

Homework 10.5 (Fake)

Due: Never

Vinod Vaikuntanathan

Readings: Sipser, Sections 8.5, 8.6, and 10.2.

Problem 1: (Sipser 8.13) Show that TQBF restricted to formulas where the part following the quantifiers is in conjunctive normal form is still PSPACE-complete.

Solution 1: (borrowed from Spring 2001) To show that TQCNF is PSPACE-complete, we need to show that $\text{TQCNF} \in \text{PSPACE}$ and that $\text{TQBF} \leq_P \text{TQCNF}$.

TQCNF \in PSPACE because TQCNF is a special case of TQBF, so we can use the same decider for TQCNF as for TQBF.

To show that $\text{TQBF} \leq_P \text{TQCNF}$, we need to construct a polynomial time computable translation function, f , such that $\langle \phi \rangle \in \text{TQBF} \iff f(\langle \phi \rangle) \in \text{TQCNF}$. The following procedure F indicates, with one bug, how to compute f :

$F =$ “On input $\langle \phi \rangle$:

1. Let Q be the part of $\langle \phi \rangle$ containing all the quantifiers.
2. Let ϕ be the part of $\langle \phi \rangle$ containing no quantifiers.
3. Construct ϕ' that is the equivalent to ϕ in conjunctive normal form:
 - Using DeMorgan’s laws ensure that all NOTs in ϕ are only on individual variables and not on expressions.
 - Apply distributivity

$$(A \wedge B) \vee C = (A \vee C) \wedge (B \vee C)$$

where possible.

- Repeat until no more conversions of the above type are possible.

4. Output $Q[\phi']$.”

Since ϕ' is equivalent ϕ , we have that $\langle \phi \rangle \in \text{TQBF} \iff f(\langle \phi \rangle) \in \text{TQCNF}$.

Now we consider the time complexity of the reduction, and we discover the bug. The conversion from ϕ to ϕ' may blow up exponentially: the problem is in the applications of distributivity, because each application of the law converts one copy of C into two copies, potentially doubling the size of the formula at each application.

The fix is to construct ϕ' equivalent to ϕ such that ϕ' is a *quantified* CNF Boolean formula but $|\phi'|$ remains bounded by a polynomial in $|\phi|$. The trick for doing this is to show a slightly more general construction: construct from any Boolean formula, ϕ , a quantified Boolean formula, $E_{\phi,x}$, with one extra free variable, x , such that (i) $E_{\phi,x}$ is equivalent to the formula $x \equiv \phi$, (ii) $E_{\phi,x}$ would be in CNF if we ignored the quantifiers, and (iii) $|E_{\phi,x}|$ is bounded by a polynomial in $|\phi|$. Namely,

$$\begin{aligned} E_{y,x} &\stackrel{\text{def}}{=} (x \vee \neg y) \wedge (y \vee \neg x), \\ E_{\neg A,x} &\stackrel{\text{def}}{=} (\exists x_1)[E_{A,x_1} \wedge (\neg x \vee x_1) \wedge (x \vee \neg x_1)], \\ E_{A_1 \wedge A_2,x} &\stackrel{\text{def}}{=} (\exists x_1, x_2)[E_{A_1,x_1} \wedge E_{A_2,x_2} \wedge (x \vee \neg x_1 \vee \neg x_2) \wedge (\neg x \vee x_1) \wedge (\neg x \vee x_2)], \\ E_{A_1 \vee A_2,x} &\stackrel{\text{def}}{=} (\exists x_1, x_2)[E_{A_1,x_1} \wedge E_{A_2,x_2} \wedge (x \vee \neg x_1) \wedge (x \vee \neg x_2) \wedge (\neg x \vee x_1 \vee x_2)]. \end{aligned}$$

We leave it to the reader to verify that this construction satisfies (i–iii) above.

Now note that we can always rename quantified variables so no two quantifiers bind identical variables. Also notice that none of the quantifiers in $E_{\phi,x}$ are within a negated subformula. So if we pull all quantifiers to the beginning of $E_{\phi,x}$, we obtain an equivalent formula $E'_{\phi,x}$ of the same size as $E_{\phi,x}$ that is a quantified (in fact existentially quantified) CNF formula.

