Probabilistic Algorithms

Paul Cotofrei

information management institute

PA 2016

FAQ

Time/Place
- B29
- Monday 13:15 - 15:00 (course), 15:15 - 17:00 (lab)

Lab
- first half semester: individual exercises
- second half semester: team project (min. one, max. three students)
- Lab grade: 40% of final grade

Exam
- end-term (2 hours) written exam, during the last semester week: 60% of grade
- for 3 credits - individual exercise (40%) and written exam (60%) with problems covering the first part of the course

Course support https://moodle.unine.ch/course/view.php?id=240

Course content

Main chapters

- Probabilistic Algorithms - basic concepts
- Optimization problems - overview
- Deterministic optimization algorithms
- Heuristic optimization algorithms
  - stochastic local search, simulated annealing, genetic algorithms and evolutionary strategies
- Stochastic optimization for random noise problems
  - non-linear root finding
  - stochastic gradient algorithms, finite difference, simultaneous perturbation

References

- Johannes J. Schneider, Scott Kirkpatrick: "Stochastic Optimization", 2006, Springer.
- James C. Spall: "Introduction to Stochastic Search and Optimization", 2003, John Wiley & Sons.
- MATLAB Documentation: http://www.mathworks.com/access/helpdesk/help/techdoc/index.html

More references for curious students

- Rajeev Motwani, Prabhakar Raghavan: "Randomized algorithms", Cambridge University Press, 2000
- James H. Gentle: "Random number generation and Monte Carlo methods", 2003, Springer

Outline

Probabilistic algorithms - basic concepts

Random number generators

Some Fundamental Ideas

- What is an algorithm? No generally accepted definition.
- Almost 200 years of research.
- Something like: defining generalized processes for the creation of an *output* from an *input* by the manipulation of distinguishable symbols (counting numbers) with finite collections of rules that can be performed.
- Algorithms in Computer Science:
  - The most common approach: *imperative programming*
    - It describes the algorithm in a "mechanical" way.
    - The sequence of steps is given.
  - Other concepts are: *functional programming* or *logical programming*
    - This paradigms emphasis *What* has to be done and not *How* it has to be done
    - They use a functional or logical approach.

Algorithm Execution

The execution of the algorithm (with a fixed set of instructions) can be *deterministic* or *non-deterministic*

- **Deterministic**: For the same input the same sequence of operations is executed
- **Non-Deterministic**: For the same input different sequences of operations are possible

Goal of a Deterministic Algorithm

- The solution produced by the algorithm is (always) correct

Deterministic algorithm issues

- The running time of an algorithm may be quite high.
- Difficult to design an algorithm with good running time,

Possible solutions

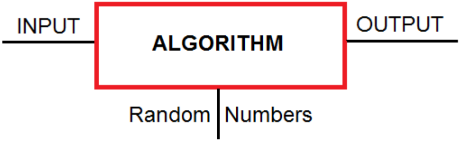
- Efficient Heuristics
- Approximation Algorithms
- *Randomized Algorithms*

Probabilistic Algorithms

The Paradigm

- Difficult to guarantee a good choice for each input? Make a random choice and hope that it is good.

A *Probabilistic* (or *Randomized*) *Algorithm* is an algorithm which employs a degree of randomness as part of its logic.



- During execution, it takes random choices depending on those random numbers.
- The behavior (output) can vary if the algorithm is run multiple times on the same input.

Pros and Cons

Pros:

- making a random choice is fast
- there is no worst case inputs
- probabilistic algorithms are often simpler and faster than their deterministic counterparts.

Cons:

- In the worst *execution* case, the algorithm may be very slow.
- For some algorithms, there is a positive probability of getting incorrect answer.
- Getting true random numbers is almost impossible !

