Computational Complexity
Homework 2024-25

Simone Bianco, 1986936

Sapienza Università di Roma, Italy

April 13, 2025

Question 1. *Prove that if* $\mathsf{NP} \subseteq \mathsf{BPP}$*, then* $\mathsf{NP} = \mathsf{RP}$.

To prove this result you may keep in mind that:

- SAT *is self-reducible, this means that you can solve it by reducing to one or more smaller instances of the problem itself.*
- *It is possible, given an algorithm that decides* SAT *in polynomial time, to obtain an algorithm that on satisfiable formulas also produces a satisfying assignment.*
- *We saw how to reduce error in probabilistic computation. Essentially, given a polynomial time probabilistic algorithm with two-sided error for* SAT*, you should define another polynomial time probabilistic algorithm for* SAT*. But this time the algorithm's only errors must be false negatives. And of course the algorithm must have an appropriate probability of success. You have to prove the correctness of your algorithm.*

Solution 1. First, we observe that the inclusion $\mathsf{RP} \subseteq \mathsf{NP}$ holds without the assumption. Suppose now that that $\mathsf{NP} \subseteq \mathsf{BPP}$. We know that there is a PDTM M such that:

$$\langle\phi\rangle \in \text{SAT} \iff \Pr[M(\phi) = L(\phi)] \geq \frac{2}{3}$$

where $L(\phi) = 1$ if $\phi \in$ SAT, otherwise $L(\phi) = 0$. Through error reduction, we know that there is a PDTM M_p that reduces the error of deciding SAT to less than $\frac{1}{2^{|\phi|^p}}$ by repeatedly running M, where p is an arbitrary positive constant.

$$\langle\phi\rangle \in \text{SAT} \iff \Pr[M_p(\phi) = L(\phi)] \geq 1 - \frac{1}{2^{|\phi|^p}}$$

We define the following procedure A that constructs a satisfying assignment:

```
function $A(\phi)$
    Let $\alpha_0$ be an empty partial assignment
    for $i = 1, \ldots, n$ do
        if $M_p(\phi_{|\alpha_{i-1} \cup \{x_i = 1\}}) = 1$ then
            $\alpha_i = \alpha_{i-1} \cup \{x_i = 1\}$
        else
            $\alpha_i = \alpha_{i-1} \cup \{x_i = 0\}$
        end if
    end for
    Return $\alpha_n$
end function
```

Let $\alpha = A(\phi)$. Suppose that ϕ is unsatisfiable. Then, α will never satisfy ϕ. Hence, we have that:

$$\langle \phi \rangle \notin \mathrm{SAT} \implies \Pr[\phi(\alpha) = 1] = 0$$

Suppose now that ϕ is satisfiable. We notice that the assignment returned by the machine is correct not only when all the n calls of M_p give a correct output, but it may also be correct when some (or all) of the calls give the wrong answer. We'll show that restricting our interest to the probability of the case where all the calls accept is sufficient. Let $\mathrm{X_i}$ be the random Bernoulli variable such that $X_i = 1$ if and only if the i-th run of M_p inside A is correct. Due to the previous observation, we have that:

$$\Pr[\phi(\alpha) = 1] > \Pr\left[\bigcap_{i=1}^{n} X_i = 1\right] = 1 - \Pr\left[\bigcup_{i=1}^{n} X_i = 0\right]$$

Claim: For each $n \in \mathbb{N}$, it holds that $\Pr\left[\bigcup_{i=1}^{n} X_i = 0\right] \leq \sum_{i=1}^{n} \Pr\left[X_i = 0\right]$

