KIV/PRO

PDF generated using the open source mwlib toolkit. See http://code.pediapress.com/ for more information.
PDF generated at: Wed, 30 Jan 2013 10:30:59 UTC

Contents

Articles

Randomized algorithm 1

Las Vegas algorithm 8

Monte Carlo algorithm 9

References

Article Sources and Contributors 11

Image Sources, Licenses and Contributors 12

Article Licenses

License 13

Randomized algorithm

A **randomized algorithm** is an algorithm which employs a degree of randomness as part of its logic. The algorithm typically uses uniformly random bits as an auxiliary input to guide its behavior, in the hope of achieving good performance in the "average case" over all possible choices of random bits. Formally, the algorithm's performance will be a random variable determined by the random bits; thus either the running time, or the output (or both) are random variables.

One has to distinguish between algorithms that use the random input to reduce the expected running time or memory usage, but always terminate with a correct result in a bounded amount of time, and **probabilistic algorithms**, which, depending on the random input, have a chance of producing an incorrect result (Monte Carlo algorithms) or fail to produce a result (Las Vegas algorithms) either by signalling a failure or failing to terminate.

In the second case, random performance and random output, the term "algorithm" for a procedure is somewhat questionable. In the case of random output, it is no longer formally effective.[1] However, in some cases, probabilistic algorithms are the only practical means of solving a problem.[2]

In common practice, randomized algorithms are approximated using a pseudorandom number generator in place of a true source of random bits; such an implementation may deviate from the expected theoretical behavior.

Motivation

As a motivating example, consider the problem of finding an ‘*a*’ in an array of *n* elements.

Input: An array of $n \geq 2$ elements, in which half are ‘*a*’s and the other half are ‘*b*’s.

Output: Find an ‘*a*’ in the array.

We give two versions of the algorithm, one Las Vegas algorithm and one Monte Carlo algorithm.

Las Vegas algorithm:

```
findingA_LV(array A, n)
begin
    repeat
        Randomly select one element out of n elements.
    until 'a' is found
end
```

This algorithm succeeds with probability 1. The running time is random (and arbitrarily large) but its expectation is upper-bounded by $O(1)$. (See Big O notation)

Monte Carlo algorithm:

```
findingA_MC(array A, n, k)
begin
    i=1
    repeat
        Randomly select one element out of n elements.
        i = i + 1
    until i=k or 'a' is found
end
```

If an ‘*a*’ is found, the algorithm succeeds, else the algorithm fails. After *k* iterations, the probability of finding an ‘*a*’ is:

$$\Pr[\text{find a}] = 1 - (1/2)^k$$

This algorithm does not guarantee success, but the run time is fixed. The selection is executed exactly *k* times, therefore the runtime is $O(k)$.

Randomized algorithms are particularly useful when faced with a malicious "adversary" or attacker who deliberately tries to feed a bad input to the algorithm (see worst-case complexity and competitive analysis (online algorithm)) such as in the Prisoner's dilemma. It is for this reason that randomness is ubiquitous in cryptography. In cryptographic applications, pseudo-random numbers cannot be used, since the adversary can predict them, making the algorithm effectively deterministic. Therefore either a source of truly random numbers or a cryptographically secure pseudo-random number generator is required. Another area in which randomness is inherent is quantum computing.

In the example above, the Las Vegas algorithm always outputs the correct answer, but its running time is a random variable. The Monte Carlo algorithm (related to the Monte Carlo method for simulation) completes in a fixed amount of time (as a function of the input size), but allow a *small probability of error*. Observe that any Las Vegas algorithm can be converted into a Monte Carlo algorithm (via Markov's inequality), by having it output an arbitrary, possibly incorrect answer if it fails to complete within a specified time. Conversely, if an efficient verification procedure exists to check whether an answer is correct, then a Monte Carlo algorithm can be converted into a Las Vegas algorithm by running the Monte Carlo algorithm repeatedly till a correct answer is obtained.