Refreshing Probability Theory

- Discrete sample space Ω: a set of elementary events $e_i, i = 1..n$, the possible outputs of an experience with non-deterministic behavior. Examples :
  - flipping a coin toss $\Rightarrow \Omega = \{H, T\}$
  - throwing a dice $\Rightarrow \Omega = \{1, 2, 3, 4, 5, 6\}$
- $\mathcal{P}(\Omega)$: all the subsets of Ω
  - An element of $\mathcal{P}(\Omega)$ is an event
  - Example: $\mathcal{P}(\Omega) = \{\emptyset, \Omega, H, T\}$
- Probability measure: a function $P : \mathcal{P}(\Omega) \to [0, 1]$ satisfying
  - $P(\Omega) = 1$, $P(\emptyset) = 0$
  - $P(E_1 \cup E_2) = P(E_1) + P(E_2)$, if $E_1 \cap E_2 = \emptyset$
  - Example (fair coin toss):
    $P(\emptyset) = 0, P(H) = 0.5, P(T) = 0.5, P(\Omega) = 1$

Refreshing Probability Theory

- Random variable: a function $X : \Omega \to \mathbf{R}$; coin toss example:
  - X - a random variable defining the number of heads resulting from a coin toss
  - $X(H) = 1, X(T) = 0$;
  - If Ω is discrete, then the random variable X is also discrete; denote $Y = \{X(e_1), .., X(e_n)\}$
- Event defined by a discrete random variable:

$$\{X = y\} = \{e \in \Omega | X(e) = y\}$$

  where $y \in Y$; example (coin toss and variable X):
  - Events $\{X = 0\}$, $\{X = 1\}$
- Probability mass function: a function $f : Y \to [0, 1]$, $f(y) = P(\{X = y\})$, where P is a probability measure
  - Example (fair coin toss): $\begin{array}{c} y \\ f(y) \end{array} \left(\begin{array}{cc} 0 & 1 \\ 0.5 & 0.5 \end{array} \right)$

Refreshing Probability Theory

- Continuos random variables: the set Y is an interval of $\mathbf{R}$
  - Example: X - a random variable measuring the speed of a car
- Event defined by a continuous random variable: $\{a \leq X \leq b\}$
- Cumulative distribution function $F : \mathbf{R} \rightarrow [0, 1]$

$$F(y) = P(X \leq y)$$

- Probability density function: $f(\cdot) = \frac{\partial F}{\partial y}$
  - $f(x) \geq 0, \forall x \in \mathbf{R}$
  - $\int_{-\infty}^{\infty} f(x)dx = 1$
  - $P(a \leq X \leq b) = \int_{a}^{b} f(x)dx$

Refreshing Probability Theory

- Expected value of discrete X: $E[X] = \sum_{y \in Y} y \cdot P(\{X = y\})$
  - Example (fair coin toss and variable X):
    $E[X] = 0 \cdot P(\{X = 0\}) + 1 \cdot P(\{X = 1\}) = 0 \cdot \frac{1}{2} + 1 \cdot \frac{1}{2} = \frac{1}{2}$
- Expected value of continuous X: $E[X] = \int_{-\infty}^{\infty} x f(x) dx$
- Properties of $E[X]$
  - $E[X + Y] = E[X] + E[Y]$
  - $E[cX] = cE[X]$, where c is a constant

A First Example

Problem: Given A, B and C three matrices $n \times n$, verify

$$AB = C$$

- Deterministic algorithm: compute $A \cdot B$ and check if the resulted matrix is equal C
  - Complexity: $O(n^3)$ - basic algorithm - or $O(n^{2.37})$ - fastest algorithm
- Probabilistic algorithm:
  1. Choose randomly n numbers $r_1, \ldots, r_n$ from $\{0, 1\}$
  2. Construct the vector $r = (r_1, r_2, \ldots, r_n)^T$
  3. If $A \cdot (B \cdot r) = C \cdot r$ THEN return *True* ELSE return *False*

  - Complexity: $O(n^2) < O(n^{2.37})$, but there is a positive probability to get a wrong answer

