Problem

Prove that TIME(2^n) = TIME(2^{n+1}).

Step-by-step solution

Step 1 of 1

Proof of $\mathbf{TIME}\left(2^n\right)=\mathbf{TIME}\left(2^{n+1}\right)$

Classification of computation time can be done on the basis of the minimum time required by the most efficient algorithm to determine upper and lower bounds.

Usually Big-O notation is used to state the lower and upper bond which **hides smaller terms** and **constant factors.**

• Big-O notation is used to define the **time complexity** classes. So there is no effect of constant factors used in complexity calculation.

• Therefore the time complexity of the function 2^{n+1} in big-O notation is O $\left(2^n\right)$.

• Thus, $\mathbf{A}\in\mathbf{TIME}\left(2^n\right)$ if and only if $\mathbf{A}\in\mathbf{TIME}\left(2^{n+1}\right)$.

• Hence, $\mathbf{TIME}\left(2^n\right)=\mathbf{TIME}\left(2^{n+1}\right)$

Comment

Problem

Prove that TIME(2^n) $\subsetneq$ TIME(2^{2n}).

Step-by-step solution

Step 1 of 1

Proving $\mathbf{TIME}\left(2^{n}\right) \subsetneq \mathbf{TIME}\left(2^{2n}\right)$

The containment $\mathrm{TIME}\left(2^{n}\right) \subseteq \mathrm{TIME}\left(2^{2n}\right)$ holds because $2^{n} \leq 2^{2n}$. Now, consider the **time hierarchy theorem** which says that, "if f, g are time-constructible functions and $\mathrm{f}(\mathrm{n}) \log \mathrm{f}(\mathrm{n})=\mathrm{O}(\mathrm{g}(\mathrm{n}))$ then $\mathrm{DTIME}(\mathrm{f}(\mathrm{n})) \subsetneq D\mathrm{TIME}(\mathrm{g}(\mathrm{n}))$.

Thus, the above containment is proper by virtue of the time hierarchy theorem as discussed.

• A Turing machine can write the number 1 followed by 2n 0s in $O\left(2^{2n}\right)$ time. So, the function 2^{2n} is time constructible

• Hence, the time hierarchy theorem guarantees that a language A exists that can be decided in $O\left(2^{2n}\right)$ time but not in $o\left(2^{2n} / \log 2^{2n}\right)=o\left(2^{2n} / 2n\right)$ time.

• Therefore, $A \in TIME\left(2^{2n}\right)$ but $A \notin TIME\left(2^{n}\right)$.

From the above explanation it can be said that $\mathrm{TIME}\left(2^{n}\right) \subsetneq \mathrm{TIME}\left(2^{2n}\right)$.

Comment

Problem

Prove that NTIME(n) $\subsetneq$ PSPACE.

Step-by-step solution

Step 1 of 1

$\mathrm{NTIME}(\mathrm{n})$ is strict subset of $\mathrm{PSPACE}(\mathrm{n})$

At most $\mathrm{t}(\mathrm{n})$ tape cells on each branch can be used by any Turing machine that operates in time $\mathrm{t}(\mathrm{n})$ on each computation branch. So, it can be stated that $\mathrm{NTIME}(\mathrm{n}) \subsetneq \mathrm{NSPACE}(\mathrm{n})$

• Now, consider the **Savitch's theorem** which says that: "Let $f : N \to R$ be a function, with $f(n) \geq n$ then $\mathrm{NSPACE}(f(\mathrm{n})) \subseteq \mathrm{SPACE}\left((f(\mathrm{n}))^2\right)$".

Therefore according to Savitch's theorem $\mathrm{NSPACE}(\mathrm{n}) \subseteq \mathrm{SPACE}(\mathrm{n}^2)$.

• Now, consider the **space hierarchy theorem** which says that "if g is space-constructible ($1^n \to 1^{g(n)}$ can be computed in space $O(g(n))$), $f(n) = O(g(n))$ then $\mathrm{SPACE}(\mathrm{f}(\mathrm{n})) \subsetneq \mathrm{SPACE}(\mathrm{g}(\mathrm{n}))$.

Therefore, according to space hierarchy theorem it can be said that $\mathrm{SPACE}(\mathrm{n}^2) \subsetneq \mathrm{SPACE}(n^3)$. The result follows because $\mathrm{SPACE}(\mathrm{n}^3) \subseteq \mathrm{PSPACE}$.

From the above explanation, it can be said that $\mathrm{NTIME}(\mathrm{n}) \subsetneq \mathrm{PSPACE}(\mathrm{n})$.

Comment

Problem

Show how the circuit depicted in Figure 9.26 computes on input 0110 by showing the values computed by all of the gates, as we did in Figure 9.24.