Computational complexity

Computational complexity theory models randomized algorithms as *probabilistic Turing machines*. Both Las Vegas and Monte Carlo algorithms are considered, and several complexity classes are studied. The most basic randomized complexity class is RP, which is the class of decision problems for which there is an efficient (polynomial time) randomized algorithm (or probabilistic Turing machine) which recognizes NO-instances with absolute certainty and recognizes YES-instances with a probability of at least 1/2. The complement class for RP is co-RP. Problem classes having (possibly nonterminating) algorithms with polynomial time average case running time whose output is always correct are said to be in ZPP.

The class of problems for which both YES and NO-instances are allowed to be identified with some error is called BPP. This class acts as the randomized equivalent of P, i.e. BPP represents the class of efficient randomized algorithms.

History

Historically, the first randomized algorithm was a method developed by Michael O. Rabin for the closest pair problem in computational geometry. The study of randomized algorithms was spurred by the 1977 discovery of a randomized primality test (i.e., determining the primality of a number) by Robert M. Solovay and Volker Strassen. Soon afterwards Michael O. Rabin demonstrated that the 1976 Miller's primality test can be turned into a randomized algorithm. At that time, no practical deterministic algorithm for primality was known.

The Miller-Rabin primality test relies on a binary relation between two positive integers *k* and *n* that can be expressed by saying that *k* "is a witness to the compositeness of" *n*. It can be shown that

- If there is a witness to the compositeness of *n*, then *n* is composite (i.e., *n* is not prime), and
- If *n* is composite then at least three-fourths of the natural numbers less than *n* are witnesses to its compositeness, and
- There is a fast algorithm that, given *k* and *n*, ascertains whether *k* is a witness to the compositeness of *n*.

Observe that this implies that the primality problem is in Co-RP.

If one randomly chooses 100 numbers less than a composite number *n*, then the probability of failing to find such a "witness" is $(1/4)^{100}$ so that for most practical purposes, this is a good primality test. If *n* is big, there may be no

other test that is practical. The probability of error can be reduced to an arbitrary degree by performing enough independent tests.

Therefore, in practice, there is no penalty associated with accepting a small probability of error, since with a little care the probability of error can be made astronomically small. Indeed, even though a deterministic polynomial-time primality test has since been found (see AKS primality test), it has not replaced the older probabilistic tests in cryptographic software nor is it expected to do so for the foreseeable future.

Applications

Quicksort

Quicksort is a familiar, commonly used algorithm in which randomness can be useful. Any deterministic version of this algorithm requires $O(n^2)$ time to sort n numbers for some well-defined class of degenerate inputs (such as an already sorted array), with the specific class of inputs that generate this behavior defined by the protocol for pivot selection. However, if the algorithm selects pivot elements uniformly at random, it has a provably high probability of finishing in $O(n \log n)$ time regardless of the characteristics of the input.

Randomized incremental constructions in geometry

In computational geometry, a standard technique to build a structure like a convex hull or Delaunay triangulation is to randomly permute the input points and then insert them one by one into the existing structure. The randomization ensures that the expected number of changes to the structure caused by an insertion is small, and so the expected running time of the algorithm can be upper bounded. This technique is known as randomized incremental construction.[3]

Verifying matrix multiplication

Input: Matrix $A \in R^{m \times p}$, $B \in R^{p \times n}$, and $C \in R^{m \times n}$.

Output: True if $C = A \cdot B$; false if $C \neq A \cdot B$

We give a Monte Carlo algorithm to solve the problem.[4]

```
begin
  i=1
  repeat
      Choose r=(r_1,...,r_n) ∈{0,1}^n at random.
      Compute C · r and A · (B · r)
      if C · r ≠A · (B · r)
          return FALSE
      endif
      i = i + 1
  until i=k
  return TRUE
end
```

The running time of the algorithm is $O(kn^2)$.

Theorem: The algorithm is correct with probability at least $1-(\frac{1}{2})^k$.

We will prove that if $A \cdot B \neq C$ then $Pr[A \cdot B \cdot r = C \cdot r] \leq 1/2$.

If $A \cdot B \neq C$, by definition we have $D = A \cdot B - C \neq 0$. Without loss of generality, we assume that $d_{11} \neq 0$.