Error Probability

Theorem: If $AB \neq C$ then $P[A(Br) = Cr] \leq 1/2$.
Proof:

- Let be $D = AB - C \neq 0$
- $A(Br) = Cr \Rightarrow A(Br) - Cr = 0 \Leftrightarrow (AB)r - Cr = 0 \Leftrightarrow (AB - C)r = 0 \Leftrightarrow Dr = 0$
- $D \neq 0 \Rightarrow \exists i, j, 1 \leq i, j \leq n$, such that $d_{ij} \neq 0$. By re-indexing the elements of initial matrices, we can always get $i = 1, j = n$.
- $Dr = 0 \Rightarrow D_{1.}r = 0 \Rightarrow \sum_{k=1}^{n} d_{1k} r_k = 0 \Rightarrow r_n = -\frac{1}{d_{1n}} \sum_{k=1}^{n-1} d_{1k} r_k$
- Suppose that $r_1, r_2, .., r_{n-1}$ have been chosen randomly from $\{0, 1\}$. If $r_n \in \{0, 1\}$ we have one chance over two to select it. If not, we have zero chances to select it. Anyway,

$$P[Dr = 0] \leq 1/2$$

Error Probability

- This probabilistic algorithm is a one-sided error algorithm: only for output *True* there is a probability to get a wrong answer.
- How to reduce the error probability ?
  1. Select the random numbers r_i from $\{1, 2, ..., m\}$, $m > 2$
  2. Repeat the algorithm k times: the probability to get the wrong answer k successive times is $\leq \frac{1}{2^k}$

A Second Example

Problem: Given two polynomials, Q and R over n variables, check if $Q \equiv R$

- Deterministic algorithm: there is no known efficient (i.e. polynomial time) algorithm for this problem !
  - obvious idea : expand the two polynomials as sum of monomials and compare the coefficients
- Probabilistic algorithm:
  1. Calculate $T = Q - R$
  2. Select randomly n numbers $r_1, \ldots, r_n$ from a finite set S
  3. If $T(r_1, ..., r_n) = 0$ THEN return *True* ELSE return *False*

Theorem: If $Q \not\equiv R$ then $P(T(r_1, ..., r_n) = 0) \leq d/|S|$, where d is the degree of T and $|S|$ the size of S

So if we take $|S| > 2d$, then the probability for a wrong answer is less than $1/2$.

A Third Example

Problem: Given a set S with n (comparable) element, sort them in increasing order.

- Deterministic algorithm: Quicksort
  1. If S has one or zero elements, return S. Otherwise continue.
  2. Choose the first element of S as a pivot; call it p
  3. Compare all elements of S to p and create two sublists
     a. S_1 contains all elements less than p
     b. S_2 contains all elements greater than p
  4. Apply Quicksort to S_1 and S_2
  5. Return S_1, p, S_2
- The worst case time complexity is $O(n^2)$
- The average time complexity (the input is selected randomly from the set of all permutations of elements $\{1, ..., n\}$) is $O(n \log n)$

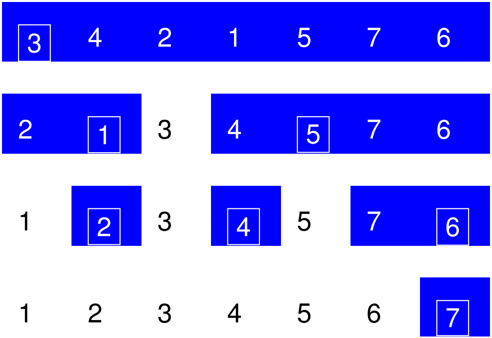
Random Quicksort

Probabilistic Algorithm: RandQSORT

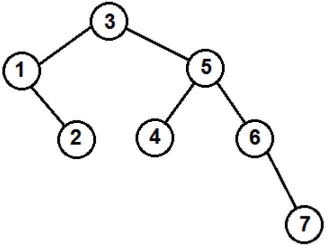
1. If S has one or zero elements, return S. Otherwise continue.
2. Choose randomly an element of S as a pivot; call it p
3. Compare all elements of S to p and create two sublists
   a. S_1 contains all elements less than p
   b. S_2 contains all elements greater than p
4. Apply RandQSORT to S_1 and S_2
5. Return S_1, p, S_2

