Complexity Theory
Space Complexity Prep

1. Some things / notes to remember i think

Solution:

- Since NP $\subseteq$ PSPACE, P = PSPACE implies P = NP. If any PSPACE-Complete problem is in P, then P = PSPACE. By the contrapositive, if P $\neq$ PSPACE, then a PSPACE-Complete problem is not in P.
- PATH $\in$ NL and PATH is NL-complete.
- If L is NP-complete and L' is coNP complete, then $L \cap L'$ is $NP \cap coNP$ complete.
  Solution *False.* SAT is NP-complete. NOSAT is coNP-complete. Their intersection is empty (so that language would trivially always return false, but we know that there are non-trivial languages in $NP \cap coNP$, primality, for example.).

2. Show that one of the following inequalities must hold: $L \neq P$ or $PSPACE \neq P$. Both are believed to be true, but no one knows how to prove either.

Solution: Suppose L = P and P = PSPACE, then L = PSPACE. However, we know by the space hierarchy theorem that $SPACE(f(n)) \subset SPACE(f(n)log(n))$. Thus, $L \neq PSPACE$. Therefore, one of the inequalities must hold.

3. A complexity class C is said to be closed under reductions if whenever L reduces to L', and $L' \in C$, then $L \in C$. Prove that P and PSPACE are closed under reductions.

Solution: Suppose $L \leq_p L'$ and $L' in P$. Then there is a function f that reduces L to L' in polynomial time. Then on an arbitrary input x, compute $f(x)$, and put results on an arbitrary tape. Use polytime procedure for L' to decide if $f(x) \in L'$, and output the result. This whole operation is polynomial in time since f is polynomial and the procedure for L' is polynomial. Then since $x \in L$ if and only if $f(x) \in L'$, we have shown that $L \in P$. Therefore, P is closed under reductions.

Now suppose $L \leq_p L'$ and $L' \in$ PSPACE. Then the same argument follows as above except that the procedure for L' is polynomial in space. Therefore, PSPACE is closed under reductions.

4. A deterministic machine is a special case of a non-deterministic machine. Copy the statement below to your answer booklet, and instead of the blanks, insert "=" where equality is known, and "$\subseteq$", which means equal or contained, otherwise, and write very brief explanations to support your choice in each case. In total, you should give six brief explanations.

Solution: $L \subseteq NL = coNL \subseteq P \subseteq NP \subseteq PSPACE = NPSPACE$

- $L \subseteq NL$: Deterministic log-space machines are a subset of non-deterministic log-space machines because non-det log-space machines can simulate det log-space machines.
- $NL = coNL$: by the Immerman Szelepcsenyi theorem which proves that non-deterministic log-space is closed under complement. Also, complement of PATH is in NL, hence NL = coNL.
- $NL = coNL \subseteq P$: PATH $\in P$, and PATH is NL complete, then any problem in NL can be solved by reducing it to PATH, which is in P. Therefore, NL $\subseteq$ P.
- Determinisitc polynomial time machines are a subset of non-deterministic polynomial time machines because non-det polynomial time machines can simulate det polynomial time machines. It is still an open question whether P = NP.
- 3SAT $\in$ PSPACE, and 3SAT is NP-complete, therefore, all lanaguages in NP can be reduced to 3SAT, which is in PSPACE. Therefore, all languages in NP can be solved in PSPACE.
- PSPACE = NPSPACE: This is because the space hierarchy theorem states that for any space constructible function $f(n)$, $SPACE(f(n)) \subset SPACE(f(n)log(n))$. Therefore, PSPACE = NPSPACE. So by Savitch's Theorem, PSPACE = NPSPACE.

5. Prove that the following language is PSPACE-Complete:

$$\text{SPACE TMSAT} = \{\langle M, w, 1\rangle : \text{DTM} M accepts w in space n\}$$

Solution:

1. **SPACE TMSAT $\in$ PSPACE:** We can construct a DTM TM that simulates the given DTM M on input w and checks if it uses space n. TM basically works by simulating M on w while keeping track of the space used. If the space exceeds n, reject; if M halts and accepts without exceeding the space limit, accept; if it halts and rejects or runs out of space, reject.