On the other hand, $Pr[A \cdot B \cdot r = C \cdot r] = Pr[(A \cdot B - C) \cdot r = 0] = Pr[D \cdot r = 0]$.

If $D \cdot r = 0$, then the first entry of $D \cdot r$ is 0, that is

$$\sum_{j=1}^{n} d_{1j} r_j = 0$$

Since $d_{11} \neq 0$, we can solve for r_1:

$$r_1 = \frac{-\sum_{j=2}^{n} d_{1j} r_j}{d_{11}}$$

If we fix all r_j except r_1, the equality holds for at most one of the two choices for $r_1 \in \{0, 1\}$. Therefore, $Pr[ABr = Cr] \leq 1/2$.

We run the loop for *k* times. If $C = A \cdot B$, the algorithm is always correct; if $C \neq A \cdot B$, the probability of getting the correct answer is at least $1 - (\frac{1}{2})^k$.

Min cut

Input: A graph *G*(*V*,*E*)

Output: A cut partitioning the vertices into *L* and *R*, with the minimum number of edges between *L* and *R*.

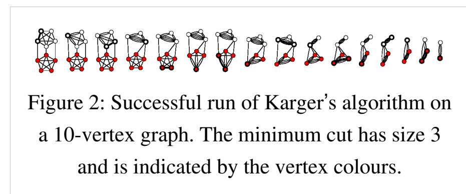
Figure 2: Successful run of Karger's algorithm on a 10-vertex graph. The minimum cut has size 3 and is indicated by the vertex colours.

Recall that the contraction of two nodes, *u* and *v*, in a (multi-)graph yields a new node *u* ' with edges that are the union of the edges incident on either *u* or *v*, except from any edge(s) connecting *u* and *v*. Figure 1 gives an example of contraction of vertex *A* and *B*. After contraction, the resulting graph may have parallel edges, but contains no self loops.

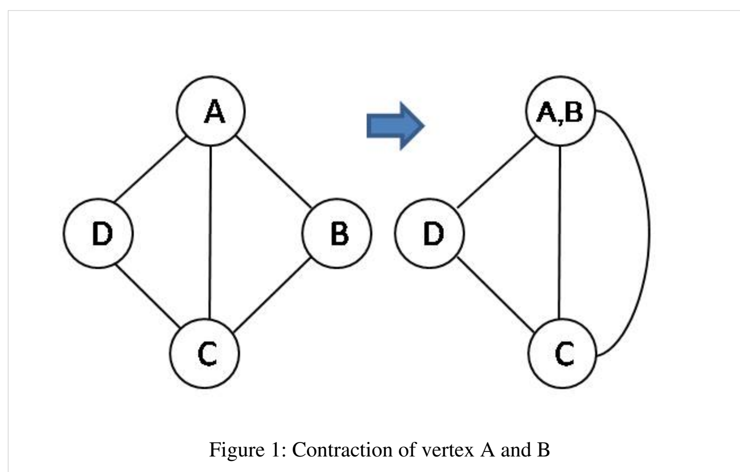

Figure 1: Contraction of vertex A and B

Karger's [5] basic algorithm:

```
begin
  i=1
  repeat
    repeat
       Take a random edge (u,v)∈E in G
       replace u and v with the contraction u'
    until only 2 nodes remain
    obtain the corresponding cut result C_i
```

```
    i=i+1
  until i=m
  output the minimum cut among C_1,C_2,...,C_m.
end
```

In each execution of the outer loop, the algorithm repeats the inner loop until only 2 nodes remain, the corresponding cut is obtained. The run time of one execution is $O(n)$, and n denotes the number of vertices. After m times executions of the outer loop, we output the minimum cut among all the results. The figure 2 gives an example of one execution of the algorithm. After execution, we get a cut of size 3.

Lemma 1: Let k be the min cut size, and let $C = \{e_1,e_2,...,e_k\}$ be the min cut. If, during iteration i, no edge $e \in C$ is selected for contraction, then $C_i = C$.