- For a fixed input S, RandQSORT always returns the right output
- For a fixed input S, at each call, RandQSORT may execute a different number of steps (comparisons)

Example of RandQSORT execution



Recursion Tree



The list of pivots (in order of selection):

$$\pi = \{3, 1, 5, 2, 4, 6, 7\}$$

The Worst Case

Execution	nr. compar.	probability
3 4 2 $\boxed{1}$ 5 7 6	6	1/7
1 3 4 $\boxed{2}$ 5 7 6	5	1/6
1 2 $\boxed{3}$ 4 5 7 6	4	1/5
1 2 3 $\boxed{4}$ 5 7 6	3	1/4
1 2 3 4 $\boxed{5}$ 7 6	2	1/3
1 2 3 4 5 7 $\boxed{6}$	1	1/2
1 2 3 4 5 6 $\boxed{7}$	0	1

$$\#comparisons = \sum_{i=1}^{n-1} i = O(n^2);\ \ probability = \prod_{i=1}^{n} \frac{1}{i} = \frac{1}{n!}$$

Types of Randomized Algorithms

- There are two main families: *Monte Carlo* algorithms and *Las Vegas* algorithms.
- Las Vegas: a randomized algorithm that
  - always gives *correct/optimal* results;
  - the execution time is variable
  - Example: RandQSORT
- Monte Carlo: a randomized algorithm that
  - the running time is *deterministic*
  - the output may be incorrect with a certain probability
  - Example: Check matrices equality
  - For *Decision problem (Yes/No)*:
    - **one-sided error** Monte Carlo algorithm: Yes or No always correct
    - **two-sided error** Monte Carlo algorithm (a non-zero probability to err for both outputs)

Las Vegas $\leftrightarrow$ Monte Carlo

Las Vegas to Monte Carlo

- Algorithm LV with expected running time $f(n)$: if T is the random variable expressing the running time, then $E[T] = f(n)$
- Algorithm MC: stop LV after $\alpha f(n)$ time ($\alpha > 0$)
  - Error source: the algorithm is stopped before completion
  - Estimation of error: the MC algorithm returns a wrong result if $T > \alpha f(n)$; using Markow inequality, $P(T \geq \alpha f(n)) \leq \frac{1}{\alpha}$

Monte Carlo to Las Vegas = Macao

- Algorithm Monte Carlo with deterministic running time at most $f(n)$ and success probability at least $p(n)$
- Suppose there is an algorithm CHECK to verify the correctness of the MC output in time $g(n)$
- Algorithm LV: REPEAT - Call MC; Call CHECK; UNTIL MC output is correct
  - The running time T is variable, as the number of cycles is variable
  - Using geometric distribution, $E[T] \leq \frac{f(n)+g(n)}{p(n)}$

Outline

Probabilistic algorithms - basic concepts

Random number generators

Random Number Generators

- Random number generation
  - the kernel of Monte Carlo simulation
  - the heart of many standard statistical methods (bootstrap, Bayesian analysis)
  - cryptography
- Random sequence:
  - do not exhibit any discernible pattern,
  - even knowing $x_1, \ldots, x_k$, the next number x_{k+1} can't be predicted
- Need to generate a sequence of independent, identically distributed (from a *known* distribution law) random variables
  - Physical methods and computational methods
  - Dice throwing, coin flipping.
  - Substance undergoing atomic decay : use the comparison of successive length intervals to generate a Bernoulli distribution (http://www.fourmilab.ch/hotbits/)

Criteria for "Good" Random Number Generators

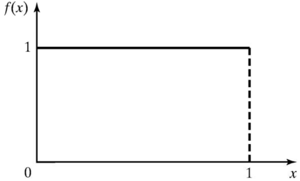
Computational random number generators (RNGs) produce a *deterministic* and *periodic* sequence of numbers - *pseudorandom* numbers

Qualities of generators.

