CS5110: Complexity Theory

Shreyas Havaldar
CS18BTECH11042

November 6, 2020

1 Assignment 2

1.1 A directed graph G is strongly-connected if for any ordered pair of vertices (u, v), there is a directed path from u to v in G. Show that the following language is NL-complete. $STRONG-CONN = \{\langle G\rangle |$ G is a directed graph, G is strongly connected$\}$.

For STRONG-CONN to be NL-complete, we need to show STRONG-CONN $\in$ NL and STRONG-CONN is NL-Hard.

I shall be using the fact that PATH = $\{\langle G, s, t\rangle |$ G is a directed graph and has an s-t directed path$\}$ is a NL-complete language as proved in class. Thus PATH $\in$ NL and PATH is NL-hard. And we know $NL = coNL$ and as $\overline{PATH} \in coNL$ thus $\overline{PATH} \in NL$. So basically $\overline{PATH} = \{\langle G, s, t\rangle |$ G is a directed graph and has NO s-t directed path$\}$

For showing STRONG-CONN $\in$ NL, I use the fact that $NL = coNL$, as proven in class by the Immerman-Szelepcsényi theorem. So showing $\overline{STRONG-CONN} \in NL$; thus there exists some pair of vertices (u,v) such that there is NO directed path between them, also suffices, as then STRONG-CONN $\in$ coNL and thus STRONG-CONN $\in$ NL.

On input graph $\langle G\rangle$, an algorithm for deciding $\overline{STRONG-CONN}$:

1. Non-deterministically select 2 distinct vertices (u,v) out of the vertex set of G.
2. Run $\overline{PATH}$ on $\{\langle G, u, v\rangle\}$ via a turing machine N.
3. If the turing machine N deciding $\overline{PATH}$ **accepts**, return TRUE as the graph G is not strongly connected, as there is NO directed path between at least one pair of vertices, namely (u,v).

   Else return FALSE.

This can be understood as:

- If for any branch of non-determinism we return TRUE, the algorithm returns TRUE as for some

vertex pair we have no directed path between them and thus they cannot be in the same strongly connected component and thus G is not strongly connected.

- If all non-det branches corresponding to all ordered pairs of vertices in the vertex set of the graph return FALSE, then there is a directed path between every vertex pair (u,v) and thus the graph would be strongly connected and we thus return FALSE.
- We repeat the operation on the same work tape cells, until until all the vertex pairs are exhausted so we only need log space to store the identity of the two vertices to check if there exists a path between them.

Thus as we know $\overline{PATH} \in$ NL, we can find if NO directed path exists between two vertices in $O(\log n)$ space where n is the input size. Thus we only use log space for this algorithm and proceed non-deterministically, thus $\overline{STRONG-CONN} \in NL \implies STRONG-CONN \in NL$

Note: I found proving $\overline{STRONG-CONN} \in NL$ more intuitive and thus proceeded in this direction.

Now for proving STRONG-CONN is NL-hard, we can show a log space reduction from PATH to STRONG-CONN as we can successively reduce a language L $\in NL$ to PATH followed by the reduction we find to reduce it to STRONG-CONN. Thus $\forall L, L \in NL, L \leq_L STRONG-CONN$

So for reducing an instance of PATH to an instance of STRONG-CONN we use the following algorithm:

- For a graph G = (V,E) and input $\langle G, s, t\rangle$, we copy all the existing edges and add additional edges (u,s) and (t,v) for every ordered vertex pair (u,v) to create G' = (V, E') onto the output tape.
- We would do the addition of new edges vertex-wise, we would proceed starting from vertex identified as the first vertex say u and output an edge from u to s and an edge from u to t on the output tape.
- We would repeat this operation of each of our n vertices successively, writing only the current vertex id on our working tape.
- Thus the space occupied on our working tape is $O(\log n)$, to denote the vertex id in binary notation and all our addition is to the output tape, where n is the number of vertices in the graph G.
- Thus our algorithm to reduce PATH to STRONG-CONN is indeed log space.

Now to prove the correctness of our reduction.