Figure 9.26

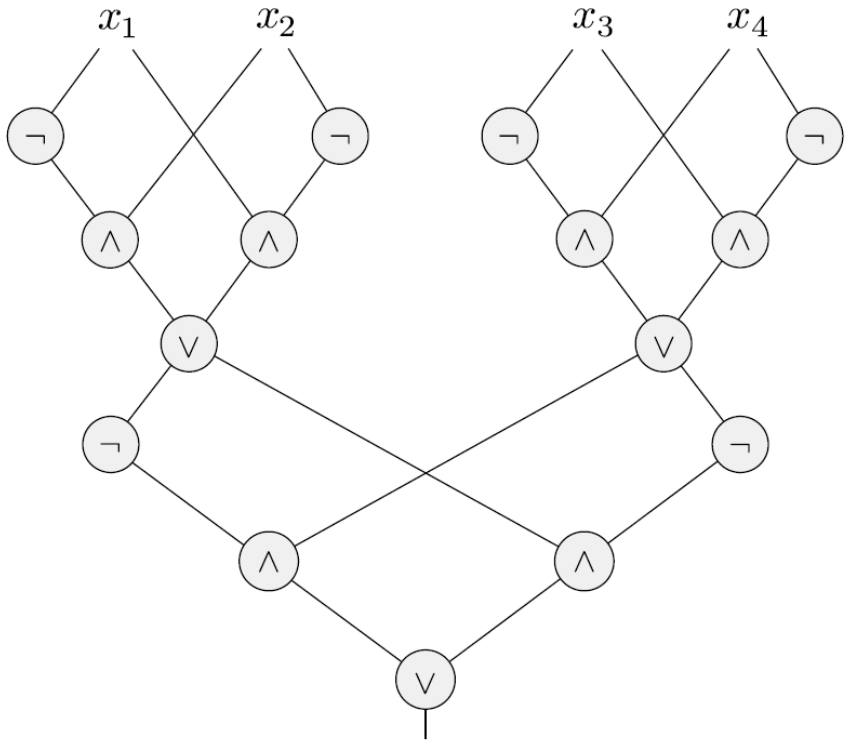


FIGURE 9.26
A Boolean circuit that computes the parity function on 4 variables

Figure 9.24

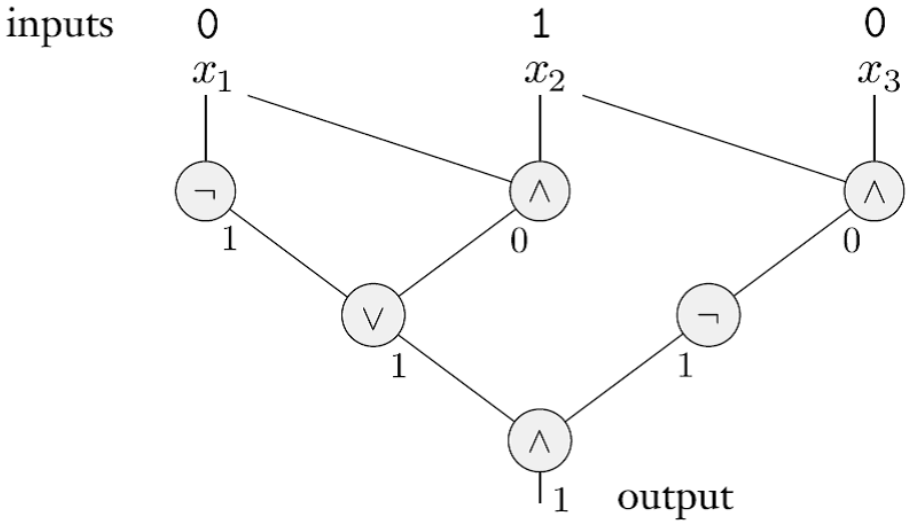


FIGURE 9.24
An example of a Boolean circuit computing

Step-by-step solution

Step 1 of 6

Representation of the parity functional circuit

The *n*-input *parity* function $parity_n : \{0,1\}^n \to \{0,1\}$ gives an output 1 if an odd number of 1's appear in the input variable and otherwise gives 0. The circuit depicted here (refer figure 9.26) is used as a parity functional circuit on four variable.

In this parity functional circuit, a number of gates are used. Therefore, first of all, the working of gates will be discussed here to understand the process performed in that circuit.

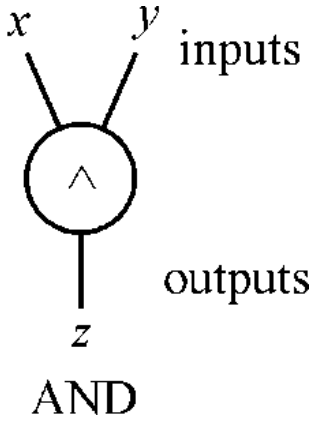
Comment

Step 2 of 6

The Truth table for and operator is as follows:

Input	Output z
x	y
0	0	0
1	0	0
0	1	0
1	1	1

The following figure shows the how input and output can be taken in the above table (for and operator):



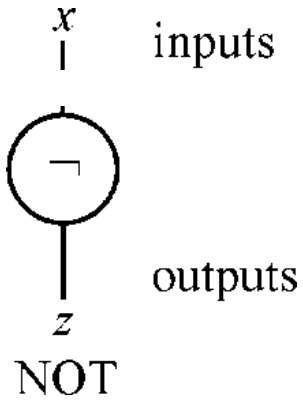
Comment

Step 3 of 6

The Truth table for not operator is as follow:

Input(x)	Output(z)
0	1
1	0

The following figure shows the how input and output can be taken in the above table (for not operator):



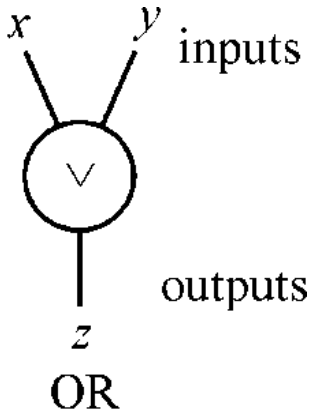
Comment

Step 4 of 6

The Truth table for or operator is as follow:

Input		Output z
x	y	
0	0	0
0	1	1
1	0	1
1	1	1

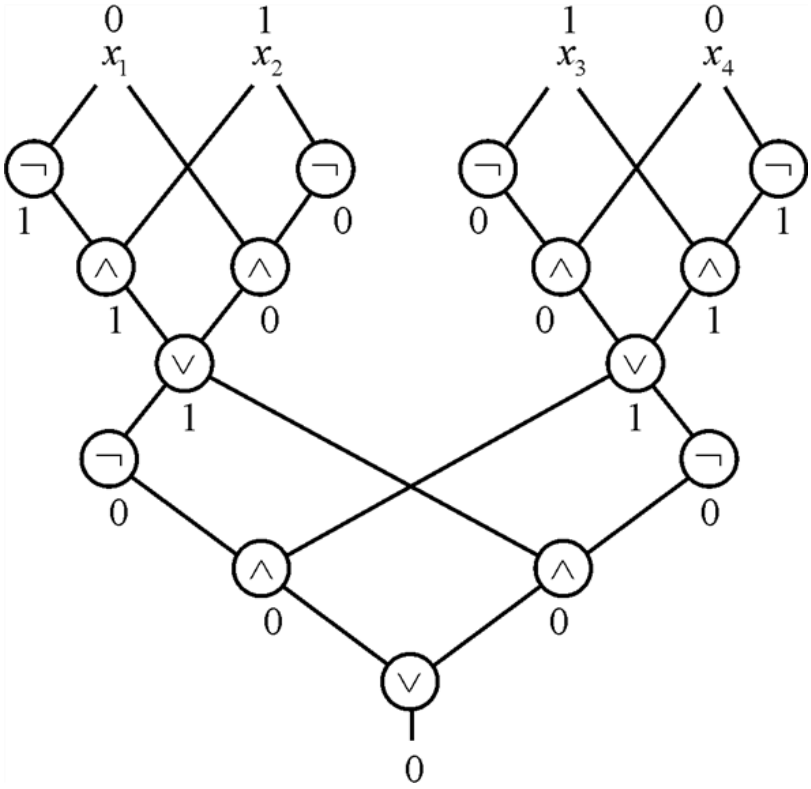
The following figure shows the how input and output can be taken in the above table (for or operator):



Comment

Step 5 of 6

Consider the figure given below:



Above figure shows the functional parity circuit on 4 variables. Here 0110 is taken as an input variable. Now AND gate, OR gate and NOT gate applied on the input in the

Comment

Step 6 of 6

same manner as it is discussed above. The final output is 0 because here even number of 1's has taken (as the definition of parity function says).

Comment

Problem

Give a circuit that computes the parity function on three input variables and show how it computes on input 011.

Step-by-step solution

Step 1 of 7

The *n*-input **parity function** is the Boolean function $f : \{0,1\}^n \to \{0,1\}$ gives an output 1 if an odd number of 1's appear in the input variable and otherwise gives 0. Here f is defined as:

$f(x) = x_1 \oplus x_2 \oplus x_3 \oplus ... \oplus x_n$, where $\oplus$ denotes an EX-OR gate operation.

• In this parity functional circuit, a number of gates are used. Therefore, first of all, the working of gates will be discussed here to understand the process performed in that circuit.