- Long period
- Good distribution of the points (low discrepancy)
- Able to pass some statistical tests
- Speed/efficiency
- Portability - can be implemented easily using different languages and computers
- Repeatability - should be able to generate the same sequence over again

Ideal Random Numbers

1. Independent
2. Uniform distributed on (0,1)
   - Distribution $U_{(0,1)}: f(x) = \begin{cases} 1 & \text{if } 0 \leq x \leq 1 \\ 0 & \text{if not} \end{cases}$



   - $E[U_{(0,1)}] = \frac{1}{2}$

There are techniques to transform a sequence of ideal random numbers into a sequence of random numbers from other distribution of interest.

Generating Random Numbers

- Given a transition function, f, the state at step n is given by

$$x_n = f(x_{n-1}, x_{n-2}, ..., x_{n-k})$$

- k - the *order* of the generator
- $x_1, x_2, ..., x_k$ - the *seed* of the generator
- The length of the sequence prior to beginning to repeat - the *period*
- To get uniform $(0, 1)$ numbers one apply a second transformation

$$u_n = g(x_n)$$

- The output sequence is $\{u_n, n \geq 1\}$

Types of Random Number Generators

- Linear - most commonly used
- Non-linear - structure is less regular than linear generators but more difficult to implement
- Combined - can increase period and improve statistical properties
- Other sources of uniform random numbers (e.g. based on cellular automata or based on chaotic systems)

Linear Congruential Generators

- LCG are defined by

$$\begin{aligned} x_n &= (ax_{n-1} + c) \mod m \\ u_n &= x_n/m \end{aligned}$$

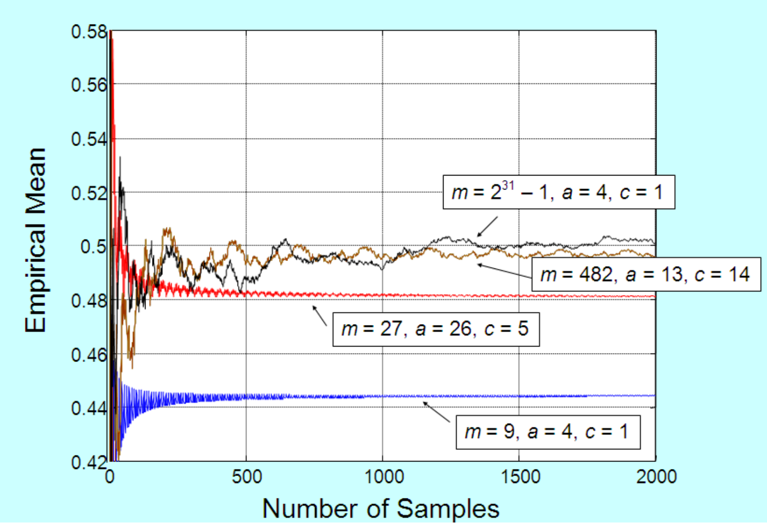
- These are the most widely used and studied random number generators
- The values a (multiplier), c (increment) and m (modulus) should be carefully chosen

$$0 < m, 0 < a < m, 0 \leq c < m$$

$$x_0 < m, x_k \in \{0, 1, ..., m-1\}$$

- Period: maximum m (usually selected as power of 2)

Empirical mean over number of samples



Combining Generators

- Used to increase period length and improve statistical properties
- Shuffling: uses the second generator to choose a random order for the numbers produced by the final generator
- Bit mixing: combines the numbers in the two sequences using some logical or arithmetic operation (addition and subtraction are preferred)

$$
\begin{aligned}
x_i &= 171 x_{i-1} \mod 30269 \\
y_i &= 172 y_{i-1} \mod 30307 \\
z_i &= 170 z_{i-1} \mod 30323 \\
u_i &= \left(\frac{x_i}{30269} + \frac{y_i}{30307} + \frac{z_i}{30323}\right) \mod 1
\end{aligned}
$$