Proof of the claim. When $n = 1$, the claim is trivially true. Assume that the claim holds for n. With $n + 1$ we have that:

$$\Pr\left[\bigcup_{i=1}^{n+1} X_i = 0\right] = \Pr[X_{n+1} = 0] + \Pr\left[\bigcup_{i=1}^{n} X_i = 0\right] - \Pr\left[X_{n+1} = 0 \cap \bigcap_{i=1}^{n} X_i\right]$$

$$\leq \Pr[X_{n+1} = 0] + \sum_{i=1}^{n} \Pr[X_i = 0]$$

$\square$

Through the claim we get that:

$$\Pr[\phi(\alpha) = 1] > 1 - \Pr\left[\bigcup_{i=1}^{n} X_i = 0\right] \geq 1 - \sum_{i=1}^{n} \Pr\left[X_i = 0\right] \geq 1 - \frac{n}{2^{|x|^p}}$$

which is always greater than $\frac{2}{3}$ for an appropriate choice of p. Finally, consider the TM M' that given ϕ in input first computes $A(\phi) = \alpha$ and then returns 1 if and only if $\phi(\alpha) = 1$. By our previous observations we have that:

$$\langle \phi \rangle \in \mathrm{SAT} \implies \Pr[M'(\phi) = 1] = \Pr[\phi(\alpha) = 1] \geq \frac{2}{3}$$

$$\langle \phi \rangle \notin \mathrm{SAT} \implies \Pr[M'(\phi) = 1] = \Pr[\phi(\alpha) = 1] = 0$$

Since M' runs in polynomial time, this concludes that SAT $\in$ RP. Moreover, since SAT is NP-Complete, this also concludes that NP $\subseteq$ RP.

Question 2. *This questions is broken in several tasks.*

1. *Show a polynomial size branching program for the inner product function*
$$\mathrm{IP}(x_1, \ldots, x_n, y_1, \ldots, y_n) = \sum_{i=1}^{n} x_i y_i \pmod 2$$
2. *Given a CNF formula $\phi(x_1, \ldots, x_n)$, describe a branching program that computes ϕ of size polynomial with respect to the size of ϕ. Computing ϕ means that the branching program must compute the value of the formula, given as assignment to its variables.*
3. *A read-once branching program is one where no variable is queried more than once on any path. Do part 1 again, but with a read-once branching program. Or show that your solution for part one is already read-once.*
4. *Can you do a read-once branching program for part 2? If not, why not? Discuss the difficulties, but a formal argument is not required here.*

Solution 2.

1. We'll construct a branching program that contains n levels. Each level evaluates the k-th sum of the inner product. In particular, each level contains two tracks. The computation will move between the two tracks based on the partial evaluation of the total inner product. To construct the levels, we use three gadgets: a start gadget, a middle gadget and an end gadget.