   This simulation can be done within polynomial space because we only need to simulate the state transitions of M and keep track of its tape, which is bounded by n. Since n is part of the input (expressed in unary as 1^n), the simulation uses space proportional to n, which is allowed in the definition of the language. Therefore, a PSPACE machine can decide whether ¡M, w, 1¿ $\in$ SPACE TMSAT. Thus, SPACE TMSAT $\in$ PSPACE.

2. **SPACE TMSAT is PSPACE-Hard:**

   Let L be any PSPACE problem. By definition, there exists a Turing Machine M that decides L in polynomial space. We can construct a TM M' that on input x simulates M on x and uses space exacty equal to the polynomial bound of M'. Next we create a transformation such that for for any string x, we generate the tuple ¡M', x, $1^{p|x|}$¿ where $p|x|$ is the polynomial bound of M. This tuple is in SPACE TMSAT if and only if M accepts x since M' is constructed to simulate M exactly within the space bound of M which is $p|x|$.

   The reduction is polynomial in time since generating $1^{p|x|}$ and constructing M' can be done in polynomial time. Therefore, $L \leq_p$ SPACE TMSAT. Since L was an arbitrary PSPACE language, this shows that SPACE TMSAT is PSPACE-Hard.

Therefore, SPACE TMSAT is PSPACE-Complete.

6. Prove that UNCONN = $\{(G, s, t) \mid$ there is no directed path in G from s to t$\} \in coNL$.

Solution: *NAIVE METHOD:* UNCONN is the complement of CONN which exists in NL and we know that NL = coNL, therefore, UNCONN ∈ coNL.

Better Method:

Let G be a directed graph and s, t vertices in G. Define: C_i = vertices reachable from s in at most i steps

We know that there is a witness $v \in C_i$ which can be verified in log-space: a list of vertices $v_0, ..., v_j$ for $j \leq i$ such that $v_0 = s$, $v_j = v$, and $(v_k, v_{k+1}) \in E$ for $0 \leq k < j$, that is, each two nodes in the path are adjacent. We can show the witness is in log-space for basically two things:

1. A witness that $v \notin C_i$, given that we know $|C_i|$
2. A witness for $|C_i|$, given that we know $|C_{i-1}|$.

A witness for $v \notin C_i$ given that we know $|C_i|$ is as so:
for each $u \in C_i$, in increasing order, we give the witness that $u \in C_i$, the verifier verifies that:

- The nodes u are in increasing order
- The witness for each u is correct
- The number of u encountered equals $|C_i|$
- The node v was not one of the nodes encountered

The verifier only needs log-space and reads the witness from left to right. The length of the witness is $O(|C_i| * n \log n) = O(n^2 \log n)$.

A witness for $|C_i|$ given that we know $|C_{i-1}|$ is as so:

- Whether $v \in C_i$ or not, if $v \in C_i$, there exists a node w such that $w, v \in G$, and a witness for $w \in C_{i-1}$.
- If $v \notin C_i$, then there is no such w.

The verifier can verify this in log-space by checking if $v \in C_i$ and if so, checking if there exists a node w such that $w, v \in G$ and a witness for $w \in C_{i-1}$.

We finally concatenate all the relevant witnessses. That is, first a witness that $|C_0| = 1$, then a witness for $|C_1|$ given that we know $|C_0|$, then a witness for $|C_2|$ given that we know $|C_1|$, and so on. At the end, we add a witness that $t \in C_n$, given that we know $|C_n|$. The total witness length is then in $O(n^4 \log n)$, and the verifier only needs log-space to verify the witness. Therefore, UNCONN ∈ coNL.

7. FORMULA-GAME = $\langle \phi \rangle$: Player E has a winning strategy in the formula game associated with ϕ is PSPACE-Complete.

Solution: It is the same as TQBF, which is PSPACE-Complete.

