Chapter 1

Introduction to Monte Carlo Methods

Monte Carlo methods present a class of computational algorithms that rely on repeated random sampling to approximate some unknown quantities. They are best suited for calculation using a computer program, and they are typically used when the exact results with a deterministic algorithm are not available.

The Monte Carlo method was developed in the 1940s by John von Neumann, Stanislaw Ulam, and Nicholas Metropolis while they were working on the Manhattan Project at the Los Alamos National Laboratory. It was named after the Monte Carlo Casino, a famous casino where Ulam's uncle often gambled away his money.

We mainly deal in this book with two well-known Monte Carlo methods, called *importance sampling* and *splitting*, and in particular with their applications to combinatorial optimization, counting, and estimation of probabilities of rare events.

Importance sampling is a well-known variance reduction technique in stochastic simulation studies. The idea behind importance sampling is that certain values of the input random variables have a greater impact on the output parameters than others. If these "important" values are sampled more frequently, the variance of the output estimator can be reduced. However, such direct use of importance sampling distributions will result in a biased estimator. To eliminate the bias, the simulation outputs must be modified (weighted) by using a likelihood ratio factor, also called the Radon Nikodym derivative [108]. The fundamental issue in implementing importance sampling is the choice of the importance sampling distribution.

In the case of counting problems, it is well known that a straightforward application of importance sampling typically yields poor approximations of the quantity of interest. In particular, Gogate and Dechter [56, 57] show that poorly chosen importance sampling in graphical models such as satisfiability models generates many useless zero-weight samples, which are often rejected, yielding an inefficient sampling process. To address this problem, which is called the problem of

Fast Sequential Monte Carlo Methods for Counting and Optimization, First Edition.
Reuven Y. Rubinstein, Ad Ridder, and Radislav Vaisman.
© 2014 John Wiley & Sons, Inc. Published 2014 by John Wiley & Sons, Inc.

COPYRIGHTED MATERIAL

losing trajectories, these authors propose a clever sample search method, which is integrated into the importance sampling framework.

With regard to probability problems, a wide range of applications of importance sampling have been reported successfully in the literature over the last decades. Siegmund [115] was the first to argue that, using an exponential change of measure, asymptotically efficient importance sampling schemes can be built for estimating gambler's ruin probabilities. His analysis is related to the theory of large deviations, which has since become an important tool for the design of efficient Monte Carlo experiments. Importance sampling is now a subject of almost any standard book on Monte Carlo simulation (see, for example, [3, 108]). We shall use importance sampling widely in this book, especially in connection to rare-event estimation.

The splitting method dates back to Kahn and Harris [62] and Rosenbluth and Rosenbluth [97]. The main idea is to partition the state-space of a system into a series of nested subsets and to consider the rare event as the intersection of a nested sequence of events. When a given subset is entered by a sample trajectory during the simulation, numerous random retrials are generated, with the initial state for each retrial being the state of the system at the entry point. By doing so, the system trajectory is split into a number of new subtrajectories, hence the name "splitting". Since then, hundreds of papers have been written on this topic, both from a theoretical and a practical point of view. Applications of the splitting method arise in particle transmission (Kahn and Harris [62]), queueing systems (Garvels [48], Garvels and Kroese [49], Garvels et al. [50]), and reliability (L'Ecuyer et al. [76]). The method has been given new impetus by the RESTART (Repetitive Simulation Trials After Reaching Thresholds) method in the sequence of papers by Villén-Altimirano and Villén-Altimirano [122–124]. A fundamental theory of the splitting method was developed by Melas [85], Glasserman et al. [54, 55], and Dean and Dupuis [38, 39]. Recent developments include the adaptive selection of the splitting levels in Cérou and Guyader [24], the use of splitting in reliability networks [73, 109], quasi-Monte Carlo estimators in L'Ecuyer et al. [77], and the connection between splitting for Markovian processes and interacting particle methods based on the Feynman-Kac model in Del Moral [89].

