is defined. Consider the entire game tree. It is exponentially big, but you can search it in polynomial time.)

Step-by-step solution

Step 1 of 1

Firstly note that game can have only $2n^2$ configuration, defined by position of the cat, position of the mouse and if it is cat's turn. So can construct a directed graph consisting of $2n^2$ nodes where every node corresponds to a configuration of game and there is an edge from node u to the node v, if we can go from configuration corresponding to u to configuration corresponding to v in one move.

Now following algorithm solves HAPPY-CAT.

1. Mark all nodes (a,a,x) where a is a node in G, and $x \in \{true, false\}$.

2. If for a node $u=(a,b,true)$, there is a node $v=(c,b,false)$ which is marked and (u,v) is an edge then mark u.

3. If for a node $u=(a,b,false)$, all nodes $v=(a,c,true)$ are marked and (u,v) is an edge then mark u.

4. Repeat steps 2 and 3 until no new nodes are marked.

5. Accept if start node $s=(c,m,true)$ is marked.

Here algorithm takes $O(n^2)$ time to perform step 1, $O(n^2)$ time per iteration of the loop while loop is executed $O(n^2)$ times. Hence runs in polynomial time.

Comment

Problem

Consider the following two-person version of the language *PUZZLE* that was described in Problem 7.28. Each player starts with an ordered stack of puzzle cards. The players take turns placing the cards in order in the box and may choose which side faces up. Player I wins if all hole positions are blocked in the final stack, and Player II wins if some hole position remains unblocked. Show that the problem of determining which player has a winning strategy for a given starting configuration of the cards is PSPACE-complete.

Step-by-step solution

Step 1 of 4

Consider the two-people version of the language $PUZZLE$. In this, every player uses an ordered puzzle card stack to start.

• The cards are placed in each turn in order, by the player, in the box and can select which part faces up. Player I wins if every hole place are closed in the resultant stack and player II wins if some position of the holes remains in unblocked state.

• Now, it can be shows that “the problem of determining which player has a winning strategy for a given starting configuration of the cards is $PSPACE$-complete”.

Comment

Step 2 of 4

Consider the formula φ, which is defined as:

$$\varphi = \exists p_1 \forall p_2 \exists p_3 \left[(p_1 \vee p_2) \wedge (p_2 \vee p_3) \wedge (\overline{p_2} \vee \overline{p_3}) \right]$$

In the above given formula for φ, player I picks the value of p_1 and then player II picks the value of p_2 and finally player I picks the value of p_3.

• Now, assign true with the value 1 and false with the value 0. Suppose that player I picks p_1=1, then the player II picks p_2=0, and finally player I picks p_3=1.

• Now by using these values in the sub-formula which is described above as

$$(p_1 \vee p_2) \wedge (p_2 \vee p_3) \wedge (\overline{p_2} \vee \overline{p_3})$$

is1, therefore the player I has won the game. Then, it can be said that “player I has a strategy of winning for the given puzzle”.

Comment

Step 3 of 4

Now, a slightly change in formula, which is given above, will change the strategy of winning of the player.

• Consider the modified formula:

$$(p_1 \vee p_2) \wedge (p_2 \vee p_3) \wedge (p_2 \vee \overline{p_3})$$

The above formula is for winning strategy for the player II.

Comment

Step 4 of 4

Consider the $PUZZLE$, which is described above, is $PSPACE-complete$. It is because, it is same as $TQBF$. The $PUZZLE$ problem is $PSPACE-complete$ can be proved by using the fact of $PUZZLE = TQBF$.

• Consider the formula $\varphi = \exists p_1 \forall p_2 \exists p_3 ... [\psi]$ became TRUE when setting of p_1 exists in such a way that, for every setting of p_2, a setting of p_3 exist is such a way and so on. Where, ψ sows TRUE value under the setting of the taken variable.

• The same process will be applied for the winning strategy for the player II.

• If the formula φ has of the form $\forall p_1, p_2, p_3 \exists p_4, p_5 \forall p_6 [\psi]$, player I would make the first three move in e $PUZZLE$, to assign values to $p_1, p_2 \, and \, p_3$.

• Then, player II makes two moves to assign $p_4 \, and \, p_5$. Finally, the player I assigned a value p_6.

Therefore, $\varphi \in TQBF$ exactly when $\varphi \in PUZZLE$. Hence it can be said that "the problem of determining which player has a winning strategy for a given starting configuration of the cards is $PSPACE$ -complete".

Comment

Problem

Read the definition of *MIN-FORMULA* in Problem 7.46.

a. Show that $MIN\text{-}FORMULA \in$ PSPACE.

b. Explain why this argument fails to show that $MIN\text{-}FORMULA \in$ coNP: If $\phi \notin MIN\text{-}FORMULA$, then ϕ has a smaller equivalent formula. An NTM can verify that $\phi \in \overline{MIN\text{-}FORMULA}$ by guessing that formula.

Problem 7.46

Say that two Boolean formulas are ***equivalent*** if they have the same set of variables and are true on the same set of assignments to those variables (i.e., they describe the same Boolean function). A Boolean formula is ***minimal*** if no shorter Boolean formula is equivalent to it. Let *MIN-FORMULA* be the collection of minimal Boolean formulas. Show that if P = NP, then *MIN-FORMULA* ? P.

Step-by-step solution

Step 1 of 1

a. Consider following algorithm

On Input F:

1. For each string s such that $|s| < |F|$, if s is a valid representation of a formula (this can be easily checked) which is equivalent to F (this can be checked in polynomial space by evaluating both F and s over all possible truth assignments and comparing the results) then reject.

2. If all string has been tried without rejecting, accept.

Correctness of the algorithm should be evident. Only space used by the algorithm is for storing formula F, string s and current assignment of literals which amounts to polynomial space. Hence, $MIN-FORMULA \in PSPACE$.

b. It fails to show $MIN-FORMULA \in coNP$ because it is not known if one can verify equivalence of two Boolean formulae is polynomial time.

Comment

Problem

Let A be the language of properly nested parentheses. For example, (()) and ((()(()))() are in A, but)(is not. Show that A is in L.

Step-by-step solution

Step 1 of 2

The class L: L is the class of languages that are decidable in logarithmic space on a deterministic truing machine.

That is $L = SPACE(\log n)$ log一般需要构造双带图灵机

Given that

'*A*' be the language of property nested parenthesis.

For example, $(( ))$ and $(((\ )))$ etc are in *A*. But not $)($.

Comment

Step 2 of 2

We have to show that *A* is in *L*.