- For the forward direction, if there is indeed a directed path from s to t, meaning PATH accepts $\langle G, s, t\rangle$, the graph G' is indeed strongly connected as each ordered vertex pair (u,v) would have a directed path $(u \to s \to t \to v)$ and thus STRONG-CONN would accept G'.
- For the opposite direction, if G' is indeed strongly connected, there must exist a directed path s to t in G' as all ordered vertex pairs must have a directed path between them, thus also the vertex

pair (s,t), thus in our original graph G, an s-t path would exist as we did not add any edges to connect s to t, thus $\langle G, s, t\rangle$ would be accepted by PATH if G' is accepted by STRONG-CONN.

Thus we have shown STRONG-CONN $\in$ NL, and STRONG-CONN is NL-hard, thus STRONG-CONN is NL-complete.

Hence Proved!

1.2 To show that A_{NFA} is NL complete

$A_{NFA} = \{\langle N, w\rangle |$ N is an NFA that accepts string w$\}$.

So we need to show two things. $A_{NFA} \in NL$ and A_{NFA} is NL-hard. For the former we just need to that the space occupied by a non-deterministic turing machine for deciding A_{NFA} is $O(\log n)$. Let the turing machine deciding A_{NFA} be TM. TM has a work tape. We use this work tape to save only the current state of the NFA N running on input w. We can non-deterministically guess the next state on the input w and proceed saving only the state information in our work tape and we would need atmost $\log n$ bits to store a value from 1 to n using binary notation, where n is the number of states in the NFA N. Thus as the input must atleast list out all the states of the NFA N in the input, thus the input size say m, is $\Omega(n)$. Our TM will return true if for any branch of non-determinism, our state on the work tape belongs to set of accepting states in the NFA N when we reach the end of the input w. Therefore our work tape requires atmost $\log n$ space and therefore our non-deterministic TM occupies only $\log n$ space, which is necessarily $\log m$. Thus we have shown A_{NFA} can be decided by a non-deterministic TM in $\log m$ space. Thus, $A_{NFA} \in NL$.

Now for proving A_{NFA} is NL-hard. We know that PATH is NL-Complete where PATH $= \{\langle G, s, t\rangle |$ G is a directed graph and has an s-t directed path$\}$. Which implies, PATH is NL-Hard, therefore $\forall L \in NL, L \leq_L PATH$. If $PATH \leq_L A_{NFA}$, we can conclude that A_{NFA} is NL-Hard as we can reduce every language L $\in NL$ to PATH first and then can use another reduction to reduce PATH $\in$ NL to A_{NFA} sequentially in total O log space. Thus, we just need to show a reduction from PATH to A_{NFA}. To show $PATH \leq_L A_{NFA}$:

- Let the source vertex s in G be the start state S of the NFA N.
- Let the target vertex t in G be the accepting state T of the NFA N. Note: Our accepting state T has a self loop.
- For every edge (u,u') in G, there exists a transition from state U to state U' on reading 1 from the input tape in the NFA.

Let the input w be 1^{n-1}, that is the input tape to N would consist of n-1 ones, where n is the number of vertices in G.

Thus, this construction ensures that if and only if there is a path from s-t, we have a sequence of transitions in the NFA that causes an accepting configuration T at some point of reading the input tape, and due to self loop the NTM TM stays in T till the end of the input. This reduction can be

implemented in logarithmic space as we only need to represent the current state in the NTM TM which can be done in $O(\log n)$ space in binary notation where n is the number of vertices in G and equivalently the number of states in N. The reduction will not take more than $\log m$ space as the construction of the NFA N, would just require us move one vertex of the graph G at a time in each branch, and we would only store information corresponding to the state in N representing the vertex. Note: Our output tape would require polynomial space in input to save the configuration of the entire NFA N, but only the space taken by the work space is considered for the reduction and thus we are using $O(\log m)$ space. The reduction is also deterministic as we are proceeding in a sequential manner, in the order of vertex identities from 1 to n. There is no non-determinism involved at any point of the reduction.

An shortest s-t path cannot be of more than n-1 length. Thus if there is a shortest path s-t, it must be of length less than n, and the NTM TM would necessarily reach the state T in less than n transitions. Thus we defined w of length n-1. On reaching T, NTM TM stays there so at the end of the input, that branch of N stays in T, because we are considering all shortest s-t paths. thus if the length of an s-t path is smaller than n-1, we still accept.

