Probabilistic (Randomized) algorithms

Idea:

Build algorithms using a ‘random’ element so as gain improved performance. For some cases, improved performance is very dramatic, moving from intractable to tractable.

Often however, there is some loss in the reliability of results: either no result at all may be produced, or an incorrect result may be returned, or, for numerical results, an approximate answer may be produced.

Because of the random element, different runs may produce different results and therefore the reliability of results may be improved with multiple runs.

References

Algorithmics: Theory and Practice, G. Brassard and P. Bratley. Prentice-Hall. 1988. Introductory chapter on the topic.

Randomized Algorithms, R. Motwani and P. Raghavan. Cambridge University Press, 1995. Very good and detailed development of the subject.

Algorithms in Java, R. Sedgewick, Addison Wesley. Chapter on random number generators.

Classification of probabilistic algorithms

There is a variety of behaviours associated with probabilistic algorithms:

- Algorithms which always return a result, but the result may not always be correct. We attempt to minimise the probability of an incorrect result, and using the random element, multiple runs of the algorithm will reduce the probability of incorrect results.

  These are sometimes called **Monte Carlo** algorithms. Terminology has not been fixed, and varies with author.

- Algorithms that never return an incorrect result, but may not produce results at all on some runs. Again, we wish to minimise the probability of no result, and, because of the random element, multiple runs will reduce the probability of no result.

  These are sometimes called **Las Vegas** algorithms.

Classification (cont.)

- Algorithms which always return a result and the correct result, but where a random element increases the efficiency, by avoiding or reducing the probability of worst-case behaviour. Useful for algorithms which have a poor worst-case behaviour but a good average-case behaviour.

  These are sometimes called **Sherwood** algorithms, *cf.* Robin Hood!

- **Numerical approximation algorithms**. Here the random element allows us to get approximate numerical results, often much faster than direct methods, and multiple runs will provide increasing approximation.

Random numbers

The random element required by probabilistic algorithms is provided by a *random number generator*. Of course, these do not generate numbers ‘at random’ (only a truly random process, eg radioactive decay, could do this!).

Random number generators

Random number generators have been devised to generate sequences of numbers which behave as though they were random, in that they pass various ‘tests of randomness’. Such sequences are sometimes called ‘pseudo-random’.

A variety of methods have been proposed. These all generate numbers in a specific range, and require an initial element, called *the seed*, to begin the generation. Each number in the sequence is computed from its predecessor.

These methods have several disadvantages: Because numbers are generated from predecessors, if a number re-occurs, then the sequence cycles repeatedly. It is intended that different choices generate different pseudo-random sequences, but if a choice coincides with an entry in any sequence it will generated the remainder of this sequence.

The analysis of the randomness and reliability of these pseudo-random sequences is, in general, very difficult.

Example: Middle squares method

Consider 4-digit numbers.

Repeat the following: Square the number and take the middle 4 digits of the square.

For example, begin with 1234, square is 1522756. Take 5227 as next entry, square is 27321529. Next number is 3215.

Sequence is 1234, 5227, 3215, 3362, 3030, 1809, 2728,...

This is not a reliable method: for some seeds it will repeat itself fairly quickly, and the selection of numbers may not be very uniform in the interval. Zeros tend to propagate.

Linear congruence methods

Linear congruence methods are widely used. They require the choice of two numbers m (which is the divisor for the congruence), and b (which allows us to generate new instances).

Then for any seed a, we generate the next number in the sequence as

$$a \times b + 1 \mod m.$$

The choice of m and b is critical, m should be large and there are rules for choosing a suitable b.

For example, take $m = 1000$, $b = 921$ and the seed $a = 123$, then we generate the sequence:

$$123, 284, 405, 6, 527, 368, \ldots$$

Again, the analysis of the randomness of these sequences is difficult.

Other methods have been used to generate random numbers, including ‘additive congruence’ methods (see references).

Probabilistic algorithms: ‘Monte Carlo’ methods

Recall: The following behaviour we called Monte Carlo:

> Algorithms which always return a result, but the result may not always be correct. We attempt to minimise the probability of an incorrect result, and using the random element, multiple runs of the algorithm will reduce the probability of incorrect results.

Majority element

An array A of N natural numbers has a *majority element* n, if the number of occurrences of n in A is greater than $N/2$.

Question: How would you decide whether an array has a majority element using a deterministic algorithm, and what is its complexity?

By choosing an element *at random* and testing to see if it is a majority element we get a probabilistic algorithm:

```
Majority(A,N) =
 {  i = random(0..N-1)
    x = A[i}
    k = 0
    for j from 0 to N-1 do
      {if A[j] = x then k = k+1}
    return (k > N/2) }
```

Here, `random(0..N-1)` is a uniform random selection of integers between 0 and N-1.