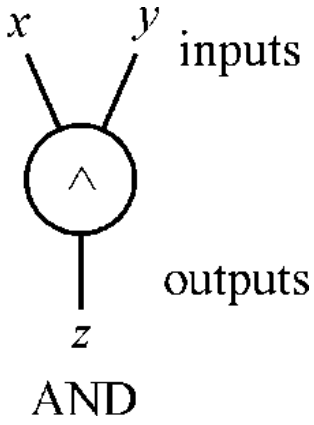
Comment

Step 2 of 7

The Truth table for **and** operator is as follow:

Input	Output z
x	y
0 1 0 1	0 0 1 1

The following figure shows the how input and output can be taken in the above table (for **and** operator):



Comment

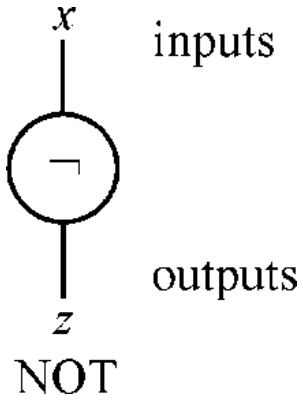
Step 3 of 7

The Truth table for **not** operator is as follow:

Input(x)	Output(z)

0 1	1 0

The following figure shows the how input and output can be taken in the above table (for **not** operator):



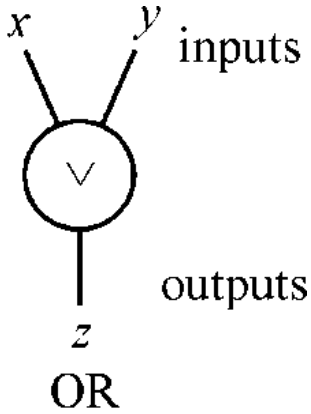
Comment

Step 4 of 7

The Truth table for **or** operator is as follow:

Input	Output z	
x	y	
0 0 1 1	0 1 0 1	0 1 1 1

The following figure shows the how input and output can be taken in the above table (for **or** operator):



Comment

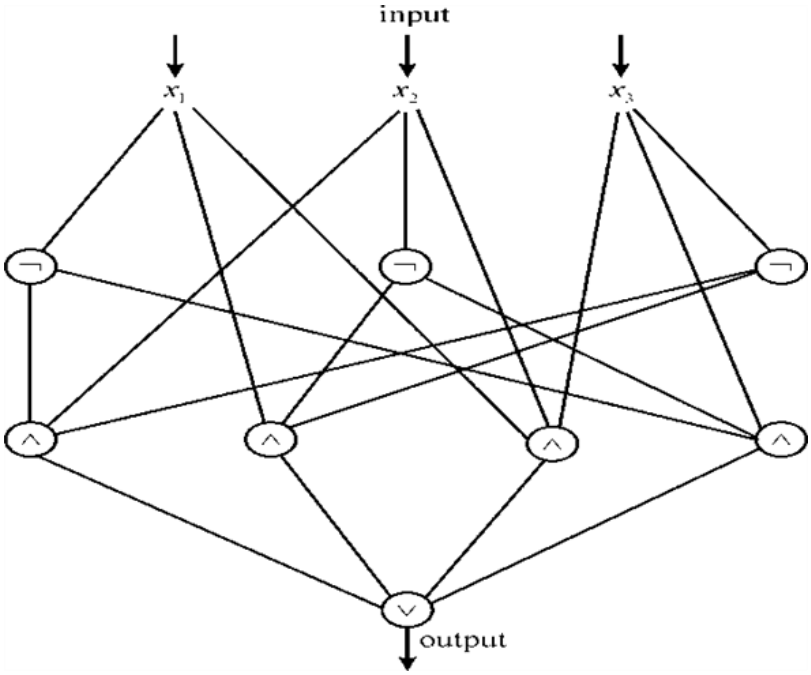
Step 5 of 7

From the definition of **parity function**, the n-input parity functional circuit can be obtained by applying an EX-OR gate operation on the n- inputs.

• For three input variables, the parity function may be defined as:

$$f(x) = x_1 \oplus x_2 \oplus x_3$$
$$= \left[(x_1 \cdot \overline{x}_2) + (\overline{x}_1 \cdot x_2) \right] \oplus x_3$$
$$- \left[x_1 \cdot \overline{x}_2 \cdot \overline{x}_3 + \overline{x}_1 \cdot x_2 \cdot \overline{x}_3 + x_1 \cdot x_2 \cdot \overline{x}_3 + \overline{x}_1 \cdot \overline{x}_2 \cdot x_3 \right]$$

Consider the figure which is given below:



•

Comment

Step 6 of 7

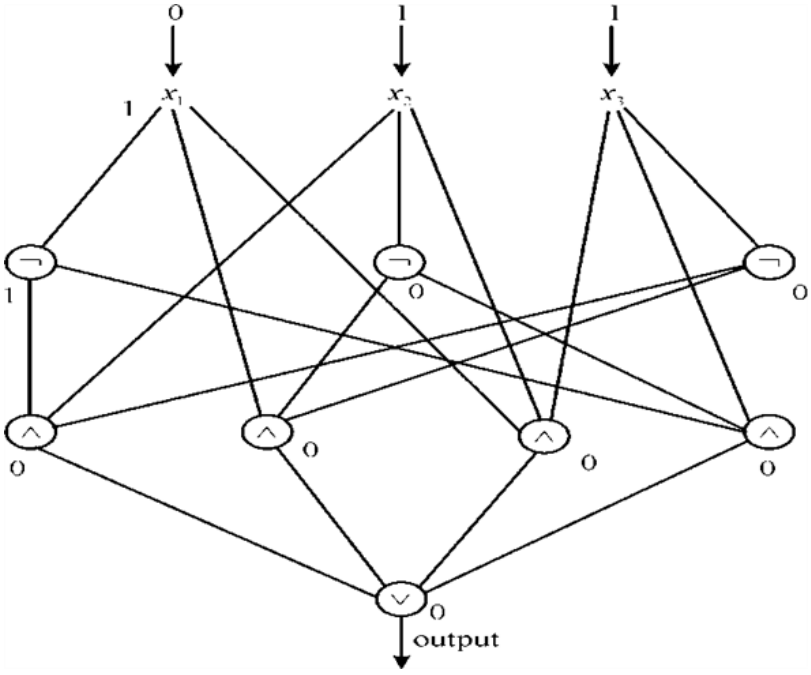
Above figure shows the functional circuit on three input variable. It shows how three variables are taken as an input, and how an EX-OR gate operation is applied on these inputs.

• An EX-OR gate consists of an AND, NOT and OR gate operations.

Comment

Step 7 of 7

Consider the figure which is given below:



Above figure shows, the computation performed by three input parity functional circuit when 001 is taken as an input. After taking an input , an EX-OR operation is applied as discussed above.

• The output given by the parity functional circuit is 0 because even number of 1's are present in the input.

• which follow the definition of parity funtional circuit(that is, it gives an output 1 if an odd number of 1's appear in the input variable and otherwise gives 0).

Comment

Problem

Prove that if A ? P, then P^A = P.

Step-by-step solution

Step 1 of 1

Given:

Language *A* belongs to the polynomial type oracle Turing machine *P*.

$A \in P$

Proof:

P^A, is basically the class of the language which is decidable with the polynomial Turing machine which uses language *A*.

Here, it is to be proved that $P^A = P$

Construction:

Language *A* is the device which checks whether the string *w* is the member of *A*.

Suppose string *w* contains the value $\{a^n b^n\}$ and user has given the string $\{aabb\}$ then it is accepted by *A*. If the string $\{aabbb\}$ is specified then it is not expected by the language *A*.

Now, user checks this for the poly-time Turing machine:

Suppose *N* is the poly-time Turing machine, here, construction of the machine is done so that it can be proved that P^A, is basically the class of the language which is decidable with the polynomial Turing machine which uses language *A*.

