Aalto University
School of Science

CS-E4530 Computational Complexity Theory

Lecture 12: Randomised Computation

Aalto University
School of Science
Department of Computer Science

Spring 2017

Agenda

- Monte Carlo algorithms and complexity class **RP**
- Las Vegas algorithms and complexity class **ZPP**
- Unbounded error and complexity class **PP**
- Bounded error and complexity class **BPP**
- **BPP** in view of circuit complexity and **PH**
- Random sources [optional]

(C. Papadimitriou: *Computational Complexity*, Chapter 11)

Case I: A randomised algorithm for Perfect Matching

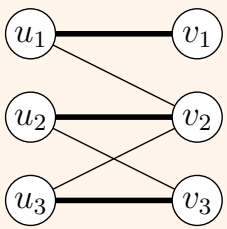
A randomised algorithm $\sim$ ability to flip independent, unbiased coins

Definition

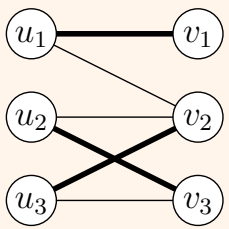
PERFECT MATCHING:
INSTANCE: Bipartite graph $B = (U, V, E)$, where $U = \{u_1, \ldots, u_n\}$, $V = \{v_1, \ldots, v_n\}$, $E \subseteq U \times V$.
QUESTION: Is there a set $E' \subseteq E$ of n edges such that for any two distinct edges $(u, v), (u', v') \in E'$, $u \neq u'$ and $v \neq v'$ (i.e., is there a *perfect matching*)?

A perfect matching can be seen as a permutation π of $1, \ldots, n$ such that $(u_i, v_{\pi(i)}) \in E$ for all $u_i \in U$.

Example (perfect matchings as permutations)



$\pi_1 = \begin{pmatrix} 1 & 2 & 3 \\ 1 & 2 & 3 \end{pmatrix}$



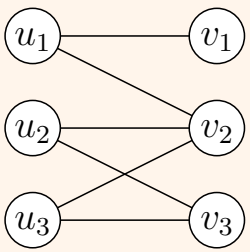
$\pi_2 = \begin{pmatrix} 1 & 2 & 3 \\ 1 & 3 & 2 \end{pmatrix}$

- This is related to computing the determinant of a matrix: Given a graph G, construct an $n \times n$ matrix A^G, where the element $a_{i,j}$ is a variable x_{ij} if $(u_i, v_j) \in E$ and 0 otherwise.

$$\det A^G = \sum_{\pi} \operatorname{sgn}(\pi) \prod_{i=1}^{n} a_{i,\pi(i)}$$

  where π ranges over permutations of n elements and each term in the sum is of the form $\operatorname{sgn}(\pi) a_{1,\pi(1)} \cdots a_{n,\pi(n)}$
- Now a bipartite graph G has a perfect matching iff there is a term for which $a_{i,\pi(i)} \neq 0$ for all $i = 1, \ldots, n$.
- Hence, G has a perfect matching iff $\det A^G$ is not identically 0.

Example (perfect matchings and determinants)



$$A^G = \begin{pmatrix} x_{1,1} & x_{1,2} & 0 \\ 0 & x_{2,2} & x_{2,3} \\ 0 & x_{3,2} & x_{3,3} \end{pmatrix} \qquad \det A^G = x_{1,1}x_{2,2}x_{3,3} - x_{1,1}x_{2,3}x_{3,2}$$

Testing whether $\det A^G$ is identically 0 for a symbolic matrix A^G containing variables can be done by using a randomised algorithm.

Randomised algorithm for perfect matching

Given a matrix $A^G(x_1, \ldots, x_m)$ with m variables:

Choose m random integers $i_1, \ldots, i_m$ (between 0 and M)
Compute $\det A^G(i_1, \ldots, i_m)$ by e.g. Gaussian elimination
If $\det A^G(i_1, \ldots, i_m) \neq 0$
    return "G has a perfect matching"
else
    return "G *probably* has no perfect matching"

Properties:

- No false positives (if "yes" is returned, this is correct).
- False negatives possible (if "no" is returned, this might be wrong).

Monte Carlo Algorithms

A *Monte Carlo algorithm*:

- Polynomial time randomised algorithm
- No false positives
- The probability of false negatives no more than $\frac{1}{2}$.

Example

The previous algorithm yields a Monte Carlo algorithm for PERFECT MATCHING if we set $M = 2m$: it can be shown that the probability of false negatives is no more than $\frac{1}{2}$ when the integers are randomly selected between 0 and $2m$.

Task: convince yourself that Gaussian elimination here can be done in polynomial time (i.e., that the numbers don't grow too large).

☞ If the probability of false negatives is $\varepsilon > \frac{1}{2}$, we can perform k *independent* experiments and answer negatively only if all of them give a negative answer. Then the probability of false negatives is reduced to ε^k (and the running time remains polynomial as k is a constant).