That means, we have to construct deterministic Turing machine (DTM) that decides *A* in logarithmic space.

Let *M* be the *DTM* that decides *A* in logarithmic space.

The construction of *M* is as follows:

$M =$ "On input *w*:

Where *w* is a sequence of parenthesis. 相当于一一匹配

1. Starting at the first character of *w*, move right across *w*.

2. when left parenthesis '(' is encountered, add 1 to the work – tape and move right.

3. when right parenthesis ')' is encounter and the work tape is blank, then reject.

Otherwise subtract 1 from the work – tape and move right.

4. When the end is reached, accept if the work tape is blank, reject if the work tape in not blank."

• Clearly, the only space used by this algorithm is for the counter on the work tape.

• If this counter is in binary, then the most space used by the algorithm is $O(\ln k)$

Where *k* is the number of $(s$.

• Since the number of $(s$ is less than or equal to *n* (the size of tape), this places the language *A* in *L*.

Thus we proved that $A \in L$

Comment

Problem

Let B be the language of properly nested parentheses and brackets. For example, ([()()]()[]) is in B but ([)] is not. Show that B is in L.

Step-by-step solution

Step 1 of 3

The class L: L is the class of languages that are decidable in logarithmic space on a deterministic truing machine.

That is, $L = SPACE(\log n)$

Given that,

' B ' be the language of property nested parenthesis. For example, $[(\ )(\ )]$.

But not $([)]$.

We have to show that B is in L.

That means, we have to construct deterministic Turing machine (DTM) that decides B in logarithmic space.

Comment

Step 2 of 3

Let *M* be the *DTM* that decides B in logarithmic space.

The construction of *M* is as follows:

$M =$ "On input w:

Where *w* is a sequence of parenthesis and brackets.

1. if $w = \in$ then accept and halt.

2. Otherwise, take a counter $i = 0$.

3. Read the input sequentially from left to right.

4. When $[$ is read from the input, increment *i* by 1.

5. When $]$ is read from the input, decrement *i* by 1.

6. If *i* becomes negative or is not restored to zero at the end of the input, then reject.

7. Otherwise, for each symbol ' a ' in the input sequentially from left to right repeat the following.

8. If a is $)$ or $]$, skip it.

9. Otherwise, ' a ' is a left delimiter (or $[$, using a counter find our the matching right delimiter ' b ' as we done from step 2 to step 6.

10. If b is not found, or if a and b are of different types (parenthesis and bracket or bracket and parenthesis), then reject.

11. If not rejected so for then accept".

Comment

Step 3 of 3

• Clearly the only space used by this algorithm is for the counter on the work tape.

• If this counters are binary, then the most space used by algorithm is $O(\ln k)$

• Where k is number of brackets and parenthesis

• Since k is always less than or equal to size of the tape this places the language B in L.

• Thus we proved that $B \in L$.So, B is in L.

Problem

The game of ***Nim*** is played with a collection of piles of sticks. In one move, a player may remove any nonzero number of sticks from a single pile. The players alternately take turns making moves. The player who removes the very last stick loses. Say that we have a game position in Nim with k piles containing $s_1, \ldots, s_k$ sticks. Call the position ***balanced*** if each column of bits contains an even number of 1s when each of the numbers s_i is written in binary, and the binary numbers are written as rows of a matrix aligned at the low order bits. Prove the following two facts.

a. Starting in an unbalanced position, a single move exists that changes the position into a balanced one.

b. Starting in a balanced position, every single move changes the position into an unbalanced one.

Let *NIM* = { $\langle s_1, \ldots, s_k \rangle$ I each s_i is a binary number and Player I has a winning strategy in the Nim game starting at this position}. Use the preceding facts about balanced positions to show that *NIM* ? L.

Step-by-step solution

Step 1 of 5

In a game of **NIM** that consists two players, there are k heaps containing $s_1, s_2, \ldots, s_k$ numbers of stacks. Now, the binary equivalent of each of the numbers (s_i) is taken and arrange them row wise and aligned them at the low order bits.

• The arrangement is said to be "balanced" if there are even no of one's in each column. An XOR operation is performed on the stick numbers in each heaps by the column values. **This is called "NIM-sum" of the numbers**.

• Hence, from the definition of balanced position, it can be said that if **zero value is acquired by the NIM-sum, then the arrangement is balanced otherwise not**.

Comment

Step 2 of 5

Now to prove the given facts, consider that $s_1, s_2, \ldots, s_k$ be the heaps size before any move is made and $s_1', s_2', \ldots, s_k'$ be the size of heaps after a move is made.

• Assume NIM_s be the NIM-sum before any move (that is, $NIM_s = s_1 \oplus s_2 \oplus \cdots \oplus s_k$) and NIM_s' be the NIM-sum after any move (that is, $NIM_s' = s_1' \oplus s_2' \oplus \cdots \oplus s_k'$).

• If the move is in the heap l then it has $s_i = s_i'$ for all $i \neq l$, and $s_l > s_l'$. It is considered that, the XOR function or more precisely in this case.

• The NIM-sum function $(\oplus)$ follow the simple associative and communicative laws and also follow one more property, that is, $s \oplus s = 0$.

Therefore, by using these properties, which are discussed above:

$$\begin{aligned} NIM_s' &= 0 \oplus NIM_s' \\ &= NIM_s \oplus NIM_s \oplus NIM_s' \\ &= NIM_s \oplus (s_1 \oplus s_2 \oplus \cdots \oplus s_k) \oplus (s_1' \oplus s_2' \oplus \cdots \oplus s_k') \\ &= NIM_s \oplus (s_1 \oplus s_1') \oplus \cdots \oplus (s_k \oplus s_k') \\ &= NIM_s \oplus 0 \oplus 0 \oplus \cdots \oplus (s_l \oplus s_l') \oplus \cdots \oplus 0 \\ &= NIM_s \oplus s_l \oplus s_l' \end{aligned}$$

Comment

Step 3 of 5

Consider an unbalanced position of piles (that is when $NIM_s \neq 0$), Suppose d be the most significant position of non-zero bit in the binary format of NIM_s.

• Now, select a l in such a way that a non-zero value is acquired by dth bit of s_l. The null or zero value acquired by the dth bit of NIM_s, when no such l exists.

• Now consider, $s'_l = NIM_s \oplus s_l$ and $s'_l < s_l$ because all bits to the left of the dth bit is are same in s_l and s'_l and bit d decreases from 1 to 0 or the value decreased by 2^d.