Notice that this algorithm *returns true exactly when it has selected a majority element, but that it may return false even if there is a majority element.*

However, if there is a majority element, then the probability of returning false is less than 1/2, since a majority element occupies more than half the array.

We call such an algorithm *true-biased, and 1/2-correct.*

Because of the random element, repeatedly running this algorithm will increase our confidence in the result: If it ever returns true, then there is a majority element. If after M runs every result is false, the probability of having a majority element is less than $1/2^M$.

Matrix multiplication

Here is another example of a Monte Carlo algorithm:

Recall that the standard way of multiplying $N \times N$ matrices A and B to get result C is

$$C[i,k] = \sum_{j \in \{1,\ldots,N\}} A[i,j] \times B[j,k]$$

This is $O(N^3)$. Strassen's algorithm is an $O(N^{2.81})$ variation of this. Further algorithms of increasing complication and computational overheads have been proposed with complexities down to $O(N^{2.376})$.

Here is a Monte Carlo method of testing whether a matrix C is the product of A and B with complexity $O(n^2)$.

```
MatrixMultiply(A,B,C,N) =
 { newarray X[1...N]      % a 1-dimensional array
   for i from 1 to N do
     { X[i] = random(-1,1) } % randomly fill X
   return { A(BX) = CX } }
```

Here `random(-1,1)` selects -1 or 1 randomly, and $A(BX)$ is computed as A times BX.

This algorithm is $O(N^2)$. Why?

It returns true whenever $AB = C$. We can show that it returns false with probability at least 1/2 when $AB \neq C$. Again, multiple runs of this algorithm will therefore increase the reliability of the result.

Other examples of the Monte Carlo method

The Monte Carlo technique has been used to provide probabilistic algorithms for a range of applications including:

- Testing the primality of numbers (Rabin, M.O. *Probabilistic algorithm for primality testing*, 1980). Deterministic primality testing is computationally difficult, but believed *not* to be NP-complete.
- Deciding set equality.
- Applications in cryptography.

Probabilistic algorithms: ‘Las Vegas’ methods

Recall that ‘Las Vegas’ algorithms were described as:

> Algorithms that never return an incorrect result, but may not produce results at all on some runs. Again, we wish to minimise the probability of no result, and, because of the random element, multiple runs will reduce the probability of no result.

Las Vegas algorithms may produce tractable computations for tasks for which deterministic algorithms are intractable even on average. However, we cannot guarantee a result and there is no upper bound on the time for a result to appear, but the expected time may in fact be small.

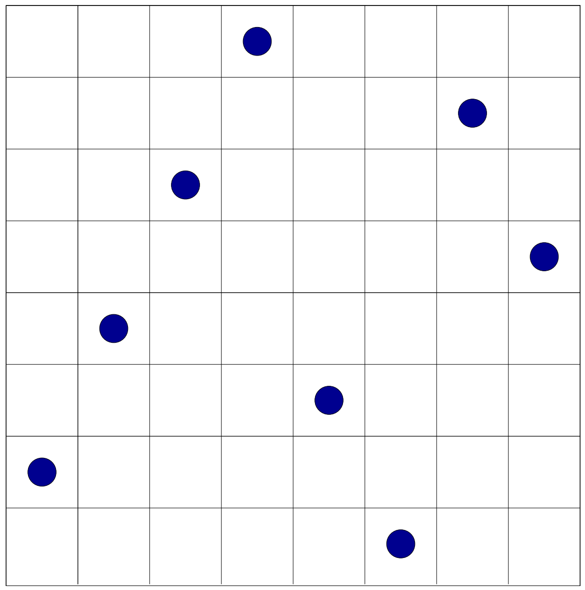
Typically, Las Vegas algorithms are used to explore a space of states some of which are the successful states we seek. Las Vegas algorithms use some random choices to move about the space, rather than computing at each state a new state to move to. This is likely to be successful if the proportion of successful states in the state-space is fairly high, and will lead to an improved efficiency if the computations of next states are difficult or if systematic exploration is not necessary.

However, some random choices of next state reach states where no further moves will lead to a successful state. But we can, of course, run the algorithm again and different choices will be made.

An example of a Las Vegas algorithm

The *Eight Queens Problem* is a well known algorithmic task, where we are asked to place eight queens on a chess board with no queen attacking another. A queen attacks other pieces in the same row, same column and along its diagonals.