Thus, $N = (Q, \Sigma, \Gamma, \iota, A, R, \delta)$, where

- Q is a finite set of states corresponding to vertices in G
- $\Sigma = \{1\}$, input alphabet
- Γ is the tape alphabet, where $\sqcup \in \Gamma$ and $\Sigma \subseteq \Gamma$
- $S \in Q$ is the start state
- $T \in Q$ is the accepting state
- $R \subseteq Q$ is the set of reject states ($R \cap A = \phi$) which here is all states other than T. ($Q \backslash T$)
- $\delta : (Q \times \Gamma) \longrightarrow P(Q \times \Gamma \times \{L, S, R\})$ is a relation on states and symbols called the transition relation. L is the movement to the left, S is no movement, and R is the movement to the right. (At any point in a computation, the machine may proceed according to several possibilities along different branches.)

Thus if any branch of our NTM TM is in state T at the end of the input, we can say that there exists a path from s-t in the graph G. If there is no s-t path, there are no possible sequence of transitions that on reading w, can take the NFA N from state S to state T. Thus we have successfully reduced PATH to A_{NFA} in log space. Thus, A_{NFA} is NL-Hard.

As $A_{NFA} \in NL$ and A_{NFA} is NL-Hard, A_{NFA} is NL-Complete.

Hence Proved!

1.3 To show that GM $\in$ PSPACE

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

To decide GM we basically need to find whether player "X" has a winning strategy, that is a set of moves such that irrespective of what move the opponent "O" plays, "X" can reach a final **winning state** of the game, here getting five of their markers consecutively in a row, column, or diagonal. We are given as input the board configuration and the identity of the player whose turn it is to play. I propose a recursive algorithm to find if a path to a final winning state exists for "X" from the current state in the tree of of all paths possible via all moves possible, a game theory tree.

Let M be the turing machine deciding GM. The algorithm is such:

1. If it is the turn of player "X", and they can reach a final winning state in one move, then return TRUE.

2. If it is the turn of player "O", and they can reach a final winning state in one move, then return FALSE.

3. If it is the turn of player "X", but they cannot reach a final winning state in one move, recursively run M on B' for every free position p on the board that can be played by "X" in one move such that, B' is the same config as B with some additional position p' being occupied by "X"'s marker and the turn of player being flipped from "X" to "O". If for any position(s) p', (one or more), this recursive call returns TRUE, then return TRUE, as then "X" would place their marker one of these position(s) p' which returns TRUE, this branch would lead to "X" on a winning state. If all positions p "X" could have placed a marker on return FALSE, return FALSE as then irrespective of which of the all possible positions "X" places their marker on, no position would return TRUE and thus in no branch of the game tree would "X" win.

4. If it is the turn of player "O", but they cannot reach a final winning state in one move, recursively run M on B' for every free position p on the board that can be played by "O" in one move such that, B' is the same config as B with some additional position p' being occupied by "O"'s marker and the turn of player being flipped from "O" to "X". If for all position p', this recursive call returns TRUE, then return TRUE, as then irrespective of which of the all possible positions "O" places their marker on, no position would return FALSE, thus "O" would not be able to force a losing state for "X" and thus in no branch of the game tree would "O" win and all branches would lead to "X" on a winning state. If on any position(s) p' (one or more) of all possible positions p, "O" could have placed a marker on return FALSE, return FALSE as then "O" would place their marker on one of these position(s) p' which returns FALSE, this branch would lead to "O" to be able to force "X" a losing state, and winning state for "O" along this branch of the game tree.