Let us introduce the notion of a *randomized algorithm*. A randomized algorithm is an algorithm that employs a degree of randomness as part of its logic to solve a deterministic problem such as a combinatorial optimization problem. As a result, the algorithm's output will be a random variable representing either the running time, its output, or both. In general, introducing randomness may result in an algorithm that is far simpler, more elegant, and sometimes even more efficient than the deterministic counterpart.

EXAMPLE 1.1 ***Checking Matrix Multiplication***

Suppose we are given three $n \times n$ matrices $\boldsymbol{A}$, $\boldsymbol{B}$, and $\boldsymbol{C}$ and we want to check whether $\boldsymbol{AB} = \boldsymbol{C}$.

A trivial deterministic algorithm would be to run a standard multiplication algorithm and compare each entry of $\boldsymbol{AB}$ with $\boldsymbol{C}$. Simple matrix multiplication

requires $\mathcal{O}(n^3)$ operations. A more sophisticated algorithm [88] takes only $\mathcal{O}(n^{2.376})$ operations. Using randomization, however, we need only $\mathcal{O}(n^2)$ operations, with an extremely small probability of error [88].

The randomized procedure is as follows:

- Pick a random n-dimensional vector $\boldsymbol{r} = (r_1, \ldots, r_n)$.
- Multiply both sides of $\boldsymbol{AB} = \boldsymbol{C}$ by $\boldsymbol{r}$, that is, obtain $\boldsymbol{A}(\boldsymbol{Br})$ and $\boldsymbol{Cr}$.
- If $\boldsymbol{A}(\boldsymbol{Br}) = \boldsymbol{Cr}$, then declare $\boldsymbol{AB} = \boldsymbol{C}$, otherwise, $\boldsymbol{AB} \neq \boldsymbol{C}$.

This algorithm runs in $\mathcal{O}(n^2)$ operations because matrix multiplication is associative, so $(\boldsymbol{AB})\boldsymbol{r}$ can be computed as $\boldsymbol{A}(\boldsymbol{Br})$, thus requiring only three matrix-vector multiplications for the algorithm. □

For more examples and foundations on randomized algorithms, see the monographs [88, 90].

We shall consider not only randomized algorithms but also random structures. The latter comprises random graphs (such as Erdös-Rényi graphs), random Boolean formulas, and so on. Random structures are of interest both as a means of understanding the behavior of algorithms on typical inputs and as a mathematical framework in which one can investigate various probabilistic techniques to analyze randomized algorithms.

This book deals with Monte Carlo methods and their associated randomized algorithms for solving combinatorial optimization and counting problems. In particular, we consider combinatorial problems that can be modeled by integer linear constraints. To clarify, denote by $\mathcal{X}^*$ the set of feasible solutions of a combinatorial problem, which is assumed to be a subset of an n-dimensional integer vector space and which is given by the following linear constraints:

$$\mathcal{X}^* = \begin{cases} \sum_{j=1}^n a_{ij} x_j = b_i, & \text{for all } i = 1, \ldots, m_1 \\ \sum_{j=1}^n a_{ij} x_j \geq b_i, & \text{for all } i = m_1 + 1, \ldots, m_1 + m_2 = m \\ \boldsymbol{x} = (x_1, \ldots, x_n) \in \mathbb{Z}^n. \end{cases} \tag{1.1}$$

Here, $\boldsymbol{A} = (a_{ij})$ is a given $m \times n$ matrix and $\boldsymbol{b} = (b_i)$ is a given m-dimensional vector. Most often we require the variables x_j to be nonnegative integers and, in particular, binary integers.

In this book, we describe in detail various problems, algorithms, and mathematical aspects that are associated with (1.1) and its relation to decision making, counting, and optimization. Below is a short list of problems associated with (1.1):

1. *Decision making*: Is $\mathcal{X}^*$ nonempty?
2. *Optimization*: Solve $\max_{\boldsymbol{x} \in \mathcal{X}^*} S(\boldsymbol{x})$ for a given objective (performance) function $S : \mathcal{X}^* \to \mathbb{R}$.
3. *Counting*: Calculate the cardinality $|