A Generalisation

- A polynomial is identically zero iff its monomial representation equals 0.
  Example: $-xy+(x-y)(x^2+y)+x^2(y-x)+y^2 = -xy+x^3+xy-yx^2-y^2+x^2y-x^3+y^2$ is identically zero.
- Two polynomials, p and q over $x_1,\ldots,x_n$, are equal iff the polynomial $p-q$ is identically zero.
- One can obtain a Monte Carlo algorithm for checking whether a polynomial is not identically zero by using the following lemma:

Lemma (Schwartz-Zippel)

Let $p(x_1,\ldots,x_n)$ be a multivariate polynomial with total degree $d \geq 0$ over a field $\mathbb{F}$. Assume that p is not identically zero. Let S be a finite subset of $\mathbb{F}$ and let $r_1, r_2, \ldots, r_n$ be selected randomly from S. Then $\Pr[p(r_1, r_2, \ldots, r_n) = 0] \leq \frac{d}{|S|}$.

- No deterministic polynomial time algorithm for the task is known.

Random Walks

- A randomised walk algorithm for SAT:

Random walk for SAT

Take any truth assignment T and repeat r times:

1. If there is no unsatisfiable clause, return "satisfiable"
2. Otherwise take any unsatisfiable clause
   Pick any of its literals at random and flip it in T.

After r repetitions return "probably unsatisfiable"

- Is this a Monte Carlo algorithm?
  No false positives but the probability of false negatives is high!
  (An exponential number of repetitions r is needed to achieve low probability for classes of 3SAT problems).

- For 2SAT a Monte Carlo algorithm is obtained by setting $r = 2n^2$.
- Then the probability of false negatives is less than $\frac{1}{2}$.
- The following lemma plays an important role:

Lemma (Markov's Inequality)

If x is a random variable taking non-negative integer values, then for any $k > 0$, $\mathbf{prob}[x \geq k \cdot \mathcal{E}(x)] \leq \frac{1}{k}$
where $\mathcal{E}(x)$ is the expected value of x.

Case II: A Monte Carlo algorithm for COMPOSITE

Theorem (Fermat's Little Theorem)

For a prime N, for all $0 < a < N$, $a^{N-1} = 1 \bmod N$.

- Fermat test for COMPOSITE:
  1. Pick random residue a modulo N.
  2. If $a^{N-1} \neq 1 \bmod N$, then return "N is composite"
  3. Otherwise answer "N is probably prime"
- Is this a Monte Carlo algorithm?
  - By Fermat's Little Theorem, no false positives.
  - But is it the case that for any composite, for at least half of its nonzero residues a, $a^{N-1} \neq 1 \bmod N$?

    No, Carmichael numbers are exceptions. A *Carmichael number* is a composite positive integer N which satisfies the congruence $b^{N-1} = 1 \bmod N$ for all integers b which are relatively prime to N.

    E.g., $561 = 3 \times 11 \times 17$ is the first Carmichael number and 320 of 560 residues fail the test.

- A Monte Carlo algorithm for testing compositeness of N:

Solovay–Strassen compositeness test

1. Generate a random integer M between 2 and $N-1$;
2. If $(M,N) > 1$ then return "N is a composite"
   else
   if $(M|N) \neq M^{\frac{N-1}{2}} \mod N$ then return "N is a composite"
   else return "N is probably a prime".

where (M,N) is the greatest common divisor of M and N, and $(M|N)$ is the Jacobi symbol.

- This is a Monte Carlo algorithm:
  (M,N) and $(M|N)$ can be computed in polynomial time, no false positives, and the probability of false negative at most $\frac{1}{2}$.
- *Miller–Rabin test* is an another test for compositeness used in cryptography applications, see e.g.
  `http://en.wikipedia.org/wiki/Miller-Rabin_primality_test`

Randomised Complexity Classes

- Randomised algorithms (such as Monte Carlo ones) can be analysed using nondeterministic Turing machines but *with a different interpretation of what it means for such a machine to accept its input.*
- No coin-flipping is needed in the Turing machine!

The class RP

Definition

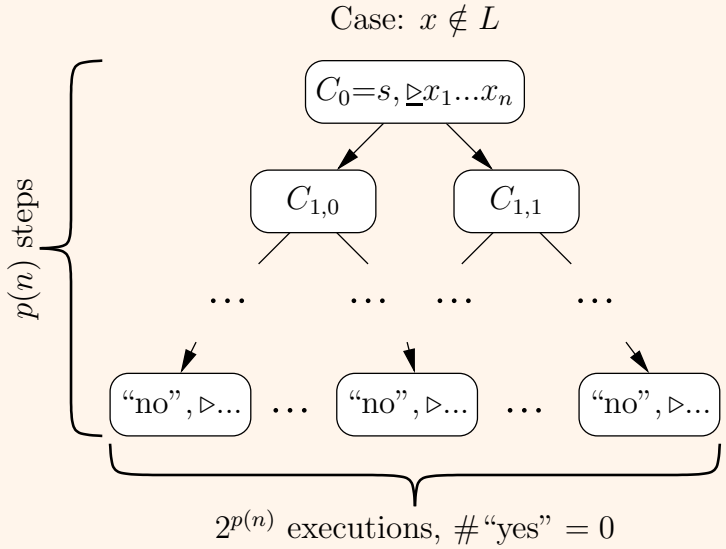
Let L be a language. A polynomial time *Monte Carlo Turing machine* for L is a nondeterministic Turing machine N satisfying:

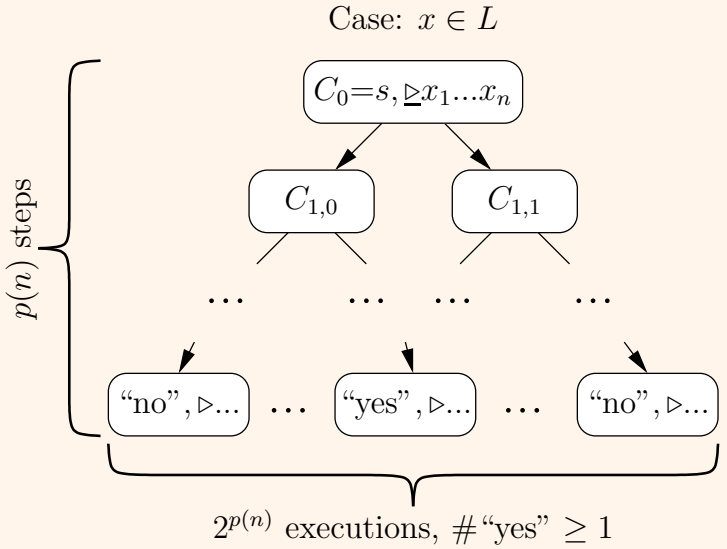
1. there are exactly two nondeterministic choices at each step;
2. the number of steps in each computation for an input of length n is $p(n)$, a polynomial;
3. for each input x:
   - If $x \in L$, then at least half of the $2^{p(|x|)}$ computations of N on x halt with "yes".
   - If $x \notin L$, then all the $2^{p(|x|)}$ computations halt with "no".

The class of all languages with polynomial time Monte Carlo Turing machines is denoted by **RP** (randomised polynomial time).

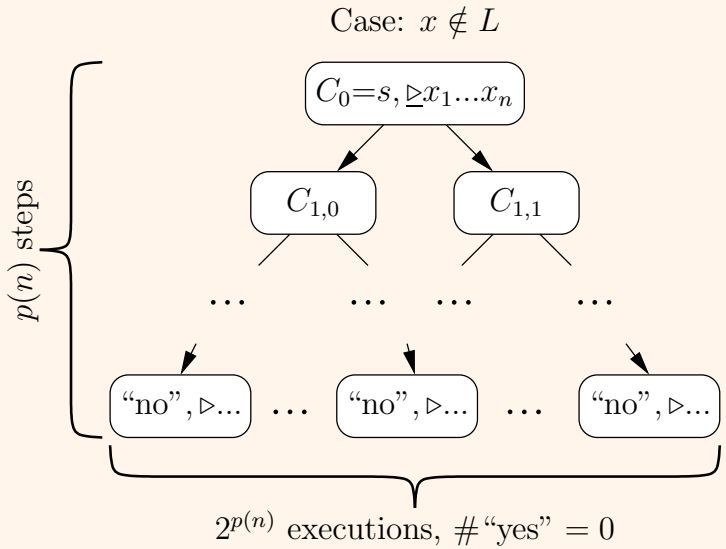
Example

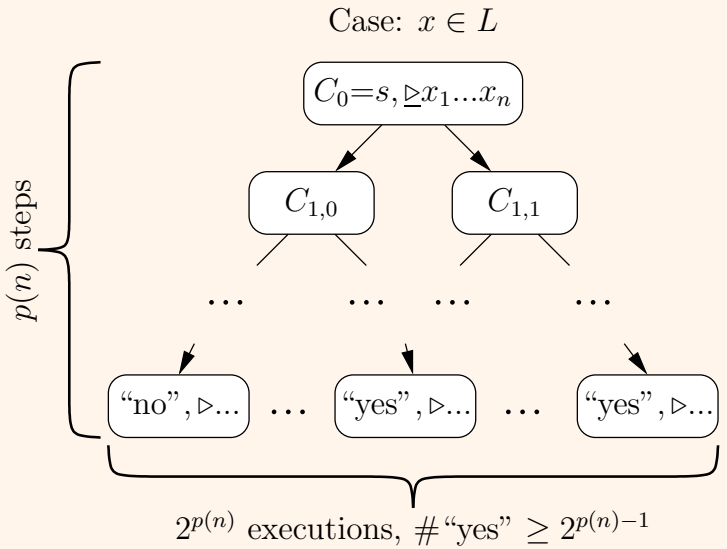
Executions of a standardised Turing machine deciding a language $L \in \mathbf{NP}$:





Executions of a Monte Carlo Turing machine deciding a language $L \in \mathbf{RP}$:





RP captures Monte Carlo algorithms

- All nondeterministic steps are “coin flips”.
- There are no false positive answers.
- All computations equiprobable (with probability $2^{-p(|x|)}$).
- The probability of a false negative is at most $\frac{1}{2}$:
  - Given a Monte Carlo Turing machine N for a language L: a false negative answer is given if N halts with “no” on $x \in L$.
  - This happens in at most half of the $2^{p(|x|)}$ computations each having a probability of $2^{-p(|x|)}$.
  - Hence, the probability of a false negative is at most $\frac{1}{2} \cdot 2^{p(|x|)} \cdot 2^{-p(|x|)} = \frac{1}{2}$

Boosting the acceptance probability

The power of $\mathbf{RP}$ would not be affected if the probability of acceptance were not $\frac{1}{2}$ but any number $0 < \varepsilon < 1$:

- If $\varepsilon < \frac{1}{2}$, perform k independent repeats of the algorithm and accept iff at least one of the k computations accepts, otherwise reject.
- Now the probability of a false negative is at most $\leq (1-\varepsilon)^k$.
- When $k \geq (-\frac{1}{\ln(1-\varepsilon)})$, the probability of a false negative is $\leq \frac{1}{2}$.
- The running time is k times the original.
- As $-\frac{1}{\ln(1-\varepsilon)} \approx \frac{1}{\varepsilon}$, ε could even be of the form $\frac{1}{p(n)}$ where $p(n)$ is a polynomial, and the runtime of the "boosted" algorithm would still remain polynomial.

Structural properties of $\mathbf{RP}$

- $\mathbf{P} \subseteq \mathbf{RP} \subseteq \mathbf{NP}$
- Given a Turing machine, it is not easy to determine whether it is a Monte Carlo machine, i.e. whether it for all inputs either rejects "unanimously" or accepts "by majority".
  ☞ A "semantic" class (like $\mathbf{NP} \cap \mathbf{coNP}$).
  ☞ No known complete problems.

For example, $\mathbf{P}$ and $\mathbf{NP}$ are "syntactic" classes with complete problems.

The class ZPP

- **coRP**: the languages having Monte Carlo machines with no false negatives and a limited number of false positives.
- PRIMES is in **coRP** (Solovay & Strassen, Miller & Rabin 1977).
- $\mathbf{ZPP} = \mathbf{RP} \cap \mathbf{coRP}$ is the class of languages with *Las Vegas algorithms* (polynomial randomised algorithms with zero probability of error).
- A Las Vegas algorithm = two Monte Carlo algorithms: one for the language and one for its complement.
- Running k independent experiments with both algorithms:
  - Sooner or later a definite answer will come: either a positive answer from the algorithm with no false positives or a negative one from the algorithm with no false negatives.
  - The probability of a definite answer is at least $1 - 2^{-k}$.
- PRIMES is also in **RP** and thus in **ZPP**.[1]

[1] In fact PRIMES is in **P** (Agrawal, Kayal & Saxena 2002).

The class PP

- Consider the problem MAJSAT:
  Given a Boolean expression, is it true that the majority of the 2^n truth assignments to its n variables satisfy it?
- It is not clear that MAJSAT is in **NP** (and thus less likely in **RP**).
- **PP** is the class of languages L having a nondeterministic polynomially bounded Turing machine N (precise and with two choices each step) such that for all inputs x, $x \in L$ iff more than half of the computations of N on input x end up accepting.