The total space occupied by this algorithm is to save the board configuration and maintain the identity if the player whose turn it is to play per move, this is the input size we will denote by $|Inp|$. Each level of recursion, that is each call of step 3 or step 4 in our algorithm uses at-most $|Inp|$, because as stated above, each level is just a modification to the board configuration and the flip in the identity of the player whose turn it is to play, so occupies the same space as the earlier input board config. There can be atmost $|Inp|$ calls made as the number of available total positions on the board are n^2 where the board is $n \times n$ and once the board is filled obviously no more markers can be placed. Thus

our recursion tree depth can go to atmost $O(|Inp|)$ depth and thus we would write atmost these many board configs on our work tape at the same time. When one branch rooted at some level is fully explored it can be simply rewritten by the board config next branch rooted at that level to be explored on the work tape. Thus as n^2 is $O(|Inp|)$, o the number of levels in our recursion stack, equal to total possible playable positions for the markers is $|Inp|$ where each level occupies $|Inp|$ space.

Thus the entire algorithm runs in space $O(|Inp|)^2$, where $f(x) = x^2$ is obviously a polynomial and thus GM $\in$ PSPACE.

Hence Proved!

1.4 To show that 2-SAT $\in$ NL

Using the same construction for the implication graph as in Assignment 1, solution to Problem 5.

2-SAT = $\{\langle\phi\rangle | \phi$ is a satisfiable 2-CNF formula$\}$

We know $a \vee b \equiv (\neg a \to b) \wedge (\neg b \to a)$.

Let there be an implication graph G with directed edges, of 2m nodes, where m are the number of variable literals in the 2-CNF formula. For each literal, we have two nodes in the graph corresponding to the literal and the negation of the literal. So for every clause $C_i = a \vee b$, we create 2 directed edges in the implication graph G, one from node $\neg a$ to b and the other from $\neg b$ to a. Thus for a 2-CNF formula with c clauses, we have 2c edges in our graph G.

Due to the propagation property of implication, $(a \to b) \wedge (b \to c) \equiv a \to c$, and therefore in G, having a path from a to c is sufficient to show that $a \to c$ and there need not necessarily be a direct edge between them.

As we know from the construction of our graph, for every edge a to b, we also have an edge from $\neg b$ to $\neg a$, and thus for every path from i to j, we also have an path from $\neg j$ to $\neg i$. Let the path be $i \to v_1 \to v_2 \ldots \to v_k \to j$. Then as edges $\neg v_1 \to \neg i, \neg v_2 \to \neg v_1 \ldots \neg j \to \neg v_k$ would also exist, a path from $\neg j$ to $\neg i$ would necessarily exist.

If there is an path from a to b, for $a \to b$ to be true: if a is true, b has to be true as well (along with each of the intermediate vertices); if a is false, then b can take any value, likewise for the intermediate vertices. We simply propagate the truth value via the sequential implication between two vertices with a directed edge between them such that they lie on this path.

Assuming the 2-CNF is satisfiable and thus each variable is assigned a value. If there exists a path from v to $\neg v$ and from $\neg v$ to v, we cannot choose an assignment for the literal v. If we assign TRUE to v, then $\neg v$ must also be TRUE due to the path from v to $\neg v$, which means v must be FALSE, which is a contradiction. If we assign FALSE to v, then $\neg v$ is TRUE, and due to the path from $\neg v$ to v, we must assign v, the value TRUE, which is a contradiction. Thus if we can say that 2-CNF is unsatisfiable iff there exists a path from v to $\neg v$ and from $\neg v$ to v, because we would find no assignment for the literal v that would satisfy the 2-CNF by our construction. Or the 2-CNF is satisfiable, iff no such pair of

paths exist.

For the case when 2-CNF is satisfiable, no such pair of paths must exist, or by contrapositive if such a pair of paths exists, 2-CNF is unsatisfiable. Thus if both v and $\neg v$ belong to the same strongly connected component, we know there exists a path from both v to $\neg v$ and from $\neg v$ to v. Thus we just need to find if there is a variable literal such that it and its negation both belong to the same strongly connected component in our construction of the graph G.

Now for the other possibility, that 2-CNF is unsatisfiable and thus for no v, v and $\neg v$ belong to the same strongly connected component. By contrapositive, if there is no variable v $\in$ V, such that there exists a path from both v to $\neg v$ and from $\neg v$ to v, we can say that the 2-CNF is satisfiable.