The formula $\phi = \exists x_1 \forall x_2 \exists x_3 ...[\psi]$ is TRUE when some setting for x_1 exists such that for any setting of x_2, a setting of x_3 exists such that, and so on... where ψ is TRUE under the settings of the variables. Similarly, Player E has a winning strategy in the game associated with ϕ when Player E can make some assignment to x_1 such that for any setting of x_2, Player E can make a setting of x_3 such that, and so on... where ψ is TRUE under the settings of the variables.

The same reasoning applies when the formula doesn't alternate between existential and quantifiers. If ϕ has the form $\forall x_1 x_2 x_3 \exists x_4 x_5 \forall x_6 ...[\psi]$, Player A woul make the first three moves in the

formual game to assign values to x_1, x_2, x_3, and then Player E would make the next two moves to assign values to x_4, x_5, and so on... where ψ is TRUE under the settings of the variables.

Hence, $\phi \in$ TQBF exactly when $\phi \in$ FORMULA-GAME. Therefore, FORMULA-GAME is PSPACE-Complete.

8. PATH = $\langle G, s, t \rangle$: there is a path from s to t in G is NL-Complete.

Solution:

1. **PATH is in NL**. We can solve PATH in log-space by using a non-deterministic machine that guesses a path from s to t and verifies that the path is correct. The verifier only needs to keep track of the current node and the next node in the path, which can be done in log-space. Therefore, PATH $\in$ NL.

2. Figure below:

PROOF We show how to give a log space reduction from any language A in NL to *PATH*. Let's say that NTM M decides A in $O(\log n)$ space. Given an input w, we construct $\langle G, s, t \rangle$ in log space, where G is a directed graph that contains a path from s to t if and only if M accepts w.

The nodes of G are the configurations of M on w. For configurations c_1 and c_2 of M on w, the pair (c_1, c_2) is an edge of G if c_2 is one of the possible next configurations of M starting from c_1. More precisely, if M's transition function indicates that c_1's state together with the tape symbols under its input and work tape heads can yield the next state and head actions to make c_1 into c_2, then (c_1, c_2) is an edge of G. Node s is the start configuration of M on w. Machine M is modified to have a unique accepting configuration, and we designate this configuration to be node t.

This mapping reduces A to *PATH* because whenever M accepts its input, some branch of its computation accepts, which corresponds to a path from the start configuration s to the accepting configuration t in G. Conversely, if some path exists from s to t in G, some computation branch accepts when M runs on input w, and M accepts w.

To show that the reduction operates in log space, we give a log space transducer that outputs $\langle G, s, t \rangle$ on input w. We describe G by listing its nodes and edges. Listing the nodes is easy because each node is a configuration of M on w and can be represented in $c \log n$ space for some constant c. The transducer sequentially goes through all possible strings of length $c \log n$, tests whether each

is a legal configuration of M on w and outputs those that pass the test. The transducer lists the edges similarly. Log space is sufficient for verifying that a configuration c_1 of M on w can yield configuration c_2 because the transducer only needs to examine the actual tape contents under the head locations given in c_1 to determine that M's transition function would give c_2 as a result. The transducer tries all pairs (c_1, c_2) in turn to find which qualify as edges of G. Those that do not are added to the output size.

9. Prove that every language L that is not in the empty set or $\{0,1\}^*$ is complete for NL under polynomial-time Karp reductions.

Solution:

1. $L \in NL$

   Since $L \neq \emptyset$ and $L \neq \{0,1\}^*$, there must exists some nondeterministic log-space turing Machine M that decides L. Hence, $L \in NL$.

2. NL Hardness:

Consider an arbitrary language $A \in NL$. We need to reduce A to L in polynomial time. By definition, there exists a Turing Machine M that decides A. The behavior of M on an input x can be modelled as a directed graph $G_{M,x}$ where each node represents a configuration of M on x and each edge represents a valid transition between configurations. Then the question that $x \in A$ reduces to the existence of a path from the start configuration to an accepting configuration in $G_{M,x}$. We already know that the PATH problem is in NL, and that it is NL Complete. Sine L is a language in NL, there exists a non-det log-space Turing machine M for L. Then by definition of NL, any computation in NL can be reduced to L, including deciding A. Further, the reduction transforms instances of A into instances $f(x)$ of L. Since $A \in NL$, M's behaviour modelled as a graph $G_{M,x}$ can be computed in log-space. Therefore, the reduction f is computable in log-space and is polynomial time. Thus, $A \leq_p L$.