• So the first player by the winning strategy makes a move by choosing $s_l - s'_l$ objects from heap l. Then

$$\begin{aligned} NIM'_s &= NIM_s \oplus s_l \oplus s'_l \\ &= NIM_s \oplus s_l \oplus (NIM_s \oplus s_l) \\ &= 0 \end{aligned}$$

Therefore, there exists a single move that changes the position from unbalanced to a balanced one.

Comment

Step 4 of 5

Now, initialize from a balanced position, that is $NIM_s = 0$, then $NIM'_s = s_l \oplus s'_l$. Suppose, the first move is made on heap l making $s_l \neq s'_l$ then NIM'_s acquires a non-zero value**. Hence, every single move changes the position balanced to an unbalanced one.**

Comment

Step 5 of 5

In a normal play condition, after considering the fact (that is, none of the players makes any mistakes) it is easy to find a winning strategy. It is clear from the above mathematical deduction, that the strategy is to make the NIM-sum zero after each move.

• Consider the winning strategy, if the game is in balanced condition (that is NIM-sum is already zero). User has to move second letting Player II to move first and conversely when the game condition is unbalanced one (that is, the NIM-sum is non-zero). Then player I's has to move first.

• Now considering an algorithm to design a TM for Player I's winning strategy, where each input is already in its binary equivalent. These easy steps can be followed:

Suppose, M = Step

Cross off the 1's on finding of two 1s.

a. If no 1 left uncrossed after the entire scan of input, Player-I has to move second.

b. Else, Player-I has to move first.

So, the only space tradeoff is to store the binary numbers, which requires $O(\log(k))$ space. Therefore, $NIM \in L$. □

Comment

Problem

Let *MULT* = {a#b#cl a, b, c are binary natural numbers and a × b = c}. Show that *MULT* ? L.

Step-by-step solution

Step 1 of 2

The class L : L is the class of languages that are decidable in logarithmic space on a deterministic truing machine.

That is, $L = SPACE(\log n)$

Given that,

$MULT = \{ a\#b\#c \mid$ where a,b,c are binary natural numbers and $a \times b = c \}$.

Comment

Step 2 of 2

We have to show that $MULT \in L$.

That means, we have to construct a deterministic Turing machine (DTM) that decides $MULT$ in logarithmic space.

Let M be the *DTM* that decides *MULT* in logarithmic space.

The construction of M is as follows:

$M =$"On input $a\#b\#c$:

1. If either of the three strings is not a binary number as defined above then reject

2. Initialize a binary counter *i* pointing to the O.

3. Initialize *a* binary counter *j* to $\max(0, i+1-\text{length of } 1)$

4. Initialize a binary counter *k* to $i-j$

5. Now take a binary counter $x \leftarrow x + a[j] * b[k]$

6. Repeat steps 3 to 5 $\min(i, n-1)$ times.

7. Repeat steps 2 to 6 $2n-1$ times and calculate $x = \text{floor}(x/2)$

8. If any discrepancy arises between the calculate next bit of *c* and then reject.

9. If the multiplication ended with no errors then accept.

Thus, clearly *M* runs in log space and decides $MULT$.

So, $MULT \in L$.

Comment

Problem

For any positive integer x, let x^R be the integer whose binary representation is the reverse of the binary representation of x. (Assume no leading 0s in the binary representation of x.) Define the function $\mathcal{R}^+ : \mathcal{N} \longrightarrow \mathcal{N}$ where $\mathcal{R}^+(x) = x + x^{\mathcal{R}}$.

a. Let $A_2 = \{\langle x, y\rangle | \ \mathcal{R}^+(x) = y\}$. Show $A_2 \in \mathrm{L}$.

b. Let $A_3 = \{\langle x, y\rangle | \ \mathcal{R}^+(\mathcal{R}^+(x)) = y\}$. Show $A_3 \in \mathrm{L}$.

Step-by-step solution

Step 1 of 1

Given:

x is one of the positive integer and the reverse of the integer x is donated by the symbol x^R in the binary representation. But the condition is that there is no zero's in the binary representation of the number *x*.

Function R^+ need to be defined in such a way that the integer x is a natural number.

$R^+(x) = x + x^R$

Suppose, the positive integer x is 15, binary representation of $(15)_{10}$ is written as $(1111)_2$.

The reverse of binary representation x^R is represented as $(0000)_2$.

Inverse of the binary representation is done by converting the ones with zeros and zeros with ones.

Proof:

Consider the Turing machine *M* for every positive integer *x* , inverse of *x* is computed. When the computation is performed on the integer *x* and the inverse of the integer *x* then result will also be *x*.

This is because when a binary number 1 is added to 0 then result is always 1. Here, *x* positive integer is converted into binary representation but the condition is that the binary representation of *x* should not contain 0.

Turing machine will accept only those values of *x* whose binary representation is 1. Even Turing machine will accept those values of inverse of *x* whose binary representation is 0.

After that machine perform the computation between *x* and x^R.

Hence, it is proved that language $A_2 \in L$.

Consider the Turing machine *M* for every positive integer *x* , inverse of *x* is computed. When the computation is performed on the integer *x* and the inverse of the integer *x* then result will also be x.

This is because when a binary number 1 is added to 0 then result is always 1. Here, *x* positive integer is converted into binary representation but the condition is that the binary representation of *x* should not contain 0.

After performing the computation Turing machine once again call the same function. After calling the same function also the result after computation will be the value equal to *x*.

Turing machine will accept only those values of *x* whose binary representation is 1. Even Turing machine will accept those values of inverse of *x* whose binary representation is 0.

After that machine perform the computation between *x* and x^R.

Hence, it is proved that language $A_3 \in L$.

Comment

Problem

a. Let *ADD* = { $\langle x, y, z \rangle$ | x, y, z > 0 are binary integers and x+ y = z}. Show that *ADD* ? L.

b. Let *PAL-ADD* = { $\langle x, y \rangle$ | x, y > 0 are binary integers where x + y is an integer whose binary representation is a palindrome}. (Note that the binary representation of the sum is assumed not to have leading zeros. A palindrome is a string that equals its reverse.) Show that *PAL-ADD* ? L.

Step-by-step solution

Step 1 of 2

The class L: L is the class of languages that are decidable in logarithmic space on a deterministic truing machine.

That is, $L = SPACE(\log n)$

(a) Given Language is

$$ADD = \{\langle x, y, z\rangle \mid x, y, z > 0 \text{ are binary integers and } x + y = z\}$$

We have to show that $ADD \in L$.