• Computation path for *N* is done and it is found that it leads to the acceptance of the polynomial.

• If the string is accepted then yes query will execute if it is not accepted then no query will execute.

• Machine *N* checks whether all the string is accepted and rejected

• If the guess is right then string is accepted.

Here, for all the string computation is done and hence it is proved that $P^A = P$

Comment

Problem

Give regular expressions with exponentiation that generate the following languages over the alphabet {0,1}.

A**a.** All strings of length 500

A**b.** All strings of length 500 or less

A**c.** All strings of length 500 or more

A**d.** All strings of length different than 500

e. All strings that contain exactly 500 1s

f. All strings that contain at least 500 1s

g. All strings that contain at most 500 1s

h. All strings of length 500 or more that contain a 0 in the 500th position

i. All strings that contain two 0s that have at least 500 symbols between them

Step-by-step solution

Step 1 of 8

Regular expressions:

Regular expression are the expression by which searching is done very easily. It is used for matching the combination of character in the string.

Regular expression consists of normal characters or the combination of the special character.

Power of an alphabet is donated by $\sum^k$. It is basically the combination of the string of the power length which is k and the power is donated by the symbol $\sum$.

a) Given:

Every string is of the length 500

Regular expression with the exponentiation:

$\sum^{500}$

• Power of an alphabet is donated by $\sum^k$. Here, as the string is of length 500 so value of *k* is assigned to 500.

• It represents the combination of the 500 string with exponentiation.

Comment

Step 2 of 8

b) Given:

Every string is of length 500 or it is less than 500

Regular expression with the exponentiation:

$(\sum \cup \varepsilon)^{500}$

• In the above regular expression ε is donated by the empty string. The union of Exponentiation and ε will be less than or equal to 500.

• If there is an empty string then combination will of less than 500 strings. If there is no empty string then the combination will be equal to 500.

Comment

Step 3 of 8

c) **Given:**

Every string is of length 500 or it is more than 500

Regular expression with the exponentiation:

$\sum^{500}\sum^*$

• In the above regular expression is Σ^* donates all the string which is more than 500. Σ^{500} donate the string of the length 500.

• The multiplication of both the exponentiation will result the string of the length 500 or more than 500.

Comment

Step 4 of 8

d) **Given:**

Every string should have length different than 500.

Regular expression with the exponentiation:

$(\Sigma \cup \varepsilon)^{499} \cup \Sigma^{501}\Sigma^*$

• In the above regular expression is $(\Sigma \cup \varepsilon)^{499}$ donates all the string which is 499 or less than 499 as ε donate an empty string.

• Whereas the regular expression $\Sigma^{501}\Sigma^*$ donates all the string which is equal to 501 and more than 501.

• The union of the both the regular expression gives the string either greater than 500 or less than 500 but not the string of length 500.

Comment

Step 5 of 8

e) **Given:**

String containing total 500, 1's exactly.

Regular expression with the exponentiation:

$(0)^n (1)^{500}$, where $n \geq 0$

• In the above regular expression is $(0)^n (1)^{500}$ donates all the string which contains exactly 500 1's and number of 0's in the string are greater than or equal to 0.

Comment

Step 6 of 8

f) **Given:**

String contains at least 500, 1's.

Regular expression with the exponentiation:

$0^n 1^{500} 1^*$, where $n \geq 0$

• In the above regular expression is $0^n 1^{500} 1^*$ donates all the string which contains 500 1's or more than 500 1's and number of 0's in the string are greater than or equal to 0.

Comment

Step 7 of 8

g) **Given:**

String containing maximum 500 1's

Regular expression with the exponentiation:

$0^n (1 \cup \varepsilon)^{500}$, where $n \geq 0$

• In the above regular expression is $0^n (1 \cup \varepsilon)^{500}$ donates all the string which contains less than or equal to 500 1's and number of 0's in the string are greater than or equal to 0.

• If there is no empty string then the combination will be equal to 500.

Comment

Step 8 of 8

h) **Given:**

String of minimum 500 lengths and the position of 500^{th} position are fixed with 0

Regular expression with the exponentiation:

$$(\Sigma)^{499}(0)(\Sigma)^*$$

• In the above regular expression Σ^* donates all the string which is more than 500. Σ^{499} donate the string of the length 499. 0 is present after writing all the string of length 499 which implies 0 is present in the 500 position.

i) Given:

String containing minimum 500 symbols between two 0's

Regular expression with the exponentiation:

$$(0)(\Sigma^{500})(\Sigma)^*(0)$$

• In the above regular expression there are minimum 500 1's between two 0's.

• Σ^* donate all the string which is more than 500.

Comment

Problem

If R is a regular expression, let $R^{\{m,n\}}$ represent the expression

$$R^m \cup R^{m+1} \cup \cdots \cup R^n.$$

Show how to implement the $R^{\{m,n\}}$ operator, using the ordinary exponentiation operator, but without "· · ·".

Step-by-step solution

Step 1 of 1

Consider the following Regular expression:

$R^m \cup R^{m+1} \cup ... \cup R^n$

Here, user needs to be proved the implementation of $R^{m,n}$ operator with the help of the ordinary operator but without using "…".

This can be proved with the help of computation history.

Computation history is basically the series of configuration that is done by the machine at the time of processing the input.

Turing machine *M* is being considered. If the machine does not halt on the particular input then acceptation and rejection of the string is not done by the Turing machine.

With the help of deterministic machines computation history can be calculated for the particular input.

Thus it can be said that all the states which are there in the Turing machine is basically countable and even the arrangement of the all the symbol present in the tape are countable.

So, the union of the two set in the Turing machine is also said to be countable.

As, it is being proved that all the sequence is countable so exponentiation of this sequence can be written:

$$R^{m,n} = \sum_{m=0}^{n} \cup (A)^n$$

Comment

Problem

Show that if NP = P^{SAT}, then NP = coNP.

Step-by-step solution

Step 1 of 1

If $NP = P^{SAT}$ is assumed then, there is only need to show that $NP = coNP$. It can be conclude directly from the assumption $NP = P^{SAT}$, that $P^{SAT} \subseteq NP$. As $coNP \subseteq P^{SAT}$ is already known, which result in $coNP \subseteq NP$.

• Now it is known that P is closed under the complement operation, so is P^{SAT}, because it can be just swap the reject and accept states. It can be concluded that:

$L \in P^{SAT} \Rightarrow \overline{L} \in P^{SAT}$.

• The given statement can be managed by using the prediction $NP = P^{SAT}$, because if this would be the case any language in P^{SAT} would be in NP and vice versa.

• For which NP also has to be closed under the complement operation $L \in NP \Rightarrow \overline{L} \in NP$, that is just the same as $L \in NP \Rightarrow L \in coNP$ from the definition of $coNP$.

• In other words, from the above explanation it can be said that $NP \subseteq coNP$. Hence $NP = coNP$.

Comment

Problem

Problem 8.13 showed that A_{LBA} is PSPACE-complete.

a. Do we know whether A_{LBA} ? NL? Explain your answer.

b. Do we know whether A_{LBA} ? P? Explain your answer.

Step-by-step solution

Step 1 of 3

It is being given and it is being proved in the previous chapter that $A_{LBA} \in PSPACE$.

PSPACE is basically the class of the language, whether the language belongs to deterministic finite automata or non-deterministic finite automata it is decidable in the polynomial time Turing machine.