Proof: If G is not connected, then G can be partitioned into L and R without any edge between them. So the min cut in a disconnected graph is 0. Now, assume G is connected. Let $V=L\cup R$ be the partition of V induced by C : $C=\{ \{u,v\} \in E : u \in L, v \in R \}$ (well-defined since G is connected). Consider an edge $\{u,v\}$ of C. Initially, u,v are distinct vertices. *As long as we pick an edge $f \neq e$, u and v do not get merged.* Thus, at the end of the algorithm, we have two compound nodes covering the entire graph, one consisting of the vertices of L and the other consisting of the vertices of R. As in figure 2, the size of min cut is 1, and $C = \{(A,B)\}$. If we don't select (A,B) for contraction, we can get the min cut.

Lemma 2: If G is a multigraph with p vertices and whose min cut has size k, then G has at least $pk/2$ edges.

Proof: Because the min cut is k, every vertex v must satisfy degree(v) $\geq k$. Therefore, the sum of the degree is at least pk. But it is well known that the sum of vertex degrees equals $2|E|$. The lemma follows.

Analysis of algorithm

The probability that the algorithm succeeds is 1 – the probability that all attempts fail. By independence, the probability that all attempts fail is

$$\prod_{i=1}^{m} \Pr(C_i \neq C) = \prod_{i=1}^{m} (1 - \Pr(C_i = C)).$$

By lemma 1, the probability that $C_i = C$ is the probability that no edge of C is selected during iteration i. Consider the inner loop and let G_j denote the graph after j edge contractions, where $j \in \{0,1,...,n-3\}$. G_j has $n-j$ vertices. We use the chain rule of conditional possibilities. The probability that the edge chosen at iteration j is not in C, given that no edge of C has been chosen before, is $1 - \frac{k}{|E(G_j)|}$. Note that G_j still has min cut of size k, so by Lemma 2, it still has at least $\frac{(n-j)k}{2}$ edges.

Thus, $1 - \frac{k}{|E(G_j)|} \geq 1 - \frac{2}{n-j} = \frac{n-j-2}{n-j}$.

So by the chain rule, the probability of finding the min cut C is $Pr[C_i = C] \geq (\frac{n-2}{n})(\frac{n-3}{n-1})(\frac{n-4}{n-2}) \ldots (\frac{3}{5})(\frac{2}{4})(\frac{1}{3})$.

Cancellation gives $\Pr[C_i = C] \geq \frac{2}{n(n-1)}$. Thus the probability that the algorithm succeeds is at least $1 - (1 - \frac{2}{n(n-1)})^m$. For $m = \frac{n(n-1)}{2} \ln n$, this is equivalent to $1 - \frac{1}{n}$. The algorithm finds the min cut with probability $1 - \frac{1}{n}$, in time $O(mn) = O(n^3 \log n)$.

Derandomization

Randomness can be viewed as a resource, like space and time. Derandomization is then the process of *removing* randomness (or using as little of it as possible). From the viewpoint of computational complexity, derandomizing an efficient randomized algorithm is the question, is P = BPP ?

There are also specific methods that can be employed to derandomize particular randomized algorithms:

- the method of conditional probabilities, and its generalization, pessimistic estimators
- discrepancy theory (which is used to derandomize geometric algorithms)
- the exploitation of limited independence in the random variables used by the algorithm, such as the pairwise independence used in universal hashing
- the use of expander graphs (or dispersers in general) to *amplify* a limited amount of initial randomness (this last approach is also referred to as generating pseudorandom bits from a random source, and leads to the related topic of pseudorandomness)

Where randomness helps

When the model of computation is restricted to Turing machines, it is currently an open question whether the ability to make random choices allows some problems to be solved in polynomial time that cannot be solved in polynomial time without this ability; this is the question of whether P = BPP. However, in other contexts, there are specific examples of problems where randomization yields strict improvements.