That means, we have to construct a deterministic Turing machine (DTM) that decides *ADD* in logarithmic space.

Let M_1 be the *DTM* that decides *ADD* in logarithmic space.

The construction of M_1 is as follows:

M_1 ="On input $\langle x, y, z\rangle$:

1. If either of the three strings is not a binary number in the sense defined above then reject.

2. Initialize three binary counters i, j, k pointing to the right – most of x, y and respectively.

3. Perform binary addition (bit – wise long addition) using i, j, k and *a* carry flag.

4. If any discrepancy arises between the calculated next bit of *z* and the actual next bit of *z*, then reject.

5. If the end of all three number representations is reached and no error detected, then accept. So clearly M_1 runs in log space (it uses 3 counter only) and decides *ADD"*.

Thus *ADD* is in *L*. So, $ADD \in L$.

Comment

Step 2 of 2

(b) Given language is

$PAL\_ADD = \{\langle x, y\rangle \mid x, y > 0$ are binary integers where $x + y$ is an integer whose binary representation is a palindrome}

We have to show that $PAL\_ADD \in L$.

That means, we have to construct a *DTM* that decides PAL_ADD in logarithmic space.

Let M_2 be the *DTM* that decides PAL_ADD in logarithmic space.

The construction of M_2 is as follows.

M_2 ="On input $\langle x, y\rangle$:

1. If either of the two strings is not a binary number in the sence defined above, then reject.

2. Compute the length *l* of $x + y$ in binary.

3. For $i = 1$ to $l/2$.

(i) Compute the i^{th} bit a of $x + y$

(ii) Compute the $(l-i+1)^{th}$ bit *b* of $x+y$

(iii) if $\neq b$ then reject.

4. otherwise, accept"

Clearly M_2 runs in logspace and decides PAL_ADD.

Thus PAL_ADD is in L. So, $PAL\_ADD \in L$

Comment

Problem

Define *UCYCLE* = { $\langle G \rangle$ | G is an undirected graph that contains a simple cycle}. Show that *UCYCLE* ? L. (Note: G may be a graph that is not connected.)

Step-by-step solution

Step 1 of 3

The class L: L is the class of languages that are decidable in logarithmic space on a deterministic truing machine.

That is, $L = SPACE(\log n)$

Given language is

$UCYCLE = \{ \langle G \rangle \mid G$ is an undirected graph that contains a simple cycle}

We have to show that $UCYCLE \in L$.

Comment

Step 2 of 3

Let *M* be the deterministic Turing machine that decides $UCYCLE$ the construction of M is as follows:

$M =$ "On input $\langle G \rangle$ (*G* is an undirected graph):

1. Select a vertex *u* as starting vertex

2. Select an edge (u,v) from *u*.

3. Start traversal through (u,v), if we come back to *u* through an edge different that (u,v), then accept.

4. Otherwise, reject"

Comment

Step 3 of 3

• If we come back to the starting vertex through an edge different than the one we started on, we declare that the graph contain a cycle.

• Since all the vertices and all the edges are enumerated in logspace *M* decides $UCYCLE$ in logarithmic space.

• Therefore, $UCYCLE \in L$.

Comment

Problem

For each n, exhibit two regular expressions, R and S, of length poly(n), where L(R) ≠ L(S), but where the first string on which they differ is exponentially long. In other words, L(R) and L(S) must be different, yet agree on all strings of length up to $2^{\epsilon n}$ > 0.

Step-by-step solution

Step 1 of 2

Consider that two regular expressions, S and R, of length $poly(n)$ can be exhibited for every n,where $L(R) \neq L(S)$but the first string on which they differ in exponentially long.

• Now, consider the expression the expression which is given below:

$$\underbrace{(11...1)}_{p\text{ times}}^{*}\ \underbrace{11?1?...1?}_{p-2\text{ times}}$$

The above expression identify 1^k for each k which is not a multiple of p .

Comment

Step 2 of 2

Now, **a polynomial-length** can be constructed in such a way that recognize 1^k for each k and except those expression which are multiple of every first n prime.

• The nth prime is less than $n(\ln n + \ln \ln n)$foe every $n \geq 6$. Therefore, the addition of the first $n$primes is $O(n^2 \log n)$.

• It is because the first number which is not a multiple of any of the first prime number is of the order $O(n^2 \log n)$, that cause to stop it in growing exponentially with n.

Hence form the above explanation, it can be said that "**every string of the length up to $2^{\varepsilon n}$, for a constant $\varepsilon > 0$, will be agreed for two regular expression $S$and R, where $L(R)$and $L(S)$must be different".**

Comment

Problem

An undirected graph is ***bipartite*** if its nodes may be divided into two sets so that all edges go from a node in one set to a node in the other set. Show that a graph is bipartite if and only if it doesn't contain a cycle that has an odd number of nodes. Let *BIPARTITE* = { $\langle G \rangle$ | G is a bipartite graph}. Show that *BIPARTITE* ? NL.

Step-by-step solution

Step 1 of 4

Given that

$BIPARTITE = \{ \langle G \rangle \mid G$ is a bipartite graph$\}$

An undirected graph is bipartite if its nodes may be divided into two sets so that all edges go from a node in one set to a node in the other set.

Now we have to show that

• A graph is bipartite if and only if it doesn't contain a cycle that has an odd number of nodes, and

• $BIPARTITE \in NL$

Comment

Step 2 of 4

(i) First we show that if a graph is bipartite then it doesn't contain a cycle that has an odd number of nodes:

• Let *G* be a graph which is bipartite.

• Assume that *G* contain a cycle that has odd number of nodes

• $n_1, n_2, n_3 \ldots, n_{2k}, n_{2k+1}$

• By the definition of bipartite graph, as *G* is a bipartite we have n_1 is in some set *A*, and n_2 must be in some set *B*, then n_3 must be in set *A* etc.

• By induction, all the nodes with an odd subscript must be in set *A*, and all those with an even subscript must be in set *B*

• This implies that n_1 and n_{2k+1} are both in set *A*.

• This is a contradiction because they are connected.

• Thus our assumption that *G* contains a cycle that has odd number of nodes is wrong.

• Hence *G* does not contain a cycle with odd number of nodes.

Comment

Step 3 of 4

(ii) Second, if a graph doesn't contain a cycle with an odd number of nodes then the graph is bipartite:

• Suppose a graph does not contain any odd cycles.

• Pick a node, and label it *A*.

• Label all of its neighbors *B*.