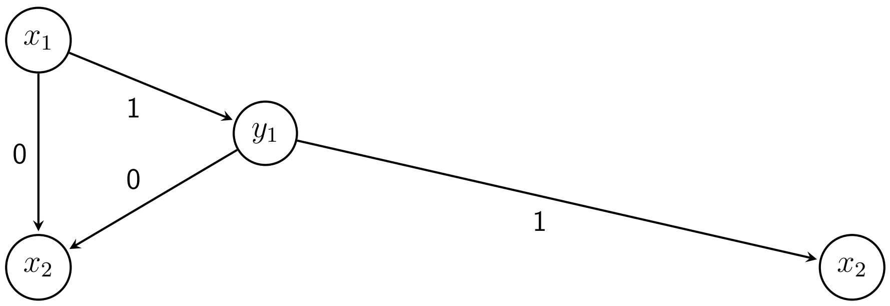

Figure 1: The start gadget

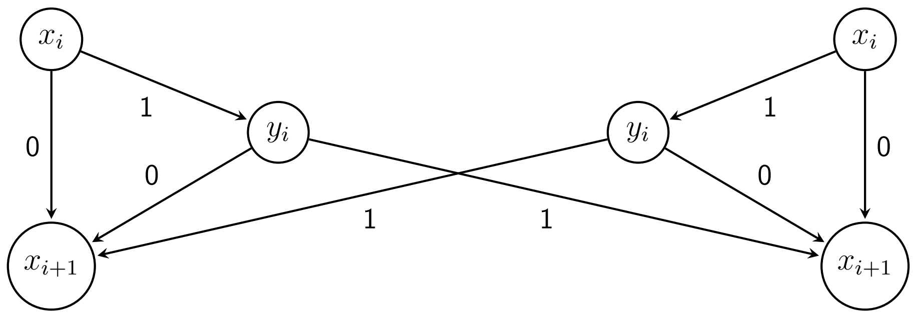

Figure 2: The middle gadget

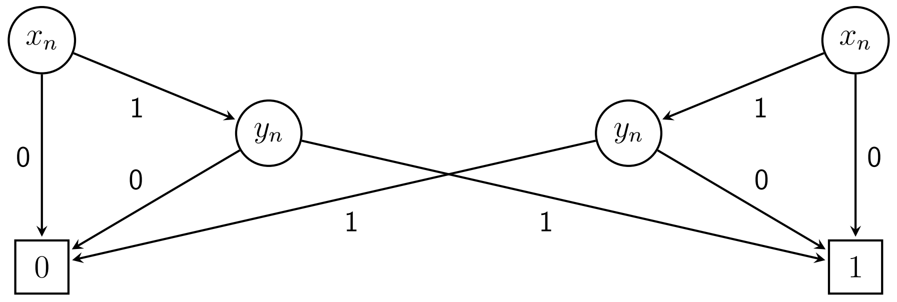

Figure 3: The end gadget

Given the input $x_1, \ldots, x_n, y_1, \ldots, y_n$, when $n > 1$, the branching program is constructed by a sequence of 1 start gadget, $n-2$ middle gadgets and 1 end gadget. When $n = 1$, the branching program contains a single start gadget, where the two nodes that query x_2 are replaced by the output nodes 0 and 1. For any $n \in \mathbb{N}$, the size of the constructed branching program is $O(n)$ (see Figure 4).

We claim the following invariant to prove the correctness of the program. Once the program reaches the $(n+1)$-th layer, the computation concludes.

Claim: For any $k \in [n+1]$, if in the k-th level the computation starts from the left track, we have that $\sum_{i=1}^{k-1} x_i y_i \equiv 0 \pmod 2$. Otherwise, if it starts from the right track, we have that $\sum_{i=1}^{k-1} x_i y_i \equiv 1 \pmod 2$.

Proof of the claim. If $k = 1$, we're in the first layer, where we always start from the left track and where the sum is equal to 0. Assume that the claim holds for the k-th layer. Assume that the $(k+1)$-th layer starts from the left track (we can use a similar argument for the other case). Then, this can happen only in two cases:

- The k-th layer started from the left track and $x_k = 0$ or $y_k = 0$. In this case, we have that:

$$\sum_{i=1}^{k} x_i y_i \equiv x_{k+1} y_{k+1} + \sum_{i=1}^{k-1} x_i y_i \equiv 0 + 0 \equiv 0 \pmod 2$$

- The k-th layer started from the right track and $x_k = 1$ or $y_k = 1$. In this case, we have that:

$$\sum_{i=1}^{k} x_i y_i \equiv x_{k+1} y_{k+1} + \sum_{i=1}^{k-1} x_i y_i \equiv 1 + 1 \equiv 0 \pmod 2$$