We can prove this direction by giving an algorithm for providing a satisfying assignment independent of the input configuration given to us by following these steps:

1. We topologically sort the strongly connected components. A vertex u, such that there is a path from u to w, lies before w in the sorting. Each vertex is addressed by the strongly connected component it belongs to henceforth.
2. We assign value TRUE to all variable vertices such that they lie after their negation variable vertices in the topological sort and FALSE to the negations of all these vertices.
3. We assign value FALSE to all variable vertices such that they lie before their negation variable vertices in the topological sort and TRUE to the negations of all these vertices.

This kind of assignment is bound to work as both a variable and its negation don't lie in the same SCC and thus occupy different positions in the topological sort. Suppose a vertex v is assigned value TRUE, then it must lie after $\neg v$ in the sorted order and thus no path from v to $\neg v$ would exist, so we would not be forced to assign TRUE to $\neg v$ causing a contradiction and we can safely assign $\neg v$ value FALSE. Likewise if we assign FALSE to a vertex u, then it must lie to the left of $\neg u$, and we would have no contradiction.

Also, there won't exist a w such that there would have a path to both w and $\neg w$ from the vertex v, because by the construction of our implication graph, it would cause a contradiction if we assign TRUE to v, thus making w and $\neg w$ both TRUE. If there indeed exists such a variable w, then due to the property we proved earlier, there would exist a path from $\neg w$ to $\neg v$ and w to $\neg v$. And thus there would be a path from v to $\neg v$, which would contradict our assumption made above that v could be assigned value TRUE. Thus no such vertex w can exist.

Note: The construction of this graph G can be achieved by simply not saving all the edges on our work tape. Our approach is to check whether two vertices have a path between them on the fly, and thus we only note the ID's of the 2 vertices, the source s and the target t on our work tape in binary notation, thus taking $O(\log m)$ space, where 2m is the number of vertices, thus $O(|Inp|)$, where $|Inp|$ is the input size is $\Omega(m)$.

For showing SAT $\in$ NL, I use the fact that $NL = coNL$, as proven in class by the Immerman-Szelepcsényi theorem. So showing $\overline{SAT} \in NL$; also suffices, as then SAT $\in$ coNL and thus SAT $\in$

NL.

I shall be using the fact that PATH = $\{\langle G, s, t\rangle|$ G is a directed graph and has an s-t directed path$\}$ is a NL-complete language as proved in class. Thus PATH $\in$ NL and PATH is NL-hard. And now we know PATH $\in$ NL, thus we can use a non-deterministic log space turing machine to find whether a directed PATH exists between two vertices s and t. The implication graph G constructed above is the graph we shall be providing as input to the turing machine deciding PATH implicitly in the form of the 2-CNF ϕ. So basically the input tape for our turing machine deciding $\overline{2-SAT}$ is also a part of the input tape for deciding $PATH$ repetitively along with the variable literal identity from the counter tape.

The algorithm to decide if an input is in $\overline{2-SAT}$ is as follows:

1. Non-deterministically pick a variable literal v until all variables are exhausted from the possible m, and thus the corresponding vertex v in the graph G from the vertex set of 2m vertices.
2. Run PATH on $\langle G, v, \neg v\rangle$ and $\langle G, \neg v, v\rangle$ via a turing machine N.
3. If the turing machine N deciding PATH **accepts** both the inputs, we return TRUE as the 2-CNF would be unsatisfiable irrespective of the value assigned to the variable corresponding to v

   else return FALSE.

This can be understood as:

- If for any branch of non-determinism we return TRUE, the algorithm returns TRUE as for some variable literal we would not be able to assign any value and thus the 2-CNF is unsatisfiable.
- If all the branches of non-determinism return FALSE, the algorithm returns FALSE, as there is no directed cycle between a variable and its negation thus we would be able to assign some value to each variable and thus $2-CNF$ would be satisfiable.
- We repeat computation on the same work tape cells, until until all the m literals are exhausted.

Note that this is indeed a non-det log space algorithm as the only space occupied is by the work tape for noting the vertex ID that is being checked now. We maintain it in binary and rewrite on the old ID, thus we never exceed $O \log |Inp|$ space. We use non-determinism to evaluate all possible vertices. PATH does not actually use a graph but rather interprets whether an edge between two vertices exists based on our construction from the $2-CNF$ itself.