• Label all of their unlabeled neighbors *A*, etc, until all nodes are labeled.

• Suppose that this construction caused two adjacent nodes *x* and *y* to have the same label.

• Then that would mean that both *x* and *y* were reached by taking an even number of steps.

• In either case, the total number of nodes traversed in getting to *x* from the start node, and getting to *y* from the start node (excluding the start node itself), is even.

• But adding the start node in makes the total number of nodes odd, contradicting the hypothesis that there were no odd cycles.

• Thus, the construction succeeds in properly dividing the nodes, so the graph is bipartite.

Comment

Step 4 of 4

(iii) Finally, we show that $BIPARTITE \in NL$

• We know that $NL = CONL$

• So if we prove that $\overline{BIPARTITE} \in NL$ then than implies that $BIPARTITE \in NL$

• $\overline{BIPARTITE}$ ={ $\langle G \rangle \mid G$ is a graph that contain an add cycle}

We know that

" *NL* is the class of languages that are decidable in logarithmic space on a non deterministic Turing machine "

Let *M* be the *NTM* that decides $\langle G \rangle \mid G \ \overline{BIPARTITE}$ in logarithmic space. *M* can be constructed as follows:

$M =$ "on input $\langle G \rangle$:

1. Counter : = 0

2. start: = Nondeterministically select a starting node.

3. Successor := Nondeterministically select a successor of *s*.

4. while (counter $\leq$ the number of nodes)

5. If successor = start and if counter is odd accept

6. otherwise successor $:=$ nondeterministically select a successor of successor

7. If counter has increased above the number of nodes, reject."

Since, the only space used by the above algorithm is for keeping track of the node numbers 'start' and 'Successor' and keeping track of the 'counter', the algorithm clearly only uses log space on any one branch.

• There fore $\overline{BIPARTITE} \in NL$

• Because $NL = CONL, BIPARTITE \in NL$

Comment

Problem

Define *UPATH* to be the counterpart of *PATH* for undirected graphs. Show that

$\overline{BIPARTITE} \leq_{\mathrm{L}} UPATH$. (Note: In fact, we can prove $UPATH \in \mathrm{L}$, and therefore $BIPARTITE \in \mathrm{L}$, but the algorithm [62] is too difficult to present here.)

Step-by-step solution

Step 1 of 2

An assumption $UPATH \in L$ can be used to show $BIPART \in L$. **Now,** As it is known that, all paths which contain a length of two in G are used to acquire the graph of G^2. To show the bipartite condition, all graphs have to show a certain criteria. One of the criteria is that all the graphs should consist a cycle of odd length and also an edge between p and q or an edge (p,q). By considering the above explanation, G can be said as bipartite iff $\forall$ edges (p,q) of G, there is no connection between p and q by the path taken in G^2 . Consider the algorithm which is given below shows how a logarithmic space can be used to solve $BIPART$.

//algorithm

On input G

For every edges (p,q) **in** G

// check for the existence of path.

"Check if there is a path exists from p **to** q **in** G^2**"**.

Now, the logarithmic space can be used to implement whole the procedure discussed above and **this logarithmic space can be achieved by checking if there are edges between** (u,r) and (r,q).

Comment

Step 2 of 2

Now, $UPATH \in L$ **can be shown by taking an assumption** $BIPART \in L$. The condition of bipartite can also be checked by the color of the vertices (which is generated by any algorithm). Now, consider the graph algorithm (which is used to generate the graph) which is given below:

For all vertex q of G

"**Two copy of** q_1, q_2 **is made in** H**"**.

For all edge (q,r) of G

"**Put the edges** (q_1,r_2) **and** (q_2,r_1) **in** H".

By using the facts which is discussed above, H will be **bipartite** only if every vertices of q_1 and q_2 are colored red and blue respectively. It gives that there exists no monochromatic edge. So, a cycle of odd length cannot exist. Suppose, a moment (G,m,n) of $BIPART$ is given.

• Now, by connecting the edges (m_1,n_1) and (m_2,n_2), the graph H_1 and H_2 can be obtained respectively. If there exists a path from m to n in G, then minimum one of the graphs H_1 and H_2 **must consists a cycle of odd length**.

• However, if n and m are not joined in G, then both H_1 and H_2 will show bipartite behavior. In H_1, m_1 and n_1 will be in different component.

• Since, bipartite behavior is shown by both of these components, it exists also after adding the edge (m_1,n_1) and acquiring H_1, the graph remains bipartite**. In the same way,** H_2 **must be bipartite.**

Therefore, it can be said that $UPATH \in L$ **and therefore,** $BIPART \in L$ **or In other words "** $\overline{BIPARTITE} \leq_L UPATH$**"**.

Comment

Problem

Recall that a directed graph is ***strongly connected*** if every two nodes are connected by a directed path in each direction. Let

$$STRONGLY\text{-}CONNECTED = \{\langle G\rangle \mid G \text{ is a strongly connected graph}\}.$$

Show that *STRONGLY-CONNECTED* is NL-complete.

Step-by-step solution

Step 1 of 4

Specified that

A directed graph is strongly connected if every two nodes are connected by a directed path in each direction.

Let $STRONGLY\_CONNETED = \{\langle G\rangle \mid G$ is a strongly connected graph$\}$

We have to show that $STRONGLY\_CONNETED$ is NL _ complete

NL -completeness: A language '*B*' is *NL*_ complete if

1. $B \in NL$, and

2. Every *A* in *NL* is log space reducible to *B*.

So, to show that $STRONGLY\_CONNECTED$ is *NL* _ complete

We need to prove the 2 conditions of NL _ completeness.

Comment

Step 2 of 4

(1) $STRONGLY\_CONNECTED \in NL$:

We know that

"*NL* is the class of languages that are decidable in logarithmic space on non-deterministic Turing machine (NTM)„

So to prove $STRONGLY\_CONNECTED \in NL$, we need to construct a *NTM N* that decides $\overline{STRONGLY\_CONNECTED}$ in logarithmic space.

The construction of *N* is as follows:

$N_1 =$ "On input $\langle G\rangle$:

1. Select two nodes *a* and *b* non-deterministically.

2. Run $PATH(a,b)$.

• If it rejects, then the graph is not strongly connected, so accept.

• Otherwise, reject.

Since storing the node numbers *a* and *b* only takes log space, and *PATH* uses only log space, so

$\overline{STRONGLY\_CONNECTED} \in NL$.

We know that $NL = CONL$.