- Based on the initial motivating example: given an exponentially long string of 2^k characters, half a's and half b's, a random access machine requires at least 2^{k-1} lookups in the worst-case to find the index of an *a*; if it is permitted to make random choices, it can solve this problem in an expected polynomial number of lookups.
- In communication complexity, the equality of two strings can be verified using $\log n$ bits of communication with a randomized protocol. Any deterministic protocol requires $\Theta(n)$ bits.
- The volume of a convex body can be estimated by a randomized algorithm to arbitrary precision in polynomial time.[6] Bárány and Füredi showed that no deterministic algorithm can do the same.[7] This is true unconditionally, i.e. without relying on any complexity-theoretic assumptions.
- A more complexity-theoretic example of a place where randomness appears to help is the class IP. IP consists of all languages that can be accepted (with high probability) by a polynomially long interaction between an all-powerful prover and a verifier that implements a BPP algorithm. IP = PSPACE.[8] However, if it is required that the verifier be deterministic, then IP = NP.
- In a chemical reaction network (a finite set of reactions like A+B → 2C + D operating on a finite number of molecules), the ability to ever reach a given target state from an initial state is decidable, while even approximating the probability of ever reaching a given target state (using the standard concentration-based probability for which reaction will occur next) is undecidable. More specifically, a Turing machine can be simulated with arbitrarily high probability of running correctly for all time, only if a random chemical reaction network is used. With a simple nondeterministic chemical reaction network (any possible reaction can happen next), the computational power is limited to primitive recursive functions.
- The inherent randomness of algorithms such as Hyper-encryption, Bayesian networks, Random neural networks and Probabilistic Cellular Automata was harnessed by Krishna Palem et al. to design highly efficient hardware systems using Probabilistic CMOS or PCMOS technology that were shown to achieve gains that are as high as a multiplicative factor of 560 when compared to a competing energy-efficient CMOS based realizations.[9]

Notes

[1] "Probabilistic algorithms should not be mistaken with methods (which I refuse to call algorithms), which produce a result which has a high probability of being correct. It is essential that an algorithm produces correct results (discounting human or computer errors), even if this happens after a very long time." Henri Cohen (2000). *A Course in Computational Algebraic Number Theory*. Springer-Verlag, p. 2.

[2] "In testing primality of very large numbers chosen at random, the chance of stumbling upon a value that fools the Fermat test is less than the chance that cosmic radiation will cause the computer to make an error in carrying out a 'correct' algorithm. Considering an algorithm to be inadequate for the first reason but not for the second illustrates the difference between mathematics and engineering." Hal Abelson and Gerald J. Sussman (1996). *Structure and Interpretation of Computer Programs*. MIT Press, section 1.2 (http://mitpress.mit.edu/sicp/full-text/book/book-Z-H-11.html#footnote_Temp_80).

[3] Seidel R. Backwards Analysis of Randomized Geometric Algorithms (http://www.cs.berkeley.edu/~jrs/meshpapers/Seidel.ps.gz).

[4] Michael Mitzenmacher, Eli Upfal. Probability and Computing:Randomized Algorithms and Probabilistic Analysis, April 2005. Cambridge University Press

[5] A. A. Tsay, W. S. Lovejoy, David R. Karger, Random Sampling in Cut, Flow, and Network Design Problems, Mathematics of Operations Research, 24(2):383–413, 1999.

[6] Dyer, M.; Frieze, A.; Kannan, R. (1991), "A random polynomial-time algorithm for approximating the volume of convex bodies" (http://portal.acm.org/citation.cfm?id=102783), *Journal of the ACM* **38** (1): 1–17, doi:10.1145/102782.102783,

[7] Füredi, Z.; Bárány, I. (1986), "Computing the volume is difficult", *Proc. 18th ACM Symposium on Theory of Computing (Berkeley, California, May 28–30, 1986)*, New York, NY: ACM, pp. 442–447, doi:10.1145/12130.12176, ISBN 0-89791-193-8

[8] Shamir, A. (1992), "IP = PSPACE", *Journal of the ACM* **39** (4): 869–877, doi:10.1145/146585.146609

[9] Lakshmi N. Chakrapani, Bilge E. S. Akgul, Suresh Cheemalavagu, Pinar Korkmaz, Krishna V. Palem and Balasubramanian Seshasayee. "Ultra Efficient Embedded SOC Architectures based on Probabilistic CMOS (PCMOS) Technology" (http://www.pubzone.org/dblp/conf/date/ChakrapaniACKPS06). Design Automation and Test in Europe Conference (DATE), 2006. .