Therefore, L is NL-Complete under polynomial-time Karp reductions.

10. Di-Graph (also note that ST-CONN is nothing but PATH)

Solution:

A directed graph $G = (V, E)$ is *strongly connected* if for every pair of verties (x, y), there is a directed path from x to y and a directed path from y to x. Define STRONGLY-CONN to be the language consisting of all graphs that are strongly connected. Either show that this problem is in **L** or show a complexity consequence that results if this problem is in **L**.

We will show that ST-CONN $\prec$ STRONGLY-CONN, by a log-space reduction. Since ST-CONN is **NL**-complete, this establishes that STRONGLY-CONN is also **NL**-complete, so if STRONGLY-CONN $\in$ Ł, then **NL** = Ł.

The reduction will take as input $G = (V, E), s, t$ and will produce a new graph $G' = (V', E')$ such that there is a path from s to t in G if and only if G' is strongly connected. The vertex set of the two graphs is the same: $V = V'$. All the edges in E are also in E'. In addition, for every vertex $v \in V$, the edges (v, s) and (t, v) are added to E'. This reduction can be done easily on log-space. The reduction copies all the vertices and edges in G from the input tape to the output tape (requiring no extra workspace). Then on the work tape, enumerate each vertex v and write the two new edges (v, s) and (t, v) to the output tape.

Now suppose that there is a path in G from s to t: $\langle s, v_1, v_2, \ldots, v_{l-1}, t\rangle$. Then for every two vertices x and y, the following path is a path from x to y in G': $\langle x, s, v_1, v_2, \ldots, v_{l-1}, t, y\rangle$. Therefore G' is strongly connected.

Suppose that G' is strongly connected. Then there must be a path from s to t in G'. It is possible to remove cycles from the path to get a simple path (that doesn't repeat any vertices) from s to t in G': $\langle s, v_1, v_2, \ldots, v_{l-1}, t\rangle$. Since the path is simple, none of the verices v_1 through v_{l-1} can be s or t. Therefore, the path does not contain any of the new edges which have the form (v, s) or (t, v). Since the path from s to t does not contain any of the new edges, there must also be a path from s to t in G.

11. Prove that PSPACE is closed under Union, intersection, concatenation, kleene star, and complement. Hint: You can conveniently use the fact that PSPACE = NPSPACE.

Solution:

1. **Union:**

   Let $L_1, L_2 \in PSPACE$. Construct a DTM that decides $L_1 \cup L_2$ as so:

   (a) Simulate the PSPACE Machines for L_1 and L_2 on the input x.

   (b) Accept if either of the machines accepts x.

   (c) Else reject.

   The simulation of the PSPACE machines can be done in polynomial space since $L_1, L_2 \in PSPACE$, and space is reusable, so we can use the same space for both the machines. Therefore, $L_1 \cup L_2 \in PSPACE$.

2. **Intersection:**

   Let $L_1, L_2 \in PSPACE$. Construct a DTM that decides $L_1 \cap L_2$ as so:

   (a) Simulate the PSPACE Machines for L_1 and L_2 sequentially on the input x.

(b) Accept if both of the machines accept x.

(c) Else reject.

The simulation of the PSPACE machines can be done in polynomial space since $L_1, L_2 \in PSPACE$, and space is reusable, so we can use the same space for both the machines. Therefore, $L_1 \cap L_2 \in PSPACE$.

3. **Concatenation:**

Let $L_1, L_2 \in PSPACE$. To show that $L_1\dot{L}_2 \in PSPACE$, we consider a string $w \in L_1\dot{L}_2$. And this string can be written as $w = uv$ where $u \in L_1$ and $v \in L_2$. We can construct a Non-DTM for $L_1\dot{L}_2$ as so:

(a) Nondeterministically split the input string w into two parts u and v (guess)

(b) Simulate the PSPACE machine for L_1 on u and the PSPACE machine for L_2 on v.