Theorem

MAJSAT is **PP***-complete.*

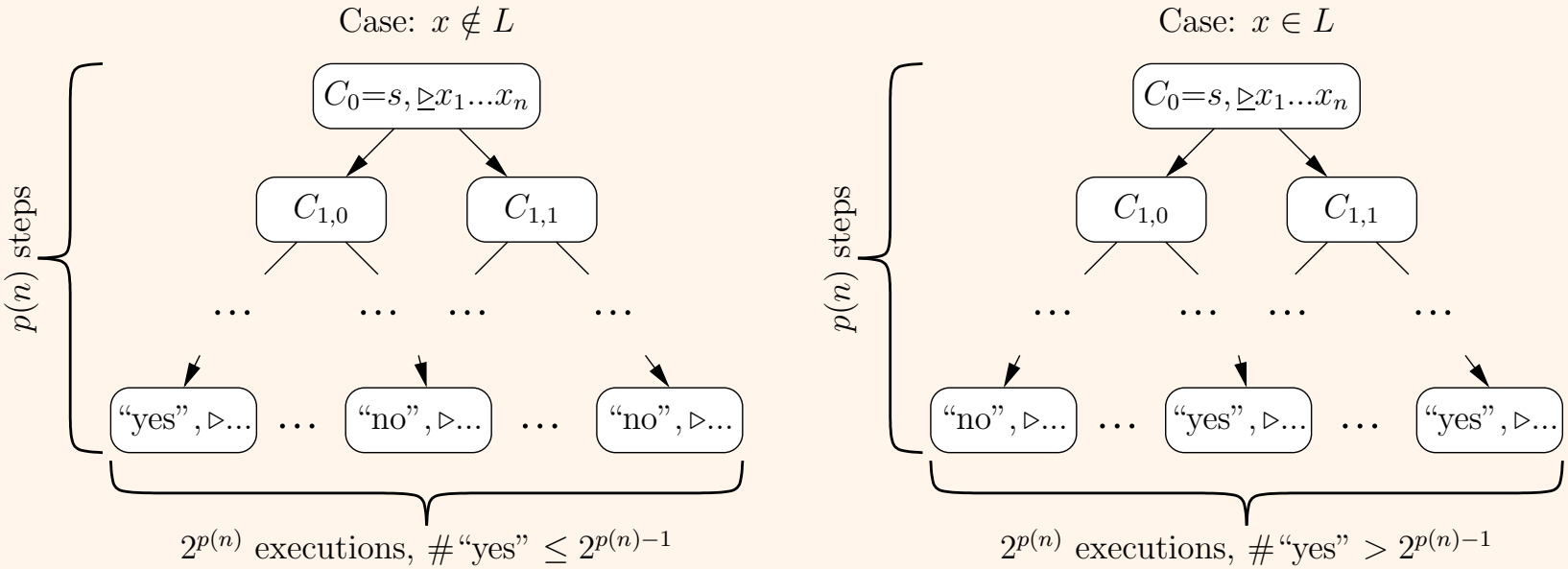
Theorem

$\mathbf{NP} \subseteq \mathbf{PP}$.

PP is closed under complement.

Example

Executions of a standardised Turing machine deciding a language $L \in \mathbf{PP}$:



- $\mathbf{P} \subseteq \mathbf{ZPP} \subseteq \mathbf{RP} \subseteq \mathbf{NP} \subseteq \mathbf{PP} \subseteq \mathbf{PSPACE}$
- $\mathbf{ZPP}, \mathbf{RP}$ are plausible notions of efficient randomised computations (but $\mathbf{PP}$ is not).
- $\mathbf{PP}$ cannot be used algorithmically because acceptance by majority is too fragile: the acceptance probability can be $\frac{1}{2} + 2^{-p(|x|)}$ and there is no plausible efficient experimentation that can detect such accepting behaviour (see below).

Detecting the more likely side of a biased coin

- Consider the following problem:
  You have a biased coin with one side having probability $\frac{1}{2}+\varepsilon$ and the other $\frac{1}{2}-\varepsilon$. How to detect which side is more likely?
  Solution: Flip the coin many times and pick the side that appeared the most times. But how many times?
- *The Chernoff bound:*
  Suppose that $x_1,\ldots,x_n$ are independent random variables taking the values 1 and 0 with probabilities p and $1-p$, respectively, and consider their sum $X=\sum_{i=1}^{n} x_i$. Then for all $0\leq\theta\leq 1$,
  $\mathbf{prob}[X\geq(1+\theta)pn]\leq e^{-\frac{\theta^2}{3}pn}$.
- The probability that X deviates from its expected value (pn) decreases exponentially with the deviations.

- Corollary:

  If $p = \frac{1}{2} + \varepsilon$ for some $\varepsilon > 0$, then $\mathbf{prob}[\sum_{i=1}^{n} x_i \leq \frac{n}{2}] \leq e^{-\frac{\varepsilon^2}{6}n}$.
- A bias of ε can be detected with reasonable confidence by taking a majority of about $\frac{1}{\varepsilon^2}$ experiments ($e^{-\frac{\varepsilon^2}{6\varepsilon^2}} = 0.85$).
- For a $\mathbf{PP}$ problem the bias ε can be as small as $2^{-p(|x|)}$: an exponential number of repetitions of the algorithm is required to determine the correct answer with reasonable confidence.