References

- Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*, Second Edition. MIT Press and McGraw–Hill, 1990. ISBN 0-262-03293-7. Chapter 5: Probabilistic Analysis and Randomized Algorithms, pp. 91–122.
- Jon Kleinberg and Éva Tardos. *Algorithm Design*. Chapter 13: "Randomized algorithms".
- Don Fallis. 2000. "The Reliability of Randomized Algorithms." (http://dx.doi.org/10.1093/bjps/51.2.255) *British Journal for the Philosophy of Science* 51:255–71.
- M. Mitzenmacher and E. Upfal. Probability and Computing : Randomized Algorithms and Probabilistic Analysis. Cambridge University Press, New York (NY), 2005.
- Rajeev Motwani and P. Raghavan. Randomized Algorithms. Cambridge University Press, New York (NY), 1995.
- Rajeev Motwani and P. Raghavan. Randomized Algorithms. (http://portal.acm.org/citation.cfm?id=234313.234327) A survey on Randomized Algorithms.
- Christos Papadimitriou (1993), *Computational Complexity* (1st ed.), Addison Wesley, ISBN 0-201-53082-1 Chapter 11: Randomized computation, pp. 241–278.
- M. O. Rabin. (1980), "Probabilistic Algorithm for Testing Primality." *Journal of Number Theory* 12:128–38.
- A. A. Tsay, W. S. Lovejoy, David R. Karger, Random Sampling in Cut, Flow, and Network Design Problems, Mathematics of Operations Research, 24(2):383–413, 1999.

Las Vegas algorithm

In computing, a **Las Vegas algorithm** is a randomized algorithm that always gives correct results; that is, it always produces the correct result or it informs about the failure. In other words, a Las Vegas algorithm does not gamble with the verity of the result; it gambles only with the resources used for the computation. A simple example is randomized quicksort, where the pivot is chosen randomly, but the result is always sorted. The usual definition of a Las Vegas algorithm includes the restriction that the *expected* run time always be finite, when the expectation is carried out over the space of random information, or entropy, used in the algorithm.

Las Vegas algorithms were introduced by László Babai in 1979, in the context of the graph isomorphism problem, as a stronger version of Monte Carlo algorithms.[1][2][3] Las Vegas algorithms can be used in situations where the number of possible solutions is relatively limited, and where verifying the correctness of a candidate solution is relatively easy while actually calculating the solution is complex.

The name refers to the city of Las Vegas, Nevada, which is well known within the United States as an icon of gambling.

Complexity class

The complexity class of decision problems that have Las Vegas algorithms with expected polynomial runtime is **ZPP**.

It turns out that

$$\mathrm{ZPP} = \mathrm{RP} \cap \mathrm{co\text{-}RP},$$

which is intimately connected with the way Las Vegas algorithms are sometimes constructed. Namely the class **RP** consists of all decision problems for which a randomized polynomial-time algorithm exists that always answers correctly when the correct answer is "no", but is allowed to be wrong with a certain probability bounded away from one when the answer is "yes". When such an algorithm exists for both a problem and its complement (with the answers "yes" and "no" swapped), the two algorithms can be run simultaneously and repeatedly: a few steps of each, taking turns, until one of them returns a definitive answer. This is the standard way to construct a Las Vegas algorithm that runs in expected polynomial time. Note that in general there is no worst case upper bound on the run time of a Las Vegas algorithm.

Relation to Monte Carlo algorithms

Las Vegas algorithms can be contrasted with Monte Carlo algorithms, in which the resources used are bounded but the answer is not guaranteed to be correct 100% of the time. By an application of Markov's inequality, a Las Vegas algorithm can be converted into a Monte Carlo algorithm via early termination.

Notes

[1] László Babai, Monte-Carlo algorithms in graph isomorphism testing (http://people.cs.uchicago.edu/~laci/lasvegas79.pdf), Université de Montréal, D.M.S. No. 79-10.