Since PSPACE = NPSPACE, the non-det computation can be done in polynomial space, hence $L_1\dot{L}_2 \in PSPACE$.

4. **Kleene Star:**

Let $L \in PSPACE$. To show that $L^* \in PSPACE$, we consider a string $winL^*$ which can be written as $w = w_1w_2...w_n$ where $w_i \in L$ for $1 \leq i \leq n$ and $n \geq 0$. We can construct a PSPACE machine M that works as so:

(a) Nondeterministically split w into $w_1, w_2, ..., w_n$ (guess)

(b) Simulate the PSPACE machine for L on each w_i

Since PSPACE = NPSPACE, the non-det computation can be done in polynomial space, hence $L^* \in PSPACE$.

5. **complement:**

To show that PSPACE is closed under complement, we use the fact that PSPACE = NPSPACE. By the Immerman Szelepcsenyi Thoerem, we know that NSPACE is closed under complementation, therefore, PSPACE is closed under complementation.

12. coNP $\subseteq$ PSPACE.

Solution:

Prove that co-NP $\subseteq$ PSPACE.

Solution:
Let $L \in$ co-NP. By definition $\overline{L}$ belongs to NP and so there is a polynomial time nondeterministic TM deciding $\overline{L}$. Clearly, polynomial time TM cannot scan more then polynomially many tape cells, which implies that $\overline{L}$ is decidable in nondeterministic polynomial space. By Savitch's theorem, $\overline{L}$ is decidable in polynomial space also on a deterministic TM. By swapping the accept and reject states on that deterministic machine, we get a polynomial space deterministic decider for L. This means by definition that $L \in$ PSPACE.

13. Another

Solution:

Assume a deterministic decider M with space complexity n^3. How many steps does the machine M at most perform on an input w of length n?

Solution:
If the number of scanned cells (and hence possible positions of the head) is bounded by n^3, then there are at most $qn^3g^{n^3}$ different configurations where q is the number of states, and g is the number of tape symbols. Because q and g are constants, we get that there are at most $2^{O(n^3)}$ different configurations. A deterministic decider cannot enter the same configuration twice (otherwise it would start looping), which means that $2^{O(n^3)}$ provides also an upper bound on the number of computation steps.

14. For a complexity class C, let co-C = L' — L $\in$ C and say that C is closed under complementation whenever C = co-C. Argue as to whether the following statements are true, false, or unknown.

    1. All deterministic time complexity classes are closed under complementation.
    2. All nondeterministic time complexity classes are closed under complementation.

Solution:

1. If a det machine decides a language L in some time, we can cosntruct a det machine to accept the complement of L in the same time by swapping the accept and reject states. Therefore, all deterministic time complexity classes are closed under complementation.
2. We do not know yet whether all nondeterministic time complexity classes are closed under complementation. For example, we do not know whether NP = co-NP.

15. Consider the following two decision problems: REACH - the problem of deciding, given a directed graph G and two vertices a and b in G, whether there is a path in G from a to b.
UREACH - the problem of deciding, given an undirected graph G and two vertices a and b in G, whether there is a path in G from a to b.
It is known that REACH is NL-Complete (under logarithmic-space reductions) and that UREACH is in the complexity class L.

    Based on the above information, for each of the following statements, state whether it is true, false, or unknown. In the case it is true or false, give full justification for your answer. In the case it is unknown, there is no need to justify.

    1. REACH is reducible to UNREACH in log space
    2. UNREACH is reducible to REACH in log space
    3. UNREACH is in P

Solution:

1. REACH is NL-Complete, while UNREACH is in L. (we know this for a fact prior). To reduce REACH to UREACH, we need to transform a directed graph into an undirected one such that the existence of a path in the directed graph implies the existence of a path in the undirected graph. Given that REACH is NL-complete and typically harder than problems in L, transforming REACH to UREACH with only logarithmic space is non-trivial.