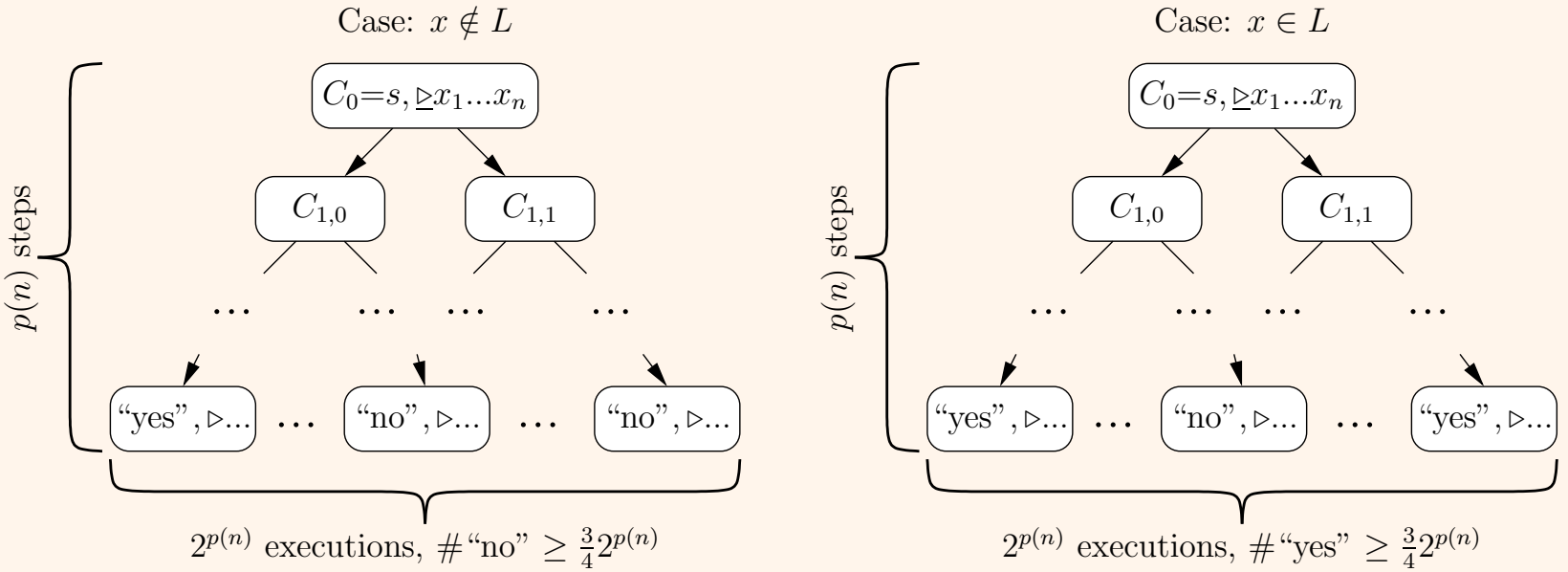
The class BPP

- A realistic notion of randomised computation between **RP** and **PP**.
- **BPP** is the class of languages L having a nondeterministic polynomially bounded Turing machine N (precise and with two choices each step) such that for all inputs x,
  - if $x \in L$, then at least $\frac{3}{4}$ of the computations of N on x accept
  - if $x \notin L$, then at least $\frac{3}{4}$ of the computations of N on x reject

  That is, N has a bounded probability of error.
- $\mathbf{RP} \subseteq \mathbf{BPP} \subseteq \mathbf{PP}$.
- Open: Is $\mathbf{BPP} \subseteq \mathbf{NP}$ or $\mathbf{NP} \subseteq \mathbf{BPP}$? (Both alternatives are generally considered implausible.)
- **BPP** is closed under complement.
- Semantic class.
- No known complete problems.

Example

Executions of a standardised Turing machine deciding a language $L \in \mathbf{BPP}$:



BPP in view of circuit complexity and PH

Theorem

All languages in $\mathbf{BPP}$ *have polynomial circuits.*

Corollary

$\mathbf{BPP} \subseteq \Sigma_2^p$

Corollary

$\mathbf{BPP} \subseteq \Sigma_2^p \cap \Pi_2^p$

Proof. $\mathbf{BPP}$ is closed under complement. Hence, if $L \in \mathbf{BPP}$, $\overline{L} \in \mathbf{BPP} \subseteq \Sigma_2^p$ implying $L \in \Pi_2^p$. $\square$

Theorem

If SAT $\in \mathbf{BPP}$ *(and hence has polynomial circuits), then the polynomial time hierarchy collapses to the second level.*

Random Sources [optional]

- In order to implement randomised algorithms (e.g., those for **RP** and **BPP**), we need a *source of random bits*.
- *A perfect random source* is a random variable with values that are infinite sequences $(x_1, x_2, \ldots)$ of bits such that for all $n > 0$ and for all $(y_1, y_2, \ldots, y_n) \in \{0,1\}^n$

$$\mathbf{prob}[x_i = y_i, i = 1, \ldots, n] = 2^{-n}$$

- A Monte Carlo algorithm could be implemented using a random source by generating a sequence $(x_1, x_2, \ldots)$ of bits and choosing the transition at each step i according to the bit x_i.

- A problem: where to find a perfect random source?
- A perfect random source should be
  - *independent*: the probability that $x_i = 1$ does not depend on the previous or future outcomes
  - *fair*: the probability that $x_i = 1$ should be exactly $\frac{1}{2}$.

Example (perfect random source)

earlier bits				next bit
0	1	...	$i-1$	i
0	0	...	0	0 with prob. 0.5 1 with prob. 0.5
⋮	⋮		⋮	⋮
0	1	...	1	0 with prob. 0.5 1 with prob. 0.5
⋮	⋮		⋮	⋮
1	1	...	1	0 with prob. 0.5 1 with prob. 0.5

- The important requirement is independence:
  Any independent but unfair random sequence of bits can be turned into a fair one as follows:
  (i) Break the sequence in pairs and
  (ii) interpret: $01 \rightsquigarrow 0$, $10 \rightsquigarrow 1$ (ignoring 00 and 11).
- Now if the probability of 1 is p, the probability of 10 is $p(1-p)$ which equals to that of 01.
- To get a perfect random sequence of length n we need a sequence of expected length $\frac{2n}{1-c}$ where is $c = p^2 + (1-p)^2$ is the coincidence probability of the source.
- The real problem of physically implementing perfect random sources is that any physical process tends to be affected by its previous outcomes (and circumstances leading to it).
- Randomness in mathematical or computational process:
  pseudorandom number generators
  Typical congruential approach ($x_{i+1} = ax_i + b \bmod c$) is terrible (easy to predict bits and even deduce "secret" parameters).