Now define $\phi' \stackrel{\text{def}}{=} (\exists x)[x \wedge E'_{\phi,x}]$. With this revised construction of ϕ' , the QCNF $f(\langle\phi\rangle)$ is of size polynomial in $|\langle\phi\rangle|$ and can also be computed in time polynomial in $|\langle\phi\rangle|$. Hence this revised mapping f is the required polytime reduction from TQBF to TQCNF.

Problem 2: (Sipser 8.27) Recall that a directed graph is *strongly connected* if every two nodes are connected by a directed path in each direction. Let

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

Show that STRONGLY-CONNECTED is NL-complete.

Solution 2: First, we will show that SC is in NL. We give a nondeterministic log space TM M that decides SC.

$M =$ “On input $\langle G \rangle$,

1. Count the nodes of G ; let that total be n .
2. For $s=1$ to n ,
 For $t=1$ to n ,
 If $s \neq t$, simulate the NL PATH decider on $\langle G, s, t \rangle$.
3. If all simulations accepted, accept; otherwise, reject.”

This algorithm works by checking that there is a directed path between every two nodes in each graph – the definition of a strongly connected graph. M runs in nondeterministic log space: two log n counters are needed for s and t , the PATH algorithm takes nondeterministic log space – which we can reuse for each simulation. Thus, we conclude that SC is in NL.

Next, we will show that $\text{PATH} \leq_L \text{SC}$.

$F =$ “On input $\langle G, s, t \rangle$,

1. Count the nodes of G ; let that total be n .
2. Copy G to the output tape.
3. For $u=1$ to n ,
 If $u \neq s$, add directed edge (u, s) to output tape.
 If $u \neq t$, add directed edge (t, u) to output tape.”

This algorithm only needs log n space to store the counter for u .

If there is no path from s to t in G , then there is also no path from s to t in G' . To see this, observe that we only added incoming edges to s and outgoing edges to t . Thus, if $\langle G, s, t \rangle \notin \text{PATH}$, then $\langle G' \rangle \notin \text{SC}$.

If there is a path from s to t in G , then, for any nodes u, v in G' , there is a directed path that starts with (u, s) , takes the directed path from s to t , and ends with (t, v) . Thus, if $\langle G, s, t \rangle \in \text{PATH}$, then $\langle G' \rangle \in \text{SC}$.

Problem 3: This problem uses the ideas in the proof of Theorem 8.27.

Describe a nondeterministic log-space Turing machine M that decides the language

$$L = \{\langle G, s, m, k \rangle \mid G \text{ is a directed graph, } s \text{ is a node in } G, m, k \in \mathbb{N}, \text{ and exactly } m \text{ nodes of } G \text{ are reachable from } s \in G \text{ by paths consisting of at most } k \text{ edges}\}.$$

That is, if exactly m nodes are reachable from $s \in G$ by paths of length at most k , then M must accept $\langle G, s, m, k \rangle$ on some computation path. On the other hand, if more or fewer than m nodes are reachable from $s \in G$ by paths of length at most k , then M must reject $\langle G, s, m, k \rangle$ on all computation paths.

Explain why your Turing machine M works correctly and why it works in log space.

Solution 3: This is meant to explore the details of the proof that \overline{PATH} is in NL on page 301 of (the old edition of) Sipser. The algorithm for M is essentially lines 1-11 on page 301 of Sipser; the correctness justification begins in the second to last paragraph on page 300. M 's job is to calculate the number of nodes reachable from s in k steps, that is c_k , and then test to see if $c_k = m$. Don't confuse the m here with the m nodes used on pages 300-301 – they are two different variables.

Problem 4: Define the language class PP as follows: A language $L \in PP$ if and only if there exists a probabilistic polynomial time Turing machine such that:

- If $w \in L$, then $\Pr[M \text{ accepts } w] \geq \frac{1}{2}$.
- If $w \notin L$, then $\Pr[M \text{ accepts } w] < \frac{1}{2}$.

Prove that:

1. $BPP \subseteq PP$.
2. $NP \subseteq PP$.
3. $PP \subseteq PSPACE$.

Hint for (2): Consider a nondeterministic TM for L , and replace rejections with probabilistic decisions.