2. Given an undirected graph G, we can simualte G as a directed graph G' by replacing each undirected edge $(u, v) \in G$ with two directed edges (u, v) and (v, u). A path exists between a and b in G if and only if a path exists between a and b in G'. The transformation / reduction requires a single traveral over the graph's edge list which can be stored in $O(\log V)$ space pointers and counters, reusing the same space. Thus, UNREACH is reducible to REACH in log space.

3. Since UNREACH $\in L$, and L is a subset of P, UNREACH is in P.

16. Let $S \subseteq N$ be a set of numbers. We write BIN-S for the set of binary strings (i.e., strings in 0, 1^*) x such that x is a binary representation of a number in S'. We also write UN-S for the set $\{a^k \mid k \in S\}$ where the notation a^k means the string consisting of k repititions of the symbol a.

    Prove that BIN-S $\in SPACE(n)$ if and only if UN-S $\in L$.

Solution:

1. **BIN-S** $\in SPACE(n) \implies$ **UN-S** $\in L$

   A logarithmic space algorithm can verify membership in UN-S by converting the unary input a^k into binary and checking if it belongs to BIN-S, which is in SPACE(n).

   We convert a unary input to binary in logspace. Given a a^k, count the length k in unary (i.e., determine k). Maintian a counter in binary and increment it until it reaches k. This can be done in logspace. With k in binary, simualte the SPACE(n) bounded algorithm for BIN-S. Since SPACE(n) allows a linear amount of space in terms of the binary representation of k, this can be completed efficiently. The overall space usage is dominated by the sapce needed to convert a^k to binary, which is logarithmic, and simulate the SPACE(n) algorithm for BIN-S. The algorithm runs in logspace and decides UN-S. Hence, BIN-S $\in SPACE(n) \implies$ UN-S $\in L$.

2. **UN-S** $\in L \implies$ **BIN-S** $\in SPACE(n)$

   A linear space algorithm can verify the membership in BIN-S by converting the binary input to unary and check if it belongs to UN-S, which is in L.

   Given a binary string x, calculate its integer value k in space proportional to the length of x. Generate a unary string a^k, for which the steps requires space proportional to k which is feasible in SPACE(n). With a^k in unary, simulate the L bounded algorithm for UN-S. Since $UN - S \in L$, this step uses logarithmic space in the unary representation. The overall space usage is dominated by the space needed to convert x to unary, which is linear, and simulate the L algorithm for UN-S. The algorithm runs in linear space and decides BIN-S. Hence, UN-S $\in L \implies$ BIN-S $\in SPACE(n)$.

Therefore, BIN-S $\in SPACE(n)$ if and only if UN-S $\in L$.

17. Show that in every finite two-person game with perfect information (by finite we mean that there is an a priori upper bound n on the number of moves after which the game is over andone of the two players is declaredthe victor—there are no draws) one of the two players has a winning strategy.

Solution: Assumptions: The game is finite, meaning there is an upper bound n on the number of moves.
The game has perfect information, so at every point, both players know the entire game state and the rules.
There are no draws; one of the players is always declared the winner.

Consider the end of the game, where the result is known for all possible configurations of the board or state after n moves. At this stage, the game outcome is fixed for every possible terminal state: Player 1 wins, or Player 2 wins.

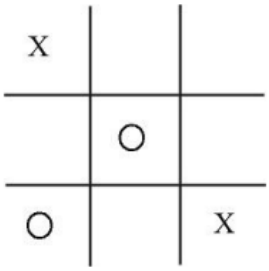
Induction:
Assume that at some stage k+1, the winner is determinable for all possible game states. For stage k:

- The current player (say Player 1) chooses a move from a finite set of legal moves.
- For each move, the game transitions to a new state at stage k+1, whose outcome is already determined.
- Player 1 will pick a move that guarantees their win if such a move exists; otherwise, the opponent (Player 2) is guaranteed to win.

At the final stage n, the outcome is already fixed, as there are no further moves. Then by induction, starting from the final stage, we can traverse backwards to determine the winner at each stage. Therefore, one of the two players has a winning strategy.

Since the game doesn't allow draws and it is finite, the induction guarantees that one of the players has a winning strategy.