One solution (of many):



Algorithmic approaches

The classic solution is via *backtracking*. Place the first queen in the top-left corner. Now with some non-attacking queens in place, place a queen in the next row, if it attacks previous queens move it across the row, square by square. If there is no suitable position in this row then move the queen in the immediately previous row on one space if possible. If neither row leads to a required arrangement, move the queen in the previous row by one space, etc.

In fact this returns a solution after examining only 114 of the 2,057 states.

Here is a Las Vegas solution to the problem:

Suppose that k rows ($0 \leq k \leq 8$) have been successfully occupied (initially $k = 0$). If $k = 8$ then stop with success. If not then we wish to occupy row $k + 1$. Calculate all the possible positions on this row ie those which are not attacked by already placed queens. If there are no such places, then fail. Otherwise *choose one such place at random*, increment k, and repeat.

Notice there is no backtracking: if at any point we cannot place a queen, we fail.

However, we can repeat the run, and because of the random element, we will get a possibly different arrangement.

Question: Is this any better than the deterministic backtracking?

Answer: Surprisingly, YES!

It can be calculated that the probability of success of this random placement is 0.1293... This may seem low, but it means that a solution is obtained 1 time out of 8 simply by guessing! The expected number of states explored is approximately 56, compared to 114 for the deterministic backtracking solution.

The reason for this, is that the solutions of the problem seem to be in some sense ‘unsystematic’ or ‘random’ arrangements.

In fact we can do better, by combining random placement and some backtracking, first fixing some queens at random, and then completing the arrangement by backtracking. If two are placed in the first two rows at random then the expect number of states explored to find a solution is only 29!

Applications of Las Vegas algorithms

The Las Vegas technique has been used to provide probabilistic algorithms for a range of applications including:

- Efficient factorization of integers.
- Breaking symmetry in networks of processes.

Probabilistic algorithms: ‘Sherwood’ methods

Recall that we described ‘Sherwood’ algorithms as:

> Algorithms which always return a result and the correct result, but where a random element increases the efficiency, by avoiding or reducing the probability of worst-case behaviour. This is useful for algorithms which have a poor worst-case behaviour but a good average-case behaviour, and in particular can be used where embedding an algorithm in an application may lead to increased worst-case behaviour.

An example: Quicksort

Quicksort is a sorting algorithm with worst case behaviour $O(N^2)$ but average case behaviour of $O(N \times \log(N))$.

The problem is that we cannot guarantee very uneven splits of the list by choosing pivots. Indeed, systematic choice of pivots (eg the first element), can lead, in certain applications (eg when almost sorted lists tend to be supplied), to worst-case behaviour. This is undesirable!

Can we do better?

It turns out that the $O(N \log(N))$ behaviour is achievable even if the lists are split 1/4:3/4. The key observation is that *there are at least N/2 pivots in the list with this property*! This is not immediately obvious - we need to remove the pivot from the list to be sorted, and also take care over where items equal to the pivot are placed.

How do we find suitable pivots?

We select at random. We can do this by modifying the quicksort algorithm to choose a random pivot, or, if modifying the algorithm is not feasible (eg it is too complicated or we don't have the source code), then use *stochastic preconditioning*, i.e. randomly shuffle the input before supplying it to the algorithm (which will have a deterministic choice of pivot).

These 'Sherwood' techniques are useful in searching, selection, median finding, sorting, hashing and more generally, where worst-case behaviours and average-case behaviours differ and worst-case behaviours arise from systematic choice which randomisation can avoid.

Probabilistic algorithms: Numerical approximation methods

Probabilistic techniques can be used to solve numerical problems by giving approximations to the result. Repeated runs give increased accuracy of the result.

Example: Numerical integration

Consider calculating the integral

$$\int_0^1 f(x)dx.$$

This is the area under the curve f. Suppose that we know that for $x \in [0 \ldots 1]$, $0 \le f(x) \le 1$. Then the following probabilistic technique approximates the solution:

```
integrate(f,N) =
   { k = 0
     for i from 1 to N do
       { x = random(0..1)
         y = random(0..1)
         if y =< f(x) then k = k+1 }
     return k/N }
```

That is, we select points in the (0,1)-square at random and test whether or not they are below the curve. The result is the proportion of those below the curve.

Another probabilistic method is to choose points at random on the x-axis, calculate the value $f(x)$ and take the average of all of these.

These methods usually do not provide a fast convergence to results and, for simple integrals, standard approximation methods (Trapezium method, or Simpson's method) are better. However:

- Occasionally, functions may perform badly on a systematic method, especially if they have multiple periodicities.
- For multiple integrals in higher dimensions, probabilistic techniques can prove useful.

Many other numerical problems in calculus, in estimating sizes of large sets, in linear algebra etc have probabilistic solutions (see the references for details).