• A language can be said *PSPACE* complete if that particular language belongs to *PSPACE*.

• For each and every language *PSPACE* hardness is satisfied.

Comment

Step 2 of 3

Here, user needs to prove that $A_{LBA} \in NL$, but as it is being given that $NP \in NPSPACE$ is *PSPACE*- complete.

With the help of space hierarchy theorem it can be proved that $A_{LBA} \in NL$.

It is being known by the Corollary of the Savitch's theorem that *PSPACE*=*NSPACE*, it implies deterministic space complexity and non-deterministic polynomial space complexity are basically the same.

Space complexity of *PSPACE* is $O(\log n)$ whereas space complexity of the NSPACE is $O(n^k)$ for each and every value of k_i.

So, this implies that language is basically accepted by *NSPACE* but it is not accepted by *NL*.

But, it is being known by the space hierarchy theorem that the language is accepted by PSPACE and also language is accepted by NL.

This implies that $NL \in PSPACE$

As, it is being given that $A_{LBA} \in PSPACE$

Hence, it is proved that $A_{LBA} \in NL$

Comment

Step 3 of 3

Here, user needs to prove that $A_{LBA} \in P$, but as it is being given that $NL \in PSPACE$ is *PSPACE*- complete.

With the help of space hierarchy theorem it can be proved that $A_{LBA} \in P$.

It is being known by the Corollary of the Savitch's theorem that *PSPACE*=*NSPACE*, it implies deterministic space complexity and non-deterministic polynomial space complexity are basically the same.

Poly-time Turing machine is not able to consume space greater than poly-space.

It implies $P \in PSPACE$ and $NP \in NPSPACE$.

As, it is being given that $A_{LBA} \in PSPACE$

Hence, it is proved that $A_{LBA} \in P$

Comment

Problem

Show that the language *MAX-CLIQUE* from Problem 7.48 is in P^{SAT}.

Step-by-step solution

Step 1 of 1

• In a graph G clique are maximum of the size k iff it has k clique size and does not have $k+1$ clique size.

• The language $CLIQUE$ is in NP, and the language $CLIQUE$ is in $co-NP$. Since $NP \in P^{SAT}$ and $CoNP \in P^{SAT}$, $MAXCLIQUE$ is in P^{SAT}.

• Since $P^{SAT} \in NP^{SAT}$, $MAXCLIQUE$ is in NP^{SAT}.

Comment

Problem

Describe the error in the following fallacious “proof” that P ≠ NP. Assume that P = NP and obtain a contradiction. If P = NP, then *SAT* ? P and so for some k, *SAT* ? TIME(n^k). Because every language in NP is polynomial time reducible to *SAT*, you have

$$\mathrm{NP} \subseteq \mathrm{TIME}(n^k). \text{ Therefore, } \mathrm{P} \subseteq \mathrm{TIME}(n^k).$$

Step-by-step solution

Step 1 of 2

Consider the following deceptive “proof “that $\mathrm{P} \neq \mathrm{NP}$.

1. Suppose that $\mathrm{P}=\mathrm{NP}$ and obtain a contradiction.

2. If $\mathrm{P}=\mathrm{NP}$ has given, then for some $k \in \mathrm{N}$ and $\mathrm{SAT} \in \mathrm{P}$, $\mathbf{SAT} \in \mathbf{TIME}\left(\mathbf{n}^{\mathbf{k}}\right)$

3. As every language in NP is polynomial time reducible to SAT ,then $\mathbf{NP} \subseteq \mathbf{TIME}\left(\mathbf{n}^{\mathbf{k}}\right)$

4. Due to the assumption $\mathrm{P}=\mathrm{NP}$ which is taken above, $\mathbf{P} \subseteq \mathbf{TIME}\left(\mathbf{n}^{\mathbf{k}}\right)$.

5. Then $\mathbf{TIME}\left(\mathbf{n}^{\mathbf{k}}\right) \subsetneq \mathbf{TIME}\left(\mathbf{n}^{\mathbf{k+1}}\right)$ is a contradiction with $\mathbf{P} \subseteq \mathbf{TIME}\left(\mathbf{n}^{\mathbf{k}}\right)$.

Comment

Step 2 of 2

The above steps are used to proof of $\mathrm{P} \neq \mathrm{NP}$, but it consists some error. Now, consider the **step 3**, **here the time needed for calculating the reduction is not taken into account. So, to make this proof error free, the time needed for calculating the reduction must be taken in to account**.

Comments (1)

Problem

Consider the function $pad: \Sigma^* \times \mathcal{N} \longrightarrow \Sigma^* \#^*$ define the language pad(A, f) as

$$pad(A, f) = \{pad(s, f(m)) | \text{ where } s \in A \text{ and } m \text{ is the length of } s\}.$$

Prove that if $A \in \text{TIME}(n^6)$, then $pad(A, n^2) \in \text{TIME}(n^3)$.

Step-by-step solution

Step 1 of 1

For any function $f: N \to N$ and language, $pad(A, f)$ is defined as:

$pad(\text{A}, f) = \{pad(s, f(m)) | \text{ where } s \in \text{A and m is the length of } s\}$.

If $\text{A} \in \text{TIME}(n^6)$ is given then, it is supposed that M be a machine that decide A in time n^6.

• Now a machine M' can be considered for $pad(A, n^2)$ that on input x, check if x is of the format $pad(w, |w|^2)$ for some string $w \in \sum^*$. Input x will be rejected if it will not matched. Otherwise, simulate M on w.

• The running time of machine M' is $O(|x|^3) + O(|w|^6) = O(|x|^3)$.

Hence, it can be said that $pad(\text{A}, n^2) \in \text{TIME}(n^3)$.

Comment

Problem

Prove that if NEXPTIME ≠ EXPTIME, then P ≠ NP. You may find the function pad, defined in Problem 9.13, to be helpful.

Step-by-step solution

Step 1 of 1

If $\mathbf{EXPTIME} \neq \mathbf{NEXPTIME}$ then $\mathbf{P} \neq \mathbf{NP}$ can be proved by taking its contra positive . If $\mathbf{P} = \mathbf{NP}$ is assumed then $\mathbf{EXPTIME} = \mathbf{NEXPTIME}$ will have to proof.

• Suppose $L \in \mathrm{NTIME}\left(2^{2^n}\right)$ then the following language $L_{\text{pad}} = \left\{ < x, 1^{2|x|^c} > : x \in L \right\}$ is in NP (in fact in $\mathrm{NTIME}(n)$). This process of adding a string of symbols to each string in the language is called **padding**.

• Hence, if $\mathbf{P} = \mathbf{NP}$ then L_{pad} is in P but if L_{pad} is in P then L is in $\mathbf{EXPTIME}$.

• To conclude whether an input x is in L, it just pads the input and decides whether it is in L_{pad}. This can be achieved by using the polynomial-time machine for L_{pad}.

Therefore, it can be said that if $\mathbf{P} = \mathbf{NP}$ is assumed then $\mathbf{EXPTIME} = \mathbf{NEXPTIME}$. Thus, from the above explanation it may also be concluded that if $\mathbf{EXPTIME} \neq \mathbf{NEXPTIME}$ then $\mathbf{P} \neq \mathbf{NP}$.