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.

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

Solution:

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 "SAT = { <Φ>| is a satisfiable Boolean formula}."
- Also we know that "TQBF ={ < Φ>| is a true fully quantified Boolean formula}."
- We know that $SAT \in$ 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.

1. (Sipser 8.25) An undirected graph is bipartite if its nodes can be divided into two sets such that all edges go from a node in one set to a node in the other set. Show that a graph is bipartite iff it does not contain a cycle that contains an odd number of nodes. Let $BIPARTITE = \{\langle G \rangle \mid G \text{ is a bipartite graph}\}$. Show that $BIPARTITE \in \mathbf{NL}$.
[25 points]

   SOLUTION: Dividing the graph into two sets is the same as 2-coloring the graph such that all the edges are between vertices of different color. If the graph has an odd cycle, then the odd cycle in particular cannot be 2-colored and hence the graph cannot be bipartite. If the graph does not contain an odd cycle, we consider the DFS tree (or forest, if it is not connected) of the graph and color the alternate levels (say) red and blue. By construction, all the tree edges are between vertices of different color. Suppose an edge not in the tree connects two vertices of the same color. But these vertices must be connected by a path of even length in the tree and this extra edge will create an odd cycle, which is not possible. Hence, the above 2-coloring is a valid one which proves that the graph is bipartite.

   We show that $BIPARTITE \in \mathbf{coNL}$ or $\overline{BIPARTITE} \in \mathbf{NL}$, which suffices since $\mathbf{NL} = \mathbf{coNL}$. However, $\overline{BIPARTITE}$ is precisely the set of all graphs which contain an odd cycle. We guess a starting vertex v, guess an (odd) cycle length l and go for l steps from v, guessing the next vertex in the cycle at each step. If we come back to v (we can remember the starting vertex in logspace), we found a tour of odd length. Since any odd tour must contain an odd (simple) cycle, we accept and declare that the graph is non-bipartite.

(Sipser 8.23) Define $UCYCLE = \{\langle G \rangle \mid G$ is an undirected graph that contains a simple cycle$\}$. Show that $UCYCLE \in \mathbf{L}$. (Note: G may not be connected.)
[20 points]

Hint: We can try to search the tree by always traversing the edges incident on a vertex in lexicographic order i.e. if we come in through the ith edge, we go out through the $(i+1)$th edge or the first edge if the degree is i. How does this algorithm behave on a tree? How about a graph with a cycle?

SOLUTION: Note that the above process performs a DFS on a tree and we always come back to a vertex through the edge we went out on. However, if the graph contains a cycle, there must exist at least one vertex u and at least one starting edge (u, v) such that if we start the traversal through (u, v), we will come back to u through an edge different that (u, v). Hence, we enumerate all the vertices and all the edges incident on them, and start a traversal through each one of them. If we come back to the starting vertex through an edge different than the one we started on, we declare that the graph contains a cycle. Since we can enumerate all vertices and edges in logspace and also remember the starting vertex and edge using logarithmic space, this algorithm shows $UCYCLE \in \mathbf{L}$.

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.

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.

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

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.

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 $($.

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

Thus we proved that $A \in L$

20. Define UPATH to be the counterpart of PATH for undirected graphs. Show that BIPARTITE is log reducible to UPATH.

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 "** $\overline{BIPARTITE} \leq_L UPATH$".

21. BOTH NFA is NL-complete

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.

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** $NL - Complete$.

22. A NFA is NL-Complete

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 $NL-complete$).

23. E DFA is NL Complete

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 $\Sigma=\{1,\ldots..,d\}$ 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.

24. CYCLE 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*.

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, $CYCLE \in NL$.

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, $A_1, A_2, \ldots A_k$ 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.

25. EQ REX is in PSPACE or not?

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 $m_k \cap A_x = \phi \Leftrightarrow m_x \cap A_x \neq \phi, accept$.

• Pick any $a \in \Sigma$ and change m_x to $\bigcup_{q \in m_x} \delta_X(q,a)$ 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 $\overline{EQ_{REX}} \in PSPACE$