Thereof $STRONGLY\_CONNECTED \in NL$

Comment

Step 3 of 4

(2) Next we must show that every language in *L* is log space reducible to $STRONGLY\_CONNECTED$.

We do this by reducing $PATH$ to $STRONGLY\_CONNECTED$

The $NTM\ \ N_2$ will do this procedure.

$N_2 =$ "On input $\langle G,s,t \rangle$, where *G* is a graph and *s, t* are vertices in *G*

1. Copy all of *G* onto the output tape

2. Additionally for each node *i* in *G*.

3. Output on edge from *i* to *s*.

4. Output an edge from *t* to *i*. "

Comment

Step 4 of 4

This algorithm only needs log space to store the counter for *i*

• If there is a path from *s* to *t*, then the constructed graph is strongly connected because every node can now get to every other node by going through the path $s-t$.

• If there is no path from *s* to *t*, then the graph is not strongly connected. Because the only additional edges in the constructed graph go into *s* and out of *t* , so there can be no new ways of getting from *s* to *t*.

So, from (1) and (2)

$STRONGLY\_CONNECTED$ is *NL* complete

Comment

Problem

Let

$$BOTH_{\text{NFA}} = \{\langle M_1, M_2\rangle \mid M_1 \text{ and } M_2 \text{ are NFAs where } L(M_1) \cap L(M_2) \neq \emptyset\}.$$

Show that *BOTH*$_{\text{NFA}}$ is NL-complete.

Step-by-step solution

Step 1 of 3

Consider the given language, $BOTH_{NFA} = \{\langle M_1, M_2\rangle | M_1$ and M_2 are NFAs where, $L(M_1) \cap L(M_2) \neq \varnothing\}$. A non-deterministic log space algorithm is given here, which determines that "there are some strings which both M_1 and M_2 accepts". Consider that q_1 and q_2 are the number of states in both NFAs accordingly.

M = "On input $\langle M_1, M_2\rangle$, where M_1, M_2 are NFAs:

1. Place markers on the start states of both of the NFAs.

2. Repeat $2^{q_1 \times q_2}$ times:

3. In each step select non-deterministically an input symbol and accordingly change the marker positions on M_1 and M_2's states to simulate reading of that symbol.

4. **Accept;** if some string is found in stage 2 and 3 on which both M_1 and M_2 attends any accepting state that is if at some point on input string both the markers are on accepting states of M_1 and M_2. Otherwise **Reject**."

The only space requirement for this algorithm is to store the location of the markers and the repeat counter. This requirement will be possible in logarithmic space. Therefore, $BOTH_{NFA} \in NL$. Next user need to prove that $BOTH_{NFA}$ is also NL-Hard.

Comment

Step 2 of 3

Now, consider a log space reduction from some language A in NL to $BOTH_{NFA}$. For this purpose, consider a Nondeterministic TM M which decides A in $O(\log n)$.

• Given an input string ω, construct two different NFAs M_1 and M_2 in log space which accepts ω if and only if M accepts ω.

• The states of M_1 and M_2 are the configurations of M on ω. For configuration q_1 and q_2 of M on ω, the pair (q_1, q_2) are two states both in M_1 and M_2 if q_2 is the possible next configuration of M starting from q_1.

• The above statement shows, if M's transition function follows that state q_1 together with its state symbols under corresponding input symbol.

• Also, each of the work tape heads can find the next state and also the head actions to take from q_1 to q_2, then q_1 and q_2 are two states in both M_1 ands M_2.

This configuration reduces A to $BOTH_{NFA}$ because whenever M accepts an input, some its computational branch accepts some state transition in both of the machines M_1 and M_2.

Comments (1)

Step 3 of 3

Again conversely, if M_1 and M_2 accepts a string ω some computational branch follows in M and accepts ω. Now to show that the given reduction works in log space, consider the log space transducer that outputs $\langle M_1, M_2\rangle$ on input ω.

• Consider describing M_1 and M_2 by listing their states. Here each state is a configuration of M on input of ω and it can be denoted by $c \log n$ space where c is some constant.

• Now, it can be said, transducer attends via all possible strings of length $c \log n$ sequentially and checks if each one is a legal configuration of M on ω and outputs only those which passes the checking.

• Hence, it is sufficient to verify in log space that any configuration q_1 of M on input ω can find another configuration q_2. This is because, the transducer only needs to check for the actual tape contents under the each head locations given in q_1 and used to determine that M's transition function would produce q_2 as a result.

• The transducer verifies all pairs (q_1, q_2) in each turn to find the states in M_1 and M_2. **Hence it is proved that** $BOTH_{NFA}$ **is** $NL-Complete$.

Comment

Problem

Show that A_{NFA} is NL-complete.

Step-by-step solution

Step 1 of 1

Consider

$A_{NFA} = \{\langle M, w\rangle \mid M \text{ is } NFA \text{ and } M \text{ accepts } w\}$ and
PATH$=\{\langle G, s, t\rangle \mid G \text{ is any directed graph also } G \text{ has path from } s \text{ to } t\}$

Firstly it is required to show A_{NFA} is NL.

Use a verifier /certificate definition of NL, similar to verifier /certificate of NP. A_{NFA} is in NL if there is some deterministic type Turing machine V such that V uses $O(\log n)$space and $\langle M, w\rangle$in A_{NFA} iff there is any certificate c such that V accepts $\langle\langle M, w\rangle, c\rangle$. The certificate will be a path in the underlying graph of NFA M, and the verifier operates as given below:

• The states in the path denoted by $(p_0, \ldots\ldots, p_n)$.

• Check that the $p_0 = q_0$.

• For each $i \in \{1, \ldots., n\}$, check that $\delta(p_{i-1}, w_i) = p_i$.

• Check that p_n is accepting state.

• If any checks fail, then reject. Otherwise accept.

This algorithm checks every condition which is required by the definition of "M accept w", if it has M accept $w$iff V accept $\langle\langle M, w\rangle, (p_0, \ldots, p_n)\rangle$ for any path $(p_0, \ldots, p_n)$.Therefore it is concluded that A_{NFA} is in NL.

Now it is required to prove that $PATH \leq_M^L A_{NFA}$. Since Path is $NL-complete$, this shows that A_{NFA} is $NL-complete$.