Thus we have shown a non-det log space algo to decide $\overline{2-SAT}$. Thus $\overline{2-SAT} \in NL$, and thus $2-SAT \in coNL$.

Thus, $2-SAT \in NL$

Hence Proved!

1.5 Question 5

1.5.1

NO. The given fully quantified formula $\phi \notin TQBF$.

We have

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

Please look at the game/evaluation tree figure which gives a clear idea of why the given formula is not TQBF.

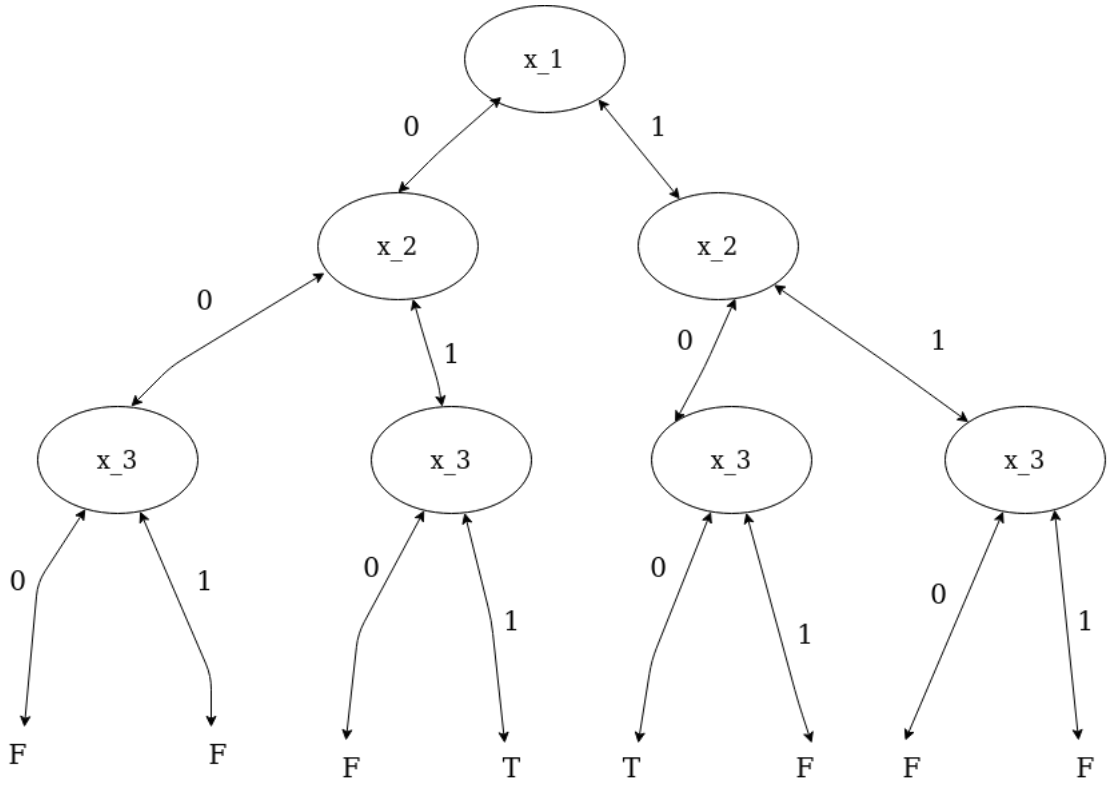

Figure 1: The game/eval tree. The last level of T/F are leaf nodes.

It can be thought of a two-player game where player 1 wants to make ϕ TRUE and player 2 wants to make ϕ FALSE by assigning alternately, the boolean variables favourable values for each of their goal. Note that player 1 starts. Now if you look at the tree carefully, you notice that if player 1 assigns value 0 to x_1, player 2 will assign value 0 to x_2 to force the output FALSE irrespective of what player 1 chooses for x_3. This is simply an extension of the quantifiers as player 1 needs to assign such values to x_1 such that irrespective of the value player 2 assigns to x_2, player 1 can choose a value for x_3 such that we reach a TRUE state. Thus for some first chosen value of x_1, the quantifier $\forall$ thus represents that irrespective of the value assigned to x_2, we can have a satisfying assignment for ϕ, by assigning some particular value to x_3 for this ϕ. So if every sub-tree originating from the child nodes of x_2 have

atleast one leaf node TRUE, ϕ would be satisfiable. But we can clearly see that is not the case as for x_1 is assigned 0, for $x_2 = 0$, we would not have a satisfying assignment for any value of x_3 and otherwise even if x_1 is assigned 1, for $x_2 = 1$, we would not have a satisfying assignment for any value of x_3.