Comment

Problem

Define pad as in Problem 9.13.

a. Prove that for every A and natural number k, A ? P iff pad(A, n^k) ? P.

b. Prove that P ≠ SPACE(n).

Step-by-step solution

Step 1 of 2

For any language A and function $f : N \to N$, the language $pad(A, f)$ is defined as:

$(A, f) = \{pad(s, f(m)) \mid \text{where } s \in A \text{ and m is the length of } s\}$.

(a) Suppose A be any language and $k \in N$. If $A \in P$, then $pad(A, n^k) \in P$ because it can be determined whether $w \in pad(A, n^k)$ by writing w as $s\#$ where s does not contain the # symbol then it is tested whether $|w| = |s|^k$; and finally it is tested that whether $s \in A$.the implementation of the first test in polynomial time is straight forward. The second test runs in time $\text{poly}(|s|)$ because $|s| \leq |w|$, the test runs in time $\text{poly}(|w|)$ and hence is in polynomial in time. If $pad(A, n^k) \in P$, then $A \in P$ because it can be determined whether $w \in A$ by padding w with # symbols until it has length $|w|^k$ and then test, whether the result is in $pad(A, n^k)$. The above explanation shows that $A \in P$ holds if and only if $pad(A, n^k) \in P$.

Comment

Step 2 of 2

(b) Assume that $P = SPACE(n)$. Suppose A be a language in $SPACE(n^2)$ but it will not exist in $SPACE(n)$ as the space hierarchy theorem says. The language $pad(A, n^2) \in SPACE(n)$ because it has enough space to run the $O(n^2)$ space algorithm for A , using that is linier in the padded language. Because of the assumption, $pad(A, n^2) \in P$, hence $A \in P$ as discussed above. Hence $A \in SPACE(n)$ by taking assumption once again. But that is a contradiction. Hence, It can be said that $P \neq SPACE(n)$.

Comment

Problem

Prove that $TQBF \notin \mathrm{SPACE}(n^{1/3})$.

Step-by-step solution

Step 1 of 1

The **space hierarchy theorem** says "if g is a space-constructible ($1^n \to 1^{g(n)}$ can be computed in space $O(g(n))$, $f(n) = o(g(n))$, then $\mathbf{SPACE}(f(n)) \subsetneq \mathbf{SPACE}(g(n))$". So, from the space hierarchy theorem it can be said that there exists a language L, which is **solvable in linear space but it cannot be solved by sub-linear space**.

• Since, $\mathbf{TQBF}$ is space complete then L can be reduced to $\mathbf{TQBF}$ in log space. Therefore, if $\mathbf{TQBF} \in \mathbf{SPACE}\left(\mathbf{n}^{1/3}\right)$, then $L \in \mathbf{SPACE}\left(\left(\mathbf{n}^{\mathbf{c}}\right)^{1/3} + \log(\mathbf{n})\right)$.

• Now suppose, if $0.33 < 1/c$, **there exists a contradiction**.

Thus, from the above result it can be said that $TQBF \notin SPACE\left(n^{1/3}\right)$.

Comment

Problem

Read the definition of a 2DFA (two-headed finite automaton) given in Problem 5.26. Prove that P contains a language that is not recognizable by a 2DFA.

Step-by-step solution

Step 1 of 2

Suppose $L = \{p \mid \text{either } p = 0x \text{ for some } x \in B_{TM}, or\ p = 1y \text{ for some } y \notin B_{TM}\}$.

• It can be designed a Turing machine S and S may be defined as:

S : On input, write 0 followed by $\langle M, p\rangle$ in the tapes and halts. Then it is easy to check that:

$\langle M, p\rangle \in B_{TM} \Leftrightarrow output\ of\ Q \in L$

Thus, a mapping reduction of B_{TM} to L or $B_{TM} \leq_m L$ can be obtained .

• Now, a Turing machine(TM) R can be formed, which shows the functionality $B_{TM} \leq_m \overline{L}$. The Turing machine(TM) R can be defined as:

R : On input, write 1 followed by $\langle M, p\rangle$ in the tapes and halts. Then it is easy to check that:

$\langle M, p\rangle \in \overline{B}_{TM} \Leftrightarrow output\ of\ R \in L$

• Similarly,

$\langle M, p\rangle \in B_{TM} \Leftrightarrow output\ of\ R \in \overline{L}$

Thus, a mapping reduction of B_{TM} to $\overline{L}$ can be obtained .

Comment

Step 2 of 2

Since, $B_{TM} \leq_m L$ and $\overline{B}_{TM} \leq_m \overline{L}$. This show that $\overline{L}$ is non Turing recognizable because B_{TM} is **non Turing recognizable**. Similarly, since $B_{TM} \leq_m \overline{L}$ and $\overline{B}_{TM} \leq_m L$. So, this allows that L is non Turing recognizable. **Therefore, the above explanation shows that " P contains a language which is not recognizable by a 2DFA"**.

Comment

Problem

Let $E_{\text{REX}\uparrow} = \{\langle R \rangle |$

Step-by-step solution

Step 1 of 1

As it is known that **"the problem of any** $\mathbf{EXSPACE}$ **- complete cannot be in** $\mathbf{PSPACE}$ **or** In other words, it can be said that **"** $\mathbf{PSPACE} \subsetneq \mathbf{EXSPACE}$**"**.

• Now, consider the statement which is given below:

$E_{REX\uparrow} = \{\langle R \rangle \mid R \textbf{ is a regular expression with exponentiation and } L(R) = \varnothing\}$

• The above statement show that R **is a regular expression** and the language, which contains this regular expression, consists NULL value.

• By applying the concept of intractability, it can be said that $E_{REX\uparrow}$ **is intractable** because it can be demonstrated in such a manner that it is complete for the class $\mathbf{EXSPACE}$.

• Now, from the above discussion, it may be concluded that "any $\mathbf{EXSPACE}$ **- complete problem cannot be in** $\mathbf{PSPACE}$ **and it is much less in** P".

• Otherwise, $\mathbf{PSPACE}$ will be same as $\mathbf{EXSPACE}$**,** which is contradicting the corollary that is discussed above.

As it is explained above that" any $\mathbf{EXSPACE}$ **- complete problem is much less in** P". Therefore, it can be said that $\mathbf{E}_{\mathbf{REX}\uparrow} \in \mathbf{P}$.

Comment

Problem

Define the ***unique-sat*** problem to be

USAT = { $\langle \phi \rangle \mid \phi$ is a Boolean formula that has a single satisfying assignment}.

Show that *USAT* ? P^{SAT}.

Step-by-step solution

Step 1 of 1

Given:

$USAT = \{\langle \phi \rangle \mid \phi\}$, is basically a Boolean formula which is use for satisfying the single assignment.

Proof:

Here, it is to be proved that every Boolean formula which is satisfied has minimum one assignment which is bounded.

For proving this user need to satisfy the assignment of the *USAT* formula which is defined syntactically which the help of propositional logic.

$USAT \in \mathrm{P}^{SAT}$

Construction:

Here, user is proving that *USAT* should have only single truth value assignment which can be proved with the help propositional logic.

• User need to define a class of bonded truth value.

• It is to be proved that formula ϕ should have minimum one assignment which is bounded. For proving this there is one assignment which is chosen that is I_ϕ which should be syntactically defined.

• Therefore, characterization of the assignment which is uniquely satisfied is $\phi \in \mathrm{USAT}$ and it is proved by the definition which is I_ϕ.