$\square$

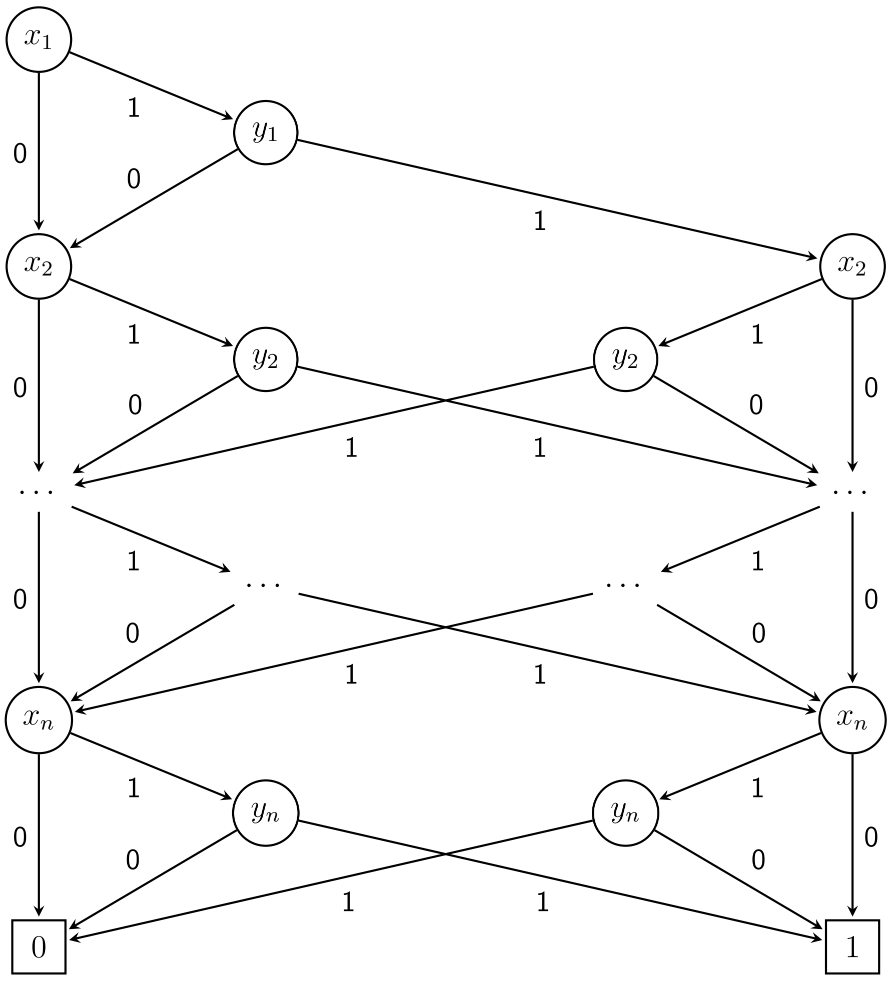

Figure 4: The branching program that computes $\mathrm{IP}(x_1, \ldots, x_n, y_1, \ldots, y_n)$

2. Consider the CNF $F = \bigwedge_{i=1}^{m} C_j$. For each $j \in [m]$, let k_j be the number of literals in the clause C_j, i.e. $C_j = \bigvee_{i=1}^{k_j} \ell_{i,j}$. We use the following clause gadget to construct the branching program.

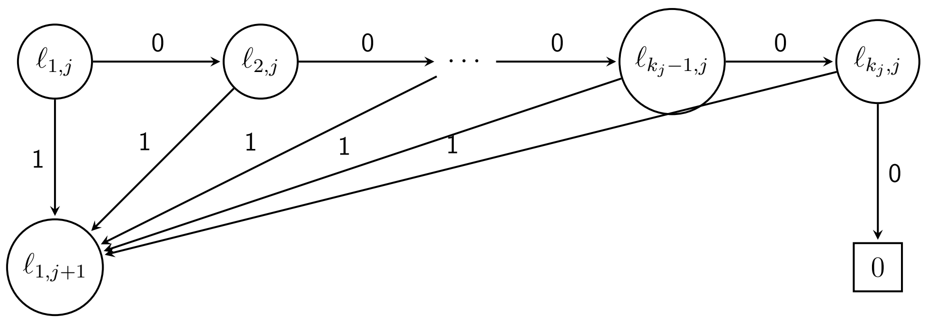

Figure 5: Clause gadget

<u>Note</u>: To make things easier, we're assuming that the branching program can also query the negation of variables. To convert it into standard form, if $\ell_{i,j} = \overline{x_t}$ for some variable x_t then we can simply invert the labels on the edges outgoing from the node labeled with $\ell_{i,j}$.

Each level of the branching program is the j-th clause gadget corresponding to the j-th clause of F. If $j = m$, the node $\ell_{1,m+1}$ is replaced by the output node 1. For any CNF ϕ, the branching program has exactly $2 + \sum_{i=1}^{m} k_j$ nodes, meaning that the size is linear with respect to the length of ϕ (see Figure 6). We claim the following property.

Claim: For any $j \in [m]$, the computation on x reaches the node labeled with $\ell_{1,j+1}$ if and only if C_j is satisfied by x.

Proof of the claim. Suppose that the computation on x reaches the node $\ell_{1,j+1}$. Then, this can only happen if one of the nodes $\ell_{1,j}, \ldots, \ell_{k_j,j}$ takes the edge labeled with 1. Let $\ell_{t',j}$ be such node. Then, this can happen if and only if $\ell_{t',j}$ is set to 1 in x.