The reduction of mapping $\langle G, s, t\rangle \to \langle M, \varepsilon\rangle$, where M is NFA whose graph is G with $\in$ labels on each edge, where $s$is initial state, and t is its accept state. If G has path from s to t, then there is few sequence of the vertices $(v_1, v_2, v_3, \ldots., v_k)$ so that

- $v_1 = s$
- $(v_i, v_{i+1}) \in E(G)$ for each $i \in \{1, 2, 3 \ldots., k-1\}$
- $v_k = t$

Thus, $(v_1, v_2, v_3, \ldots, v_k)$is sequence of the state in M such that

- v_1 is initial state,
- $\delta(v_i \in) = v_{i+1}$ for each $i \in \{1, 2, 3, \ldots, k-1\}$
- v_k is the terminal state,

Three conditions which are necessary to conclude that M accepts ε. Conversely, if G has not any path from point s to point t, then the path will violate one of three given conditions for having the path, so any state sequence will violate any one of three conditions to accept. Therefore $PATH \leq_M^L A_{NFA}$(and thus A_{NFA} is $NL-complete$).

Comment

Problem

Show that E_{DFA} is NL-complete.

Step-by-step solution

Step 1 of 1

• E_{DFA} is in $co-NL$ (a sufficient certificate is an accepted string-one always exists of length less than the number of states), thus it is in NL. Now it is required to prove that $NL-\text{hardness}$ by a reduction from $\overline{PATH}$ (which is $co-NL-\text{complete}$, and thus $NL-\text{complete}$).

• The idea of the reduction from $\overline{PATH}$ is simple. Given $G=(V,E)$ and vertices s,t and will construct a DFA having state graph is G, s is the initial state and t is the final state. Then the language of this DFA is empty if and only if t is not reachable from s $\left(\text{that is,}\langle G,s,t\rangle \notin \overline{PATH}\right)$.

• So, it sufficient to show this is possible with a log-space transducer.

• First, for each vertex, count the maximum out-degree d. Our alphabet will have $|\Sigma|=d$ (in particular, will take $\Sigma=\{1,.....,d\}$ where one can write each number in $\log d=O(\log m)$ bits). This computation is easily done in log-space, by counting at each vertex and only maintaining a maximum value.

• Each vertex u will be translated (in order vertices appear on the tape) into a state, and each edge (u,v_i) (in the order the edges out of u appear on the tape) into a transition rule $\delta(u,i)=v_i$.

• If reach the end of the list of vertices without giving transitions for all d letters, add copies of the last edge visited so that u has transitions for all letters. This step is done in log-space, since at most one vertex and two edges at a time.

• It is then easy to set s to be the start state, and t to be the final state. So it is possible to do this reduction in log-space, and hence, E_{DFA} is NL-complete.

Comment

Problem

Show that *2SAT* is NL-complete.

Step-by-step solution

Step 1 of 4

NL – completeness:- A language *B* is *NL*- complete if

1. $B \in NL$, and

2. Every *A* in *NL* is log space reducible to *B,* that is *B* is $NL-$hard.

Now we have to prove that 2*SAT* is *NL* –complete.

We know that $2SAT = \{ \langle \phi \rangle \mid \phi$ is a satisfiable Boolean formula in $2cnf \}$.

We know the fact that $NL = CONL$

Thus if we prove that $\overline{2SAT}$ is $NL-$complete, then that implies that 2*SAT* is $NL-$complete.

Comment

Step 2 of 4

To prove $\overline{2SAT}$ is *NL*- complete, we have to prove the two conditions of $NL-$completeness.

(i) $\overline{2SAT} \in NL$;

In order to prove $\overline{2SAT} \in NL$ construct the graph $NTM \quad M$ as follows:

$M =$ "On input ϕ :

(1) Construct graph *G* such that ϕ is not satisfiable if and only if there two vertices u, v

the paths uv and vu are in *G*.

(2) For each variable *x* create a node labeled *x* and another labeled $\bar{x}$.

(3) Now non-deterministically select a vertex *x* in *G*.

(4) Check if *G* contain both *x* $x\bar{x}$ and $\bar{x}x$ paths using $NL-$algorithm for $PATH$.

(5) If both paths exist, then accept

(6) Otherwise reject ".

The entire construction is done in log – space and *M* decides $\overline{2SAT}$ in log space only.

Thus $\overline{2SAT} \in NL$

Comment

Step 3 of 4

(ii) $\overline{2SAT}$ is $NL_$hard:

We do this by reducing path to $\overline{2SAT}$.

Construction of ϕ :

• Given G, s, t we have to construct a $2-cnf$ formula ϕ such that *G* has an $s,t-$path if and only if ϕ is unsatisfiable.

• Let $v(G) = \{s, t, y_1, ... y_n\}$

• The resulting formula ϕ consists of $m+1$ variables $x, y_1, ... y_m$

• The vertex *s* is identified with *x* and the vertex *t* with $\bar{x}$.

• The ıasses of ϕ will be $x \vee x$ and $\bar{u} \vee u$ every edge (u,v) of *G*.

Comment

Step 4 of 4

→ In order to show that this construction works, first assume that *G* has an s,t path. Let this path be $s,u_1,...u_k t$. Then ϕ contains the clause $(\bar{x} \vee u_1),(\bar{u}_1 \vee u_2)...(\bar{u}_k,\bar{x})$.

If we designs $x=1$, at least one of these clauses is not satisfied on the other hand, if $x=0$, then the clause $(x \vee x)$ of ϕ is not satisfied. So ϕ is zero for any assignment of $x,y_1,...y_m$.

→ Conversely, suppose that *G* contains no $s-t$ path.

• Let *U* be the set of vertices in *G* reachable from s,V the set of vertices from which *t* is reachable and $w:V(G)(U \vee V)$.

By hypothesis and construction U,V,W are pair wise disjoint, and there are no edges from *U* to $V \cup W$ or from $U \cup W$ to V. let us assign the rue value to every variable in $U \cup W$ and the false value to every variable in *V*. it is now simple to check that ϕ is satisfied for this truth assignment.

Comment

Problem

Let CNF_{H1} = { $\langle \phi \rangle$ I ? is a satisfiable cnf-formula where each clause contains any number of positive literals and at most one negated literal. Furthermore, each negated literal has at most one occurrence in ?}. Show that CNF_{H1} is NL- complete.

Step-by-step solution

Step 1 of 3

Consider the $\mathrm{CNF}_{\mathrm{H1}}$, which is defined as:

$\mathrm{CNF}_{\mathrm{H1}}$ = {<∅>| ∅ is a satisfiable cnf-formula where each clause contain nay number of positive literals and at most one negative literals}

Now, it is known that $CNF_{H1} \in NL$. It can be proved by using the following approach: Situation is quite clear as all user needs to find the d**ependability** of $\mathrm{CNF}_{\mathrm{H1}}$ on P and NL.