• I_ϕ, constraint is use for satisfying all the assignment of ϕ this is because it contains information for the truth value assignment which is use for satisfying the value of ϕ.

• The assignment I_ϕ is syntactically defined with the help of propositional logic for each and every value of the variable *p*, substitution of the value p is the variable free formula.

• Here, I_ϕ is satisfying all the truth value assignment ϕ.

• I_ϕ, contains information for the truth value assignment which is use for satisfying the value of ϕ and even it help in determining the other truth value which is use for falsifying the value of ϕ.

It is proved that unique assignments in USAT are basically bounded and can be syntactically defined with the help of propositional logic.

Hence, it is proved that $USAT \in \mathrm{P}^{SAT}$

Comment

Problem

Prove that an oracle C exists for which $NP^C \neq coNP^C$

Step-by-step solution

Step 1 of 2

It is known that an oracle A exists such that $P^A \neq NP^A$ and $L_A \notin P^A$. For a given oracle A, L_A may be defined as:

$$L_A = \{w : |w| = |x| \text{ for some } x \in A\}$$

Then, it is clear that L_A is in NP^A.So, finally $L_A \notin coNP^A$ will have to proof (that is, same as $\overline{L}_A \notin NP^A$).

• For this purpose, first of all A will be constructed and $M_1, M_2, \cdots$ is now a list of all nondeterministic polytime oracle TMs, instead of all deterministic polytime oracle TMs.

Comment

Step 2 of 2

For Stage i **,** choose n and run M_i in input 1^n. It is respond NO to the query if M_i queries a string y whose status has not yet been determined.

• If there does not exist any computation and also, if M_i does not accept 1^n under these conditions then M_i is forced to make a mistake. It is performed because A can be maintained permanently that contains no string of length n. Then, $1^n \in \overline{L}_A$, but M_i does not accept 1^n.

If, on the other hand, there is an accepting computation for M_i with input 1^n, then it may be noted that this accepting computation can only query polynomially many strings y.

• Hence, there is a string x of length n which this computation does not query. We specify that this string x is in A and no other string of length n is in A.

• M_i Still has the same accepting computation on input $1^n \notin \overline{L}_A$. So makes a mistake in this case also. So, it may be concluded from the above explanation is that, an oracle C exists for which $NP^c \neq coNP^c$.

Comment

Problem

A k-***query oracle Turing machine*** is an oracle Turing machine that is permitted to make at most k queries on each input. A k-query oracle Turing machine M with an oracle for A is written $M^{A,k}$. Define $P^{A,k}$ to be the collection of languages that are decidable by polynomial time k-query oracle Turing machines with an oracle for A.

a. Show that $\mathrm{NP} \cup \mathrm{coNP} \subseteq \mathrm{P}^{SAT,1}$.

b. Assume that $\mathrm{NP} \neq \mathrm{coNP}$. Show that $\mathrm{NP} \cup \mathrm{coNP} \subsetneq \mathrm{P}^{SAT,1}$.

Step-by-step solution

Step 1 of 2

In this user need to prove that union of the *NP* and *coNP* can be decided in polynomial time by using the oracle of the *SAT* problem.

This implies $\mathrm{NP} \cup \mathrm{coNP} \subseteq \mathrm{P}^{SAT,1}$

It is being known that the oracle problem *SAT* is basically the *NP*-complete problem. *NP* language is basically encoded in the polynomial time *SAT*.

As, it is known that $\mathrm{L} \subseteq \mathrm{NP}$ and $\mathrm{P}^{A,K} \in \mathrm{NP}$.

This implies *L* and $\mathrm{P}^{A,K} \subseteq \mathrm{NP}$ it can reduced in Poly-time by the oracle problem *SAT*.

So, $\mathrm{NP} \subseteq \mathrm{P}^{SAT}$

Similarly,

As, it is known that $\mathrm{L} \subseteq \mathrm{NP}$ and $\mathrm{P}^{A,K} \in \mathrm{NP}$.

This implies *L* and $\mathrm{P}^{A,K} \subseteq \mathrm{NP}$ it can reduced in Poly-time by the oracle problem *SAT*.

Even, $\neg \mathrm{L}\left(\in \mathrm{NP} \subseteq \mathrm{P}^{SAT}\right)$ can be reduced in Poly-time by the oracle problem *SAT*.

So, $\mathrm{coNP} \subseteq \mathrm{P}^{SAT}$

In case of *NP* "yes" answer is checked by the oracle Turing machine and in case of *coNP* "no" answer is check by the oracle Turing machine in polynomial time.

It is being known that for each and every *NP* complete problem there is *coNP* complete problem.

Suppose, *coNP* and *NP* are equal then in that case the polynomial collapsed to either *NP* or *coNP*. But as it is shown earlier that $\mathrm{NP} \subseteq \mathrm{P}^{SAT}$ and $\mathrm{coNP} \subseteq \mathrm{P}^{SAT}$.

Here union operation is done between *NP* and *coNP* as here, the output is in either "yes" or "no".

So, union is computed in the polynomial time.

Hence, $\mathrm{NP} \cup \mathrm{coNP} \subseteq \mathrm{P}^{SAT,1}$

Comment

Step 2 of 2

In this user need to prove that union of the *NP* and *coNP* can be decided in polynomial time by using the oracle of the *SAT* problem.

This implies $\mathrm{NP} \cup \mathrm{coNP} \not\subset \mathrm{P}^{SAT,1}$

It is being known that the oracle problem *SAT* is basically the *NP*-complete problem. *NP* language is basically encoded in the polynomial time *SAT*.

As, it is known that $\mathrm{L} \subseteq \mathrm{NP}$ and $\mathrm{P}^{A,K} \in \mathrm{NP}$.

This implies *L* and $\mathrm{P}^{A,K} \subseteq \mathrm{NP}$ it can reduced in Poly-time by the oracle problem *SAT*.

So, $\mathrm{NP} \subseteq \mathrm{P}^{SAT}$

Similarly,

As, it is known that $\mathrm{L} \subseteq \mathrm{NP}$ and $\mathrm{P}^{A,K} \in \mathrm{NP}$.

This implies *L* and $P^{A,K} \subseteq NP$ it can reduced in Poly-time by the oracle problem *SAT*.

Even, $\neg L\left(\in NP \subseteq P^{SAT}\right)$ can be reduced in Poly-time by the oracle problem *SAT*.

So, $coNP \subseteq P^{SAT}$

In case of *NP* "yes" answer is checked by the oracle Turing machine and in case of *coNP* "no" answer is check by the oracle Turing machine in polynomial time.

It is being known that for each and every *NP* complete problem there is *coNP* complete problem.

In this *coNP* and *NP* are not equal.

$NP \subseteq P^{SAT}$ and $coNP \subseteq P^{SAT}$.

So, union is not computed in the polynomial time.

Hence, $NP \cup coNP \not\subset P^{SAT,1}$

Comment

Problem

Suppose that A and B are two oracles. One of them is an oracle for *TQBF*, but you don't know which. Give an algorithm that has access to both A and B, and that is guaranteed to solve *TQBF* in polynomial time.

Step-by-step solution

Step 1 of 3

An Oracle Turing machine is a type of Turing machine having many different tapes and these tapes are called oracle tapes. The different states may be represented by q_{states}.