Vice versa, suppose that the clause C_j is satisfied by x. Let $\ell_{t,j}$ be the first literal of C_j that is set to 1 in x. Once the branching program reaches the node $\ell_{t,j}$ it proceeds on the edge labeled with 1, reaching the node labeled with $\ell_{1,j+1}$. $\square$

Through the claimed property, we get that the computation on x reaches the output node 1 if and only if x satisfies each clause of ϕ, proving the correctness of the program.

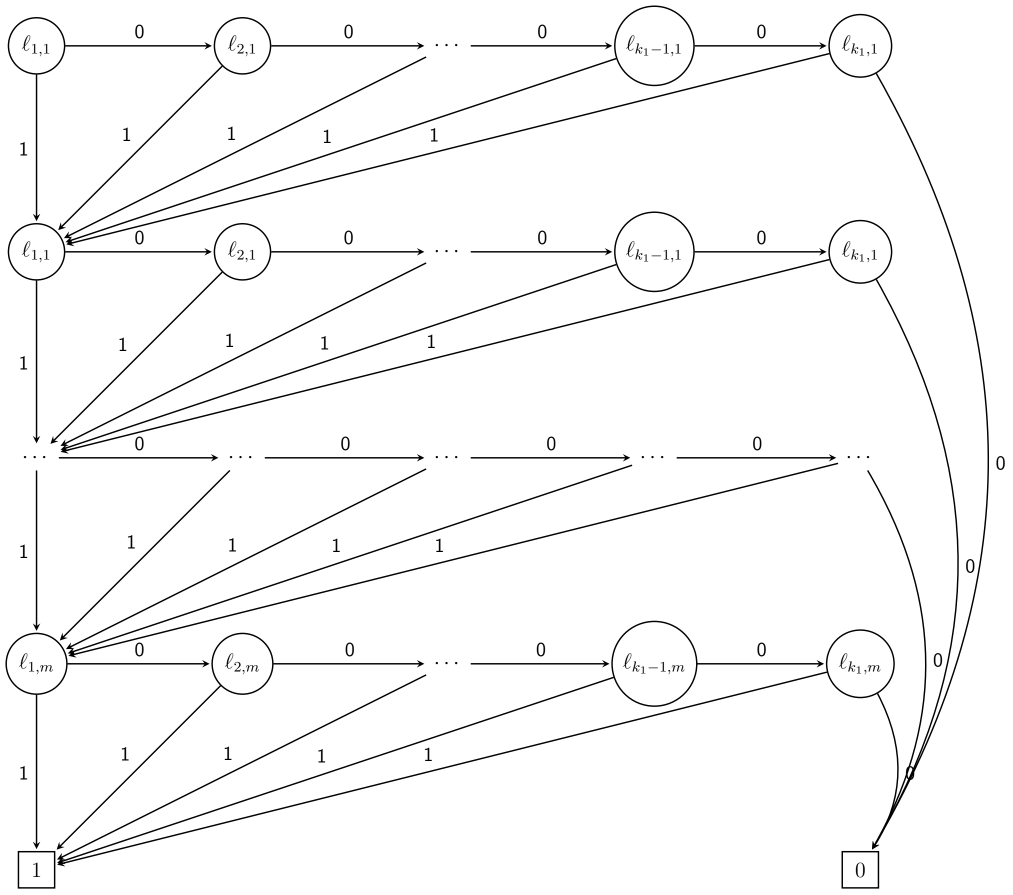

Figure 6: The branching program that computes $\phi(x_1, \dots, x_n)$

3. The branching program shown in 1. is clearly already read-once: each path of each layer queries two different variables and there are no layers that query the same variables.

4. The branching program shown in 2. is not read-once. In fact, the whole idea of the solution is to abuse the possibility of querying multiple times the same variables, which may be shared by multiple clauses. This abuse is necessary due to the lack of information regarding how the CNF is structured: if we knew which clauses contained which literals, we could build a read-once program by skipping the literals that have already been queried.

   However, even if this were the case, the program would have to split into two sub-programs each time a variable is queried for the first time in order to "store" the value. Through this process, the size of the program could become exponential.

For instance, if the first clause contains all the variables of the formula then the program would have to split into 2^n sub-programs, degenerating into a complete decision tree.