Solution 4: (Borrowed from Spring '04)

1. $BPP \subseteq PP$. Straight from the definitions; $2/3$ is greater than $1/2$.
2. $NP \subseteq PP$. With probability $1/2^n$, reject; otherwise proceed. Take two random computation branches out of the 2^n possible choices. If either branch is accepting, accept; otherwise, accept with probability $1/2$ and reject with probability $1/2$. If all branches are rejecting (i.e., $w \notin L$), then we will correctly reject with probability $1/2^n(1) + (1 - 1/2^n)(1/2) = (2^n + 1)/(2^{n+1}) > 1/2$. If at least one branch is accepting (i.e., $w \in L$), then we will correctly accept with probability $1/2^n(0) + (1 - 1/2^n)\Pr[\text{alg accepts a random branch} \mid w \in L] \geq 1/2$. The $\Pr[\text{alg accepts a random branch} \mid w \in L] \geq 2/2^n + (1 - 1/2^n)(1/2) = (2^n + 5)/(2^{n+1})$.
3. $PP \subseteq PSPACE$. Simulate each of the $\leq 2^n$ possible computation branches. Each branch takes only polynomial time, thus only polynomial space. We can reuse the space for each individual branch, keeping track of which branch we are on with one $O(n)$ -bit counter. We also keep two counters, one for accepting and one for rejecting branches. After all branches have been explored, if more than half of the branches rejected, we reject; otherwise, we accept.

Problem 5: The class RP is the class of languages L for which there is a probabilistic Turing machine M that always terminates in polynomial time, and such that for all $w \notin L$, M *always* reject w , and for all $w \in L$, M accepts w with probability at least $\frac{2}{3}$. The class coRP is the class of languages whose complement is in RP.

So far, we have only discussed machines that *always* terminate in polynomial time, but that give a correct answer only with some probability. Here we consider machines that *always* give the right answer when they terminate, but that run in time that is only polynomial on average.

Show that for any language $L \in RP \cap \text{coRP}$ there is a probabilistic Turing machine M that runs in expected polynomial time (i.e., the expected number of steps until M terminates is bounded by a polynomial), and that when w terminates it accepts if and only if $w \in L$.

This class $RP \cap \text{coRP}$ is called ZPP for “zero probability polynomial”.

Solution 5:(Borrowed from Spring '04) Let TM T be an RP decider for a language L and let B be a coRP decider for L . Then we construct a TM M that decides L in ZPP as follows.
 M = “On input w ,

1. Run T on w 10 times. If T ever accepts, accept.
2. Run B on w 10 times. If B ever rejects, reject.
3. Otherwise, output “maybe, better run me again”.

Since T and B run in probabilistic polynomial time, then M will also halt in polynomial time. M is always correct; by the definition of RP and coRP, we know that if T accepts then $w \in L$ and if B rejects then $w \notin L$. The probability that one run of M either accepts or rejects (i.e., does not output “maybe”) is $\geq 2/3$.

$$\begin{aligned} Pr[M \text{ outputs maybe}] &= Pr[T \text{ does not accept} \mid w \in L]^{10} + Pr[B \text{ does not reject} \mid w \notin L]^{10} \\ &< (2/3)^{10} + (2/3)^{10} < 0.04 \end{aligned}$$

Thus, we’ll conclude that M decides L in *expected* polynomial time. We *expect* to run M on w only once, but it is easy to see that the probability that M would need to be run k -times on the same input decreases exponentially with k (note that it has nothing to do with the input itself). For the interested reader, we note that the argument that M is indeed expected poly time can be formally shown using Chernoff bounds, but the intuition is enough for our purposes.

Problem 6: (Fermat’s test) Sipser 10.15. Prove Fermat’s little theorem. That is, prove that

$$\text{If } p \text{ is prime, and } a \in \mathbb{Z}_p^+, \text{ then } a^{p-1} \equiv 1 \pmod{p}$$

(Hint: Consider the sequence $a, 2a, 3a, \dots, (p-1)a$. What must happen, and how?)

Solution 6: Following the hint, consider the sequence $a, 2a, \dots, (p-1)a$. The claim is that no two of these are equal modulo p . Suppose $ra \equiv sa \pmod{p}$ (W.l.o.g, assume that $s > r$). Then p divides $(s-r)a$. But $1 \leq s-r < p$ and $1 \leq a < p$. Thus p cannot divide either of these numbers, and therefore (since it is prime), cannot divide their product either. Therefore, we have a contradiction. This means all the numbers $a, 2a, \dots, (p-1)a$ are distinct modulo p . In other words, these numbers are just $1, 2, \dots, p-1$, albeit in a different order.