Comment

Step 2 of 3

Dependability can be viewed as:

1. Consider that $\mathrm{CNF}_2 \in NL$

Consider first situation of ϕ and consider it belongs to some x and if there is any $\neg x$ then reject.

Consider one more situation of ϕ that will be belonging to some other variable let's say y and here situation will rejected if there is any $\neg y$ otherwise accepted.

• Two situations are discussed here remove those situation from ϕ and relate those two situations, let's say M and N, to ϕ and call the result. This way it is proved that $\mathbf{CNF_2} \in NL$.

Comment

Step 3 of 3

2. Consider another situation $\mathrm{CNF}_3 \in NL$

Consider first situation of ϕ and consider it belongs to some p and if there is any $\neg p$ then reject.

Consider one more situation of ϕ that will be belonging to some other variable, let's say q and r here situation will rejected if there is any $\neg q$ otherwise accepted.

• Two situations are discussed here remove those situation from ϕ and relate those two situations, let's say A and B, to ϕ and call the result. This way it is proved that $\mathbf{CNF_3} \in NL$.

Similarly**, this situation will be held true for $\mathbf{CNF_{H1}}$**. In other words**, it can be said that"** $CNF_{\mathrm{H1}} \in NL$**," or** $\mathbf{CNF_{H1}}$ is $NL-complete$.

Comment

Problem

Give an example of an NL-complete context-free language.

Step-by-step solution

Step 1 of 3

The statement of Ginsburg theorem state that, "Suppose G is used to generate a language $L(G)$ where G is defined as a reducible context free grammar. Then $|L(G)\cap\{0,1\}^n|$ will show a polynomial behavior in n if and only if for each non-terminal $z, l(z)$ and $r(z)$ are commutative".

Comment

Step 2 of 3

Now, suppose $M\subseteq\{0,1\}^*$ be a language which **is context free and exponential** in size. The above context free language shows a $NP-complete$ behavior.

• To perform this, suppose, D is defined as **a reduced-free grammar** for M. Here, M_n will not be in polynomial of n because M consists an **exponential size**.

• Now from the above given theorem, it can be said that there exists a nonterminal and for each non-terminal $Z, l(Z)$ and $r(Z)$ are **commutative**.

Comment

Step 3 of 3

Consider, $l(Z)$ (where, $a_1, a_2 \in l(Z)$) is not showing the commutative property and $a_1, a_2 \neq a_2, a_1$. Hence, there exists a position k such that $(a_1, a_2)_k = 0$ and $(a_2, a_1)_k = 1$.

• As from the above explanation, there exists $(b_1, b_2)\in\{0,1\}^*$ in such a way that $Z \succ^* a_1 Z b_1$ and $Z \succ^* a_2 Z b_2$. An arbitrary $1s$ and $0s$ can be generated at the position $k\cdot|a_1a_2|+i$ for any k, by applying either $Z \succ^* a_1a_2Zb_2b_1$ or $Z \succ^* a_2a_1Zb_1b_2$, k times.

• Now, to reach Z from M_0, user can use $M_0 \succ^* aZb$. Again, to acquire a word in $\{0,1\}^*$ for some $x, y, w \subseteq \{0,1\}^*$, $Z \succ^* w$ can be used.

• Hence from the above discussion, $I := \{|x|+k\cdot|a_1a_2|+i : 0 \le k \le n-1\}$ is of size n can be obtained if N is set as $N := \{|x|+|y|+|w|+n(|a_1a_2|+|b_1b_2|)\}$ and also I can be shattered by M_N.

• All the calculations (which are done above) take a time in $O(n)$ and N is linier in n and it is already known that $S-SAT$ is $NP-hard$ and $S-SAT \in NP$. Hence, it shows $NP-complete$ behavior.

Hence the above explanation shows, the context free language $M\subseteq\{0,1\}^*$ is $NP-complete$.

Comment

Problem

Define *CYCLE* = { $\langle G \rangle$ | G is a directed graph that contains a directed cycle}. Show that *CYCLE* is NL-complete.

Step-by-step solution

Step 1 of 2

NL – completeness:

A language *B* is *NL*- complete if

1. $B \in NL$, and

2. Every *A* in *NL* is log space reducible to *B*.

Given that

$CYCLE = \{ \langle G \rangle \mid G$ is a directed graph that contains a directed cycle}.

Now we have to prove that CYCLE is *NL* – complete.

1. $\underline{CYCLE \in NL}$:

We know that

"*NL* is the class of languages that are decidable in logarithmic space on a Non deterministic Turing machine"

Let *N* be the nondeterministic Turing machine that decides $CYCEL$.

The construction of *N* is as follows:

$N =$ "On input $\langle G \rangle$ (*G* is an directed graph):

1. Select a vertex *u* as starting vertex.

2. Select an edge (u,v) from *u*.

3. Run $PATH(u,v)$.

4. Start traversal through (u,v), if we come back to *u* through on edge different that

(u,v), then direct cycle will exist

5. Otherwise, reject."

• Since all vertices and all the edges are enumerated in log space *N* decides CYCLE in logarithmic space.

• Therefore, $CYCLE \in NL$.

Comment

Step 2 of 2

2. $\underline{PATH \leq_L CYCLE}$:

• Now we have to reduce $PATH$ to $CYCLE$.

• For that we have modify the *PATH* problem instance $\langle G,s,t \rangle$ by adding an edge from *t* to *s* in *G*.

• If path exists from *s* to *t* in *G* then direct cycle will exist in modified *G*.

• But some cycles may already be present in *G*.

• So, so solve that problem change *G* so that it contains no cycles.

• A leveled directed graph is non where the nodes are divided into graphs, $A_1, A_2, ...A_k$ called levels.

• Only edges from one level to next higher level are permitted.

• G' is the leveled graph of *G* which has two nodes *s* and *t*, and *m* nodes in total.

• Draw an edge from node *i* at each level to node, *j* in the next level if *G* contain an edge from *i* to *j*.

• Also, Draw on edge from node 1 in each level to node *I* in the next level.

• Let s' be the node *s* in the first level and t' be the node *t* in the next level.

• Graph *G* contains a path from *s* to *t* iff G' contain a path from s' to t'.

• If we add an edge from t' to s' in G' then reduction from *PATH* to *CYCLE* will be obtained, this reduction is implemented in log space.

Therefore (1) and (2) $CYCEL$ is $NL-$ complete.

Comment