[2] Leonid Levin, The Tale of One-way Functions (http://arxiv.org/abs/cs.CR/0012023), *Problems of Information Transmission*, vol. 39 (2003), 92-103.

[3] Dan Grundy, Concepts and Calculation in Cryptography (http://www.cs.kent.ac.uk/people/staff/eab2/crypto/thesis.web.pdf), University of Kent, Ph.D. thesis, 2008

References

- *Algorithms and Theory of Computation Handbook*, CRC Press LLC, 1999, "Las Vegas algorithm", in *Dictionary of Algorithms and Data Structures* [online], Paul E. Black, ed., U.S. National Institute of Standards and Technology. 17 July 2006. (accessed May 09, 2009) Available from: (http://www.nist.gov/dads/HTML/lasVegas.html)

Monte Carlo algorithm

In computing, a **Monte Carlo algorithm** is a randomized algorithm whose running time is deterministic, but whose output may be incorrect with a certain (typically small) probability.

The related class of Las Vegas algorithms is also randomized, but in a different way: they take an amount of time that varies randomly, but always produce the correct answer. A Monte Carlo algorithm can be converted into a Las Vegas algorithm whenever there exists a procedure to verify that the output produced by the algorithm is indeed correct. If so, then the resulting Las Vegas algorithm is merely to repeatedly run the Monte Carlo algorithm until one of the runs produces an output that can be verified to be correct.

The name refers to the grand casino in the Principality of Monaco at Monte Carlo, which is well-known around the world as an icon of gambling.

One-sided vs two-sided error

Whereas the answer returned by a deterministic algorithm is always expected to be correct, this is not the case for Monte Carlo algorithms. For decision problems, these algorithms are generally classified as either **false**-biased or **true**-biased. A **false**-biased Monte Carlo algorithm is always correct when it returns **false**; a **true**-biased algorithm is always correct when it returns **true**. While this describes algorithms with *one-sided errors*, others might have no bias; these are said to have *two-sided errors*. The answer they provide (either **true** or **false**) will be incorrect, or correct, with some bounded probability.

For instance, the Solovay–Strassen primality test is used to determine whether a given number is a prime number. It always answers **true** for prime number inputs; for composite inputs, it answers **false** with probability at least 1/2 and **true** with probability at most 1/2. Thus, **false** answers from the algorithm are certain to be correct, whereas the **true** answers remain uncertain; this is said to be a *(1/2)-correct false-biased algorithm*.

Amplification

For a Monte Carlo algorithm with one-sided errors, the failure probability can be reduced (and the success probability amplified) by running the algorithm k times. Consider again the Solovay–Strassen algorithm which is *(1/2)-correct false-biased*. One may run this algorithm multiple times returning a **false** answer if it reaches a **false** response within k iteration, and otherwise returning **true**. Thus, if number is prime then the answer is always correct, and if the number is composite then the answer is correct with probability at least $1-(1-1/2)^k = 1-2^{-k}$.

For Monte Carlo decision algorithms with two-sided error, the failure probability may again be reduced by running the algorithm k times and returning the majority function of the answers.

Complexity classes

The complexity class BPP describes decision problems that can be solved by polynomial-time Monte Carlo algorithms with a bounded probability of two-sided errors, and the complexity class RP describes problems that can be solved by a Monte Carlo algorithm with a bounded probability of one-sided error: if the correct answer is no, the algorithm always says so, but it may answer no incorrectly for some instances where the correct answer is yes. In contrast, the complexity class ZPP describes problems solvable by polynomial expected time Las Vegas algorithms. ZPP ⊂ RP ⊂ BPP, but it is not known whether any of these complexity classes is distinct from each other; that is, Monte Carlo algorithms may have more computational power than Las Vegas algorithms, but this has not been proven. Another complexity class, PP, describes decision problems with a polynomial-time Monte Carlo algorithm that is more accurate than flipping a coin but where the error probability cannot be bounded away from 1/2.

Applications in computational number theory

Well-known Monte Carlo algorithms include the Solovay–Strassen primality test, the Miller–Rabin primality test, and certain fast variants of the Schreier–Sims algorithm in computational group theory.