In particular, if we multiply these numbers using these two different ways of writing them, we get

$$a \cdot 2a \cdot \dots \cdot (p-1)a \equiv 1 \cdot 2 \cdot \dots \cdot (p-1) \pmod{p}.$$

This gives us that $a^{p-1} \equiv 1 \pmod{p}$.

Problem 7: (Branching program example) Show that the majority function can be computed by a branching program that has $O(n^2)$ nodes.

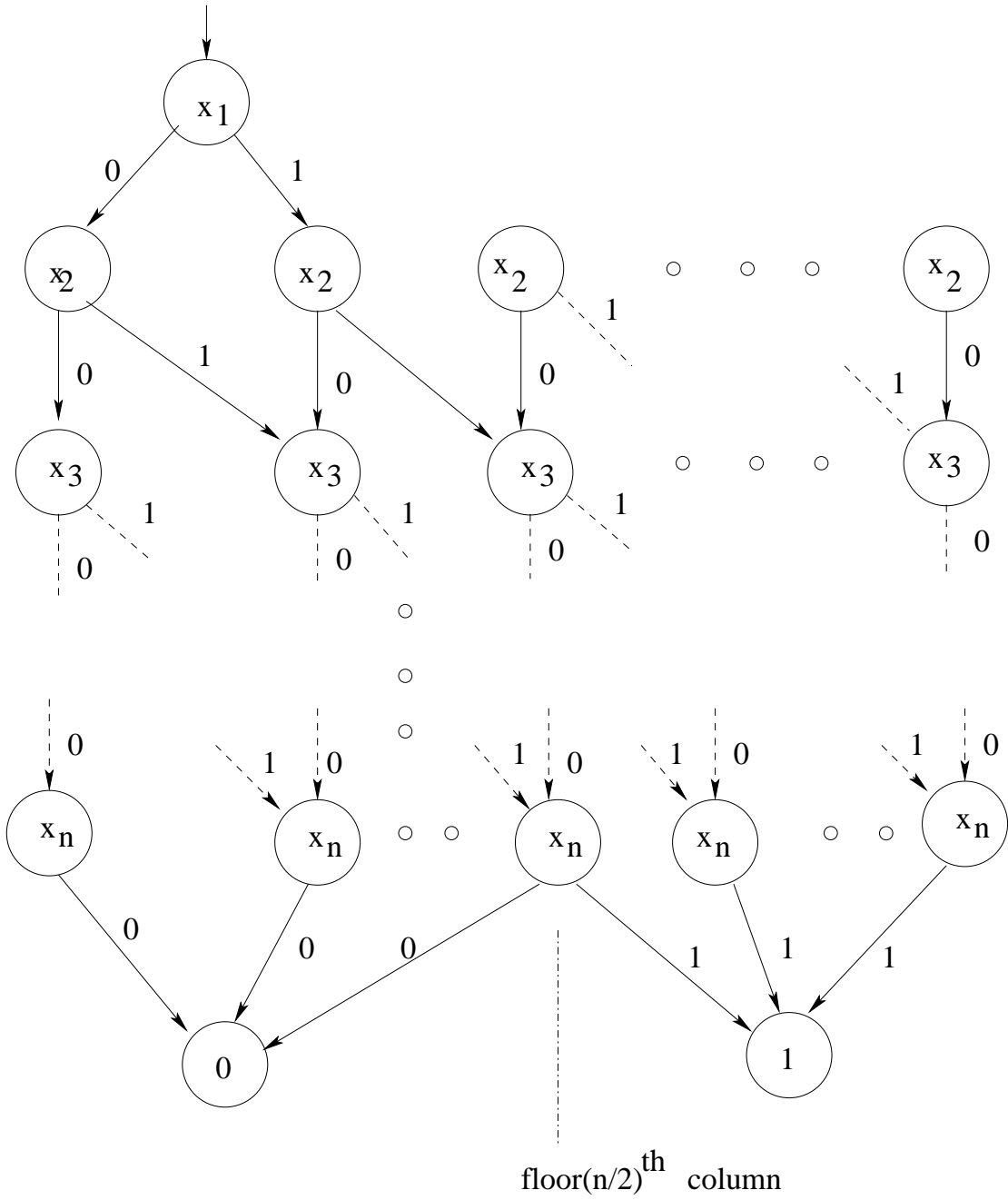
Solution 7: The branching program is as in the figure below. It has n rows of nodes, each row containing n nodes. Row i contains the nodes labeled with the variable x_i . When we are in row i and column j (in a node labeled x_i), if we see that $x_i = 1$, go to row $i+1$ and column $j+1$. On the other hand, if $x_i = 0$, go to row $i+1$ but stay in column j .