Slightly random sources

- Perfect random sources seem to be hard to implement physically.
- A weaker concept: *δ-random source*
  Let δ be a number $0 < \delta \leq \frac{1}{2}$ and p any function $\{0,1\}^* \mapsto [\delta, 1-\delta]$ (a highly complex function unknown to us). The δ-random source S_p is a random variable with infinite bit sequences as values where the probability that the first n bits have the values $y_1, y_2, \ldots, y_n$ is

$$\prod_{i=1}^{n} (y_i p(y_1 \ldots y_{i-1}) + (1 - y_i)(1 - p(y_1 \ldots y_{i-1})))$$

  (Notice: the probability that the ith bit is 1 is $p(y_1 \ldots y_{i-1})$, a number between δ and $1-\delta$ that depends in an arbitrary way on all previous outcomes $y_1 \ldots y_{i-1}$).

- Now $\delta \leq p(y_1 \dots y_{i-1}) \leq 1 - \delta$
- A $\frac{1}{2}$-random source is a perfect random source.
- A δ-random source with $\delta < \frac{1}{2}$ is a *slightly random source*.
- Slightly random sources: Geiger counters, Zehner diodes, coins

Example (perfect vs. slightly random sources)

earlier bits				next bit	
0	1	...	$i-1$	i	
0	0	...	0	0	with prob. 0.5
				1	with prob. 0.5
0	1	...	1	0	with prob. 0.5
				1	with prob. 0.5
1	1	...	1	0	with prob. 0.5
				1	with prob. 0.5

vs

earlier bits				next bit	
0	1	...	$i-1$	i	
0	0	...	0	0	with prob. $1 - p(0,0,...0)$
				1	with prob. $p(0,0,...,0) \in [\delta, 1-\delta]$
0	1	...	1	0	with prob. $1 - p(0,1,...,1)$
				1	with prob. $p(0,1,...,1) \in [\delta, 1-\delta]$
1	1	...	1	0	with prob. $1 - p(1,1,...,1)$
				1	with prob. $p(1,1,...,1) \in [\delta, 1-\delta]$

- In the worst case slightly random sources appear to be useless for running randomised algorithms.
- Suppose a Monte Carlo algorithm is driven by random bits generated by a δ-random source with δ much smaller than $\frac{1}{2}$.
- In the worst case the algorithm can make choices which very often lead to a false negative outcome:
  consider an *adversary* who knows the algorithm and monitors its executions including the random choices and sets the values of p on the basis of this.

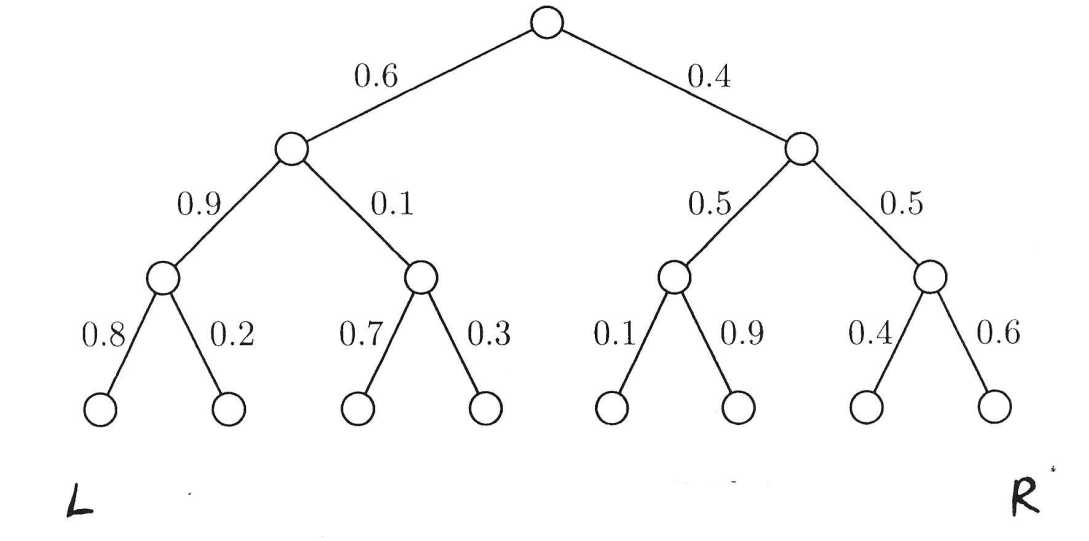
The classes δ-RP and δ-BPP

- Let N be a precise, polynomially bounded nondeterministic Turing machine with exactly two choices per step (the 0-choice and the 1-choice).
- On input x the computation $N(x)$ is in effect a full binary tree of depth $n = p(|x|)$ (having $2^{n+1} - 1$ nodes of which 2^n are leaves and $2^n - 1$ internal).
- Let δ be a number $0 < \delta < \frac{1}{2}$. A *δ-assignment F* is a mapping from the set of edges of $N(x)$ to the interval $[\delta, 1 - \delta]$ such that the two edges leaving each internal node are assigned numbers adding up to one (function p is precisely the assignment F on the 1-choice out of the node).