(Wichmann and Hill, 1984; the period of this generator is of order 10^{12})

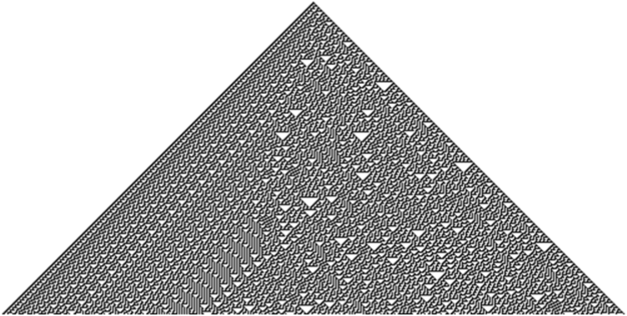
Rule 30

- Rule 30 is a one-dimensional binary cellular automaton rule introduced by Stephen Wolfram in 1983.
- This rule is of particular interest because it produces complex, seemingly-random patterns from simple, well-defined rules.
- In all of Wolfram's elementary cellular automata, an infinite one-dimensional array of cellular automaton cells with only two states is considered, with each cell in some initial state.

current pattern	111	110	101	100	011	010	001	000
new state for center cell	0	0	0	1	1	1	1	0

- Other formulations:
  - $x_{(n+1,i)} = x_{(n,i-1)} \text{ xor } \left[x_{(n,i)} \text{ or } x_{(n,i+1)}\right].$
  - $x_{(n+1,1)} = \left[x_{(n,i-1)} + x_{(n,i)} + x_{(n,i+1)} + x_{(n,i)}x_{(n,i+1)}\right] \mod 2$

Rule 30 cont.



Rule 30 cellular automaton

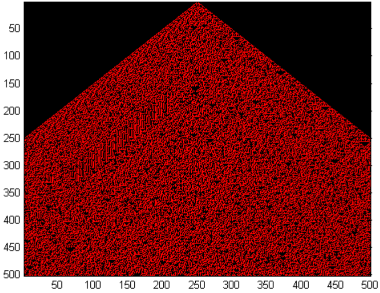
- Rule 30 produces a large number of random patters, depending on initial pattern.
- Stephen Wolfram proposed using its center column as a pseudo-random number generator
- Although Rule 30 produces randomness on many input patterns, there are also an infinite number of input patterns that result in repeating patterns.

Rule 30 cont.

```
function rule30(n, m)
% n = length of cellular automata
% m = number of iterations

    a = zeros(m, n);
    z = zeros(m, n);
    a(1, ceil(n/2)) = 1;
    for i = 2:m
        a(i, :) = generateNext(a(i-1, :));
    end
    image(cat(3,a,z,z));
end

function [r] = generateNext(a)
    n = size(a, 2);
    r(1) = xor(a(n), or(a(1),a(2)));
    r(n) = xor(a(n-1), or(a(n),0));
    for i = 2:n-1
        r(i) = xor(a(i-1), or(a(i), a(i+1)));
    end
end
```



Inverse-Transform Method for Generating Non-U(0,1) Random Numbers

- Let $F(x)$ be the distribution function of X
- Define the inverse function of F by

$$F^{-1}(y) = \inf\{x : F(x) \geq y\}, 0 \leq y \leq 1$$

- Generate X by
$$X = F^{-1}(U)$$
- Example: exponential distribution:

$$F(x) = 1 - e^{-\lambda x}, \text{ where } \lambda, x \geq 0$$

$$X = F^{-1}(U) = -\log(1 - U)/\lambda$$

- Drawback: it is often not possible to evaluate the inverse function $F^{-1}(U)$ in closed form.