References

- Motwani, Rajeev; Raghavan, Prabhakar (1995). *Randomized Algorithms*. New York: Cambridge University Press. ISBN 0-521-47465-5.
- Cormen, Thomas H.; Leiserson, Charles E.; Rivest, Ronald L.; Stein, Clifford (2001). "Ch 5. Probabilistic Analysis and Randomized Algorithms". *Introduction to Algorithms* (2nd ed.). Boston: MIT Press and McGraw-Hill. ISBN 0-262-53196-8.
- Berman, Kenneth A; Paul, Jerome L. (2005). "Ch 24. Probabilistic and Randomized Algorithms". *Algorithms: Sequential, Parallel, and Distributed*. Boston: Course Technology. ISBN 0-534-42057-5.

Article Sources and Contributors

Randomized algorithm *Source*: http://en.wikipedia.org/w/index.php?oldid=533506990 *Contributors*: Altenmann, Andris, Andrés Agustín Baldrich, Angela, Arvindn, Avinash.lingamneni, Bfinn, Bkell, Boffob, Booyabazooka, CRGreathouse, Cassowary, Centrx, Chad.netzer, Chmod007, ClamDip, Constructive editor, Courcelles, Cybercobra, David Eppstein, Dax5, Dcoetzee, Electron9, Flammifer, GPHemsley, Giftlite, Gothmog.es, GregorB, Gwern, Harold f, Jamelan, Jenny Lam, John254, Justin W Smith, Jxfeng, Karl-Henner, Kku, KoenDelaere, Kope, Ledt.uoft, Magioladitis, Maksim-e, Mandarax, Melcombe, Michael Hardy, Miym, Nikai, Oleg Alexandrov, Optim, Ott, PierreYvesLouis, Pinguin.tk, Piyush Sriva, Prosfilaes, Qwertyus, RiskAverse, Rjwilmsi, Roll-Morton, RoyBoy, RunOrDie, Ruud Koot, SPhotographer, Schizoid, That Guy, From That Show!, The Anome, TheProject, Thore Husfeldt, Tomash, Tomchiukc, Ultimus, Wavelength, Yaronf, Zmoboros, 59 anonymous edits

Las Vegas algorithm *Source*: http://en.wikipedia.org/w/index.php?oldid=532811379 *Contributors*: Adanner, Bender2k14, Booyabazooka, CesarB, Charles Matthews, Chenopodiaceous, Dcoetzee, FiP, Froth, GIrving, Giftlite, Gunsfornuns, Hermel, Jamelan, Jbp.mccabe, Jitse Niesen, Justin W Smith, Mhym, Michael Hardy, Pinguin.tk, RobinK, Root4(one), RuED, Rufus210, Ruud Koot, Schizoid, Stimpy, Superm401, Tobias Bergemann, TubularWorld, Vegaswikian, ZeroOne, 38 anonymous edits

Monte Carlo algorithm *Source*: http://en.wikipedia.org/w/index.php?oldid=511713883 *Contributors*: CBM, Charles Matthews, David Eppstein, Dcoetzee, Divide, Dylanwhs, Electron9, Giftlite, Justin W Smith, LOL, Max Libbrecht, Melcombe, Michael Hardy, NTox, Ndickson, Pcap, Pushpendera, Ruud Koot, Sankyo68, Schizoid, Shacharg, Zfeinst, 7 anonymous edits

Image Sources, Licenses and Contributors

File:Single run of Karger’s Mincut algorithm.svg *Source*: http://en.wikipedia.org/w/index.php?title=File:Single_run_of_Karger's_Mincut_algorithm.svg *License*: Creative Commons Attribution-Sharealike 3.0 *Contributors*: Thore Husfeldt

File:Contraction vertices.jpg *Source*: http://en.wikipedia.org/w/index.php?title=File:Contraction_vertices.jpg *License*: Creative Commons Attribution-Sharealike 3.0 *Contributors*: Jxfeng

License

Creative Commons Attribution-Share Alike 3.0 Unported
//creativecommons.org/licenses/by-sa/3.0/