The intuition is that the vertical column that the computation of the branching program is in, after reading some part of the input, indicates a count of the number of 1s seen so far. At the end, we go to the 1 state, if the number of 1s read in the input is larger than the number of 0s, and go to the 0 state otherwise.

Problem 8: (Branching program equivalence test)

1. Give a read-once branching program B_1 that computes the function of three Boolean variables, x_1, x_2 , and x_3 , that has value 1 if and only if exactly one or exactly three of the variables have value 1.

Solution 8: See Figure 1 below.



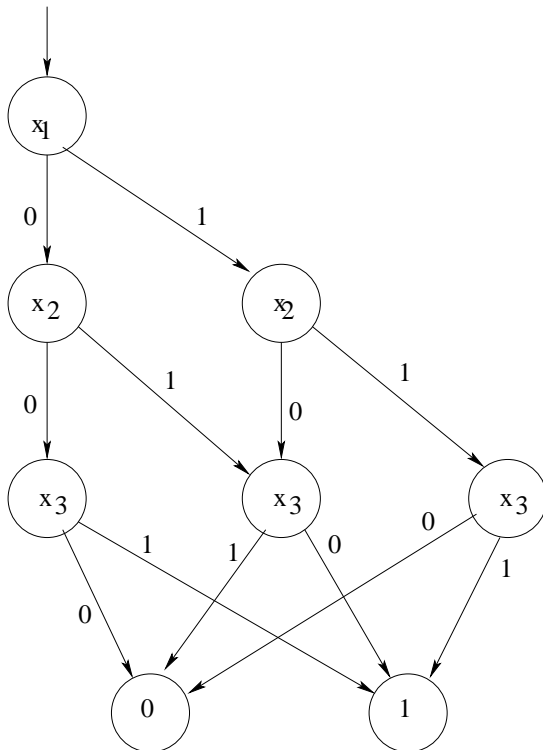


Figure 1: The first read-once BP for the function in Problem 8.

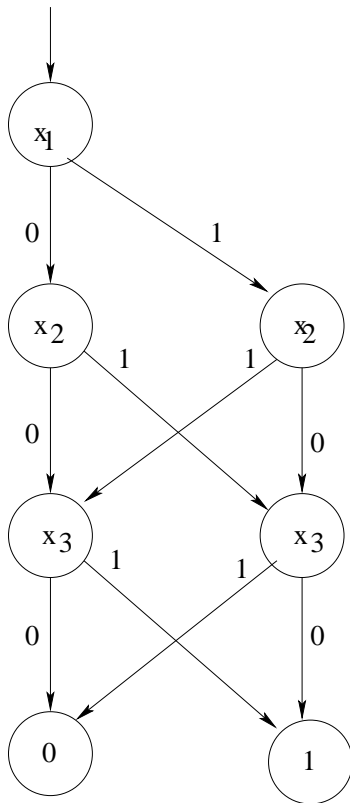


Figure 2: A second read-once BP for the function in Problem 8.

2. Give a different read-once branching program B_2 that computes the same function as in part (a).

Solution 8: See Figure 2 below.

3. Compute the polynomials p_1 and p_2 associated with the output 1 box for programs B_1 and B_2 , respectively, using the rules given in Sipser's book, p. 378.

Solution 8: $p_1 = (1 - x_1)(1 - x_2)x_3 + \{ (1 - x_1)x_2 + x_1(1 - x_2) \}(1 - x_3) + x_1x_2x_3$.
 $p_2 = x_3 \{ (1 - x_1)(1 - x_2) + x_1x_2 \} + (1 - x_3) \{ (1 - x_1)x_2 + (1 - x_2)x_1 \}$.

4. Choose arbitrary values from Z_7 for the three variables, and evaluate p_1 and p_2 to check that they indeed give the same result.

Solution 8: Let us check with $x_1 = 1, x_2 = 2, x_3 = 3$. Any choice will be good, since these two polynomials are identical.

$p_1(1, 2, 3) = 6 + 2 \equiv 1 \pmod{7}$ and $p_2(1, 2, 3) \equiv 1 \pmod{7}$ too.