A TQBF is True Qualified Boolean formula. To Show TQBF in polynomial problem which has access to turing machines A and B can be solved by using Baker Gill Solovay Theorem. The problem can be broken up in two parts of algorithm:

Comment

Step 2 of 3

For Oracle A:

1. Consider TQBF has access to A then A=TQBF.

2. In the given problem it can be proved by showing $P^A = NP^B$.

3. As $TQBF$ is $PSPACE$ problem,

4. Hence $PSPACE \subseteq P^{TQBF} and\ PSPACE^{TQBF} \subseteq PSPACE$

5. Therefore, combining both result $P^A = NP^B$.

Comments (1)

Step 3 of 3

For Oracle B:

1. For oracle B it is required to show that $P^B \neq NP^B$.

2. For this define a language L_B as: $L_B = \{0^n \mid w \in \{0,1\}\}\ Here,\ B(w) = 1$

3. For any oracle B the defined language L_B is NP^B.

4. Now the machine's output will be 1 if $x = 0^n$ with $|w| = |x|$.

5. Here $TQBF \in NL$ this shows that $PSPACE \in NL$

6. By using hierarchy theorems and Baker Gill which states that $NL \subsetneq PSPACE$.

7. It shows that $P^B \neq NP^B$.

Using both two parts of algorithm result it is sure that $TQBF$ is in polynomial time for both oracle A and B.

Comments (1)

Problem

Recall that you may consider circuits that output strings over {0,1} by designating several output gates.

Let $add_n : \{0,1\}^{2n} \longrightarrow \{0,1\}^{n+1}$ take two n bit binary integers and produce the n+1 bit sum. Show that you can compute the add_n function with O(n) size circuits.

Step-by-step solution

Step 1 of 1

Consider the following *add* functions:

$add_n : \{0,1\}^{2n} \to \{0,1\}^{n+1}$

This function basically takes two *n* bit binary integer and thus it produces *n*+1 bit sum.

User need to compute the complexity of the *add* function.

Speed of the circuits depends upon the gate which has been chosen. If the complexity is more than at that time, speed should be increased.

So, there is the trade-off between the space and time complexity. It implies that if it is taking less time than speed is more or vice versa.

In case of binary adder, input is two bit binary integer. Suppose there are *n* digits then it is represented with the help of *n* line. Summation of the two *n* bit binary number is equal to *n*+1 bit binary number.

Sum generated by the two *n* bit binary integer is *n*+1 bit long which can be represented in the form of $O(n)$.

As $n + 1$, $n + 2 \ldots n + n$ are represented in form of order of *n*.

Hence, it is proved that complexity of the add_n function is $O(n)$.

Comment

Problem

Define the function $majority_n : \{0,1\}^n \longrightarrow \{0,1\}$ as

$$majority_n(x_1, \ldots, x_n) = \begin{cases} 0 & \sum x_i < n/2\,; \\ 1 & \sum x_i \geq n/2\,. \end{cases}$$

Thus, the majority$_n$ function returns the majority vote of the inputs. Show that majority$_n$ can be computed with:

a. O(n^2) size circuits.

b. O(n log n) size circuits. (Hint: Recursively divide the number of inputs in half and use the result of Problem 9.23.)

Step-by-step solution

Step 1 of 2

a) Let the number of inputs taken is n. A **bubble-sort** can be implemented as a circuit. It is used tocompare two bits and after comparing, reordering them if necessary is rather easy. The inputs can be called as x_1, x_2 and the outputs can be called as y_1, y_2. A sub-circuit can be written which accomplishes this as $y_1 = OR(x_1, x_2)$ and $y_2 = AND(x_1, x_2)$.**This circuit contains a size of two**.

• Now, the action of the bubble-sort algorithm can be mimicked on an array. It can be implemented one step at position to be the n input, n-output sub-circuit that passes through all the inputs taken as $< k \; and \geq k+1$ are unchanged.

• Now, the compare-swap sub-circuit, which is described above, on $< k \; and \geq k+1st$ input can be used to generate the kth and $k+1st$ output. This still has size two. Now, **a pass** can be implemented as the serial concatenation of steps for each of $k = 1, 2, \ldots, n-1$, which has a size $(n-1)*2$.

• A bubble-sort can be Proceed to implement as the serial concatenation of n passes. Therefore, this gives a size $\mathbf{n(n-1)*2 = O(n^2)}$.

Therefore, it can be said that $\mathbf{majority_n}$ can be computed in $O(n^2)$ size circuits.

Comment

Step 2 of 2

b) Let the **number of inputs taken** is n. A **Merge-sort** can be implemented as a circuit. It is used to compare two bits after recursively dividing the given inputs in to half. The total time taken here (to divide the inputs into equal halves iteratively)is $\log n$.

• Finally at the last, the inputs can be called as x_1, x_2 and the outputs can be called as y_1.

• Now,**the action of the merge-sort algorithm** can be mimicked on an array. It can be implemented one step at position to be the n input, $n/2$-**output sub-circuit**.

• Now, **a pass** can be implemented as the serial concatenation of steps, which has a size $n \log n$. Therefore, this gives a size $n * \log n = O(n \log n)$.

Therefore, it can be said that $\mathbf{majority_n}$ can be computed in $O(n \log n)$ size circuits.

Comment

Problem

Define the function majority_n as in Problem 9.24. Show that it may be computed with O(n) size circuits.

Problem 9.24

Define the function

$$majority_n : \{0,1\}^n \longrightarrow \{0,1\} \text{ as}$$

$$majority_n(x_1, \ldots, x_n) = \begin{cases} 0 & \sum x_i < n/2\,; \\ 1 & \sum x_i \geq n/2\,. \end{cases}$$

Thus, the majority_n function returns the majority vote of the inputs. Show that majority_n can be computed with:

a. $O(n^2)$ size circuits.

b. O(n log n) size circuits. (Hint: Recursively divide the number of inputs in half and use the result of Problem 9.23.)

Step-by-step solution

Step 1 of 1

Consider the number of inputs taken is n. A **bubble-sort** can be implemented as a circuit. It is used to compare two bits and after comparing, reordering them if necessary is rather easy. The inputs can be called as x_1, x_2 and the outputs can be called as y_1, y_2. A sub-circuit can be written which accomplishes this as $y_1 = OR(x_1, x_2)$ and $y_2 = AND(x_1, x_2)$. **This circuit contains a size of two**.

• Now, the action of the bubble-sort algorithm can be mimicked on an array. It can be implemented one step at position to be the n input, n-output sub-circuit that passes through all the inputs taken as $< k \text{ and } \geq k+1$ are unchanged.

• Now, the compare-swap sub-circuit, which is described above, on $< k \text{ and } \geq k+1st$ input can be used to generate the kth and $k+1st$ output. This still has size two. Now, **a pass** can be implemented as the serial concatenation of steps for each of $k = 1, 2, \ldots, n-1$, which has a size $(n-1)*2$.

• A bubble-sort can be Proceed to implement as the serial concatenation of one passes. Therefore, this gives a size $\mathbf{1(n-1)*2 = O(n)}$.

This $O(n)$ complexity can be achieved only when the **input taken is already in sorted order**. In other word it can be said that, if it shows **a best case behavior**. **Therefore, it can be said that** $\mathbf{majority_n}$ **can be computed in** $O(n)$ **size circuits.**

Comment