Therefore, our formula $\phi \notin TQBF$.

1.5.2 To show that TQBF $\in$ NP $\implies$ NP = PSPACE.

We know from class that TQBF is PSPACE-complete. Thus $TQBF \in PSPACE$ and TQBF is PSPACE-hard. From the definition of hardness of a language with respect to a class, $\forall L \in PSPACE, L \leq_L TQBF$. Thus TQBF is necessarily harder than every language in PSAPCE. Now if TQBF $\in$ NP, every language in PSPACE, being less harder than TQBF, which itself is in NP, must also be in NP. $\forall L \in PSPACE, L \in NP$. Thus the class to which TQBF supposedly belongs, here NP, must be a superset of the class PSPACE. Thus $PSPACE \subseteq NP$.

We also know $NP \subseteq PSPACE$, as $\forall f(n) \geq \log n \implies NTIME(f(n)) \subseteq PSPACE(f(n))$.

Now as $PSPACE \subseteq NP$ and $NP \subseteq PSPACE$, $NP = PSPACE$ necessarily.

Thus if TQBF $\in$ NP $\implies$ NP = PSPACE.

Hence Proved!

1.5.3 We defined NL-completeness with respect to logspace reductions. Instead, if we had used polynomial time reductions to define NL-completeness, what would have been the class of NL-complete languages?

We know $NL \in P$, thus every language in the class NL can be solved via a poly time algorithm. Now if if we use poly time reductions to define NL-completeness, we would be able to decide the language $L \in NL$ during the reduction itself as it would definitely be decidable in poly time.

Thus our class of NL-complete languages would have been the set $NL \cap P$, which is equivalent to NL itself.

Note the difference as earlier with log space reductions the class of NL-complete languages is not NL, but rather $(NL \cap NL - hard)$.

Note: We are ignoring languages like the input-independent ones which accept all inputs or reject all inputs as we would not be able to reduce languages like PATH which depend on the input to them. So it is actually not completely NL. Basically it is $NL \backslash C$, where C is the set of all such input-independent languages.

1.5.4

We know Savitch's theorem states that for any function $f : \mathcal{N} \to \mathcal{R}^+, f \geq (\log(n))$:

$$\mathsf{NSPACE}\left(f\left(n\right)\right) \subseteq \mathsf{DSPACE}\left(\left(f\left(n\right)\right)^2\right)$$

As for $f(n) = \log n$, $\left((f(n))^2\right) = \left((\log n)^2\right)$, and Savitch's theorem implies:

$$\mathsf{NSPACE}(\log n) \subseteq \mathsf{DSPACE}\left((\log n)^2\right)$$

Savitch's theorem would not even hold for $f(n) = \sqrt{\log n}$

We cannot comment anything on $\mathsf{NSPACE}(\log n) \subseteq \mathsf{DSPACE}((\log n))$ using Savitch's theorem as both sides of the subset relation cannot be associated using it.

1.5.5

We know Savitch's theorem states that for any function $f : \mathcal{N} \to \mathcal{R}^+, f \geq (\log(n))$:

$$\mathsf{NSPACE}(f(n)) \subseteq \mathsf{DSPACE}\left((f(n))^2\right)$$

As for $f(n) = n^3$, $\left((f(n))^2\right) = \left(n^6\right)$, Savitch's theorem implies:

$$\mathsf{NSPACE}\left(n^3\right) \subseteq \mathsf{DSPACE}\left(n^6\right)$$

Which is what the question asked for, thus Savitch's theorem indeed implies $\mathsf{NSPACE}\left(n^3\right) \subseteq \mathsf{DSPACE}\left(n^6\right)$.