- Given a δ-assignment F then for each leaf l, $\mathbf{prob}[l] = \prod_{a \in P(l)} F(a)$ where $P(l)$ is the path from the root to leaf l.
- $\mathbf{prob}[N(x) = \text{“yes”} | F]$ is the sum of $\mathbf{prob}[l]$ for all “yes” leaves l of $N(x)$.
- We say that a language L is in **δ-RP** if there is a nondeterministic machine N, standardised as above, such that
  if $x \in L$, then $\mathbf{prob}[N(x) = \text{“yes”} | F] \geq \frac{1}{2}$ and
  if $x \notin L$, then $\mathbf{prob}[N(x) = \text{“yes”} | F] = 0$
  for *all δ-assignments F*.
- A language L is in **δ-BPP** if there is a nondeterministic machine N such that
  if $x \in L$, then $\mathbf{prob}[N(x) = \text{“yes”} | F] \geq \frac{3}{4}$ and
  if $x \notin L$, then $\mathbf{prob}[N(x) = \text{“no”} | F] \geq \frac{3}{4}$
  for *all δ-assignments F*.

Example

Consider the computation tree and 0.1-assignment F [Papadimitriou, 1994]



- For the leftmost leaf L
  $\mathbf{prob}[L] = \prod_{a \in P(L)} F(a) = 0.6 \cdot 0.9 \cdot 0.8 = 0.432$
- and for the rightmost leaf R
  $\mathbf{prob}[R] = \prod_{a \in P(R)} F(a) = 0.4 \cdot 0.5 \cdot 0.6 = 0.120$

- 0-**RP** = 0-**BPP** $=$ **P**
- $\frac{1}{2}$-**RP** = **RP**, $\frac{1}{2}$-**BPP** = **BPP**
- What about intermediate values of δ?
- A slightly random source can be used to simulate any randomised algorithm with polynomial loss of efficiency.
- This holds even if we assume that a hypothetical "adversary" can bias the slightly random source in arbitrary ways to lead the random algorithm to false answers.

Theorem

For any $\delta > 0$, δ-**BPP** $=$ **BPP**.

Proof. δ-**BPP** $\subseteq$ **BPP** clear; **BPP** $\subseteq$ δ-**BPP** tricky (see below).

Simulating a randomised algorithm

- Assume that $L \in \mathbf{BPP}$, i.e., L is decided by a NTM N by clear majority.
- Construct a machine N' deciding L by clear majority when driven by *any* slightly random source.
- The basic idea: confuse the "adversary" by shattering the slightly random bits using inner products.
- Inner product of two sequences of bits $\kappa = (\kappa_1, \ldots, \kappa_k)$ and $\lambda = (\lambda_1, \ldots, \lambda_k)$ is the bit obtained by $\kappa \cdot \lambda = \sum_{i=1}^{k} \kappa_i \lambda_i \bmod 2$.

Simulating machine N'

- On input x, let $n = p(|x|)$ be the length of N's computation on x and let k be an integer (a parameter depending on n and δ).
- Generate n sequences of bits (blocks) $\beta_1, \ldots, \beta_n$ using a δ-random source where each β_i contains k bits.
- Do 2^k parallel simulations of N with the sequences of choices

$$\begin{aligned} T &= \{(\beta_1 \cdot \kappa, \ldots, \beta_n \cdot \kappa) : \kappa = 0, 1, \ldots, 2^k - 1\} \\ &= \{(\beta_1 \cdot 0, \ldots, \beta_n \cdot 0), \ldots, (\beta_1 \cdot (2^k - 1), \ldots, \beta_n \cdot (2^k - 1))\} \end{aligned}$$

- Of the 2^k answers, adopt the majority one as the answer of N'.

Simulating machine N' — cont'd

To reduce the probability of a false answer by N' to at most 1/4:

- Assume that probability of wrong answer by N is 1/32 (instead of 1/4).
- Set $k = \lceil \frac{\log n + 5}{2\delta - 2\delta^2} \rceil$
- Now N' works within time $O(n2^k) = O(nn^{\frac{1}{2\delta-2\delta^2}}) = O(p(|x|)^{1+\frac{1}{2\delta-2\delta^2}})$, i.e. in polynomial time.

Corollary

*For any $\delta > 0$, δ-**RP** = **RP**.*

Learning Objectives

- The concepts of Monte Carlo and Las Vegas algorithms
- The key randomised complexity classes: $\mathbf{RP}, \mathbf{ZPP}, \mathbf{PP}, \mathbf{BPP}$
- The concepts of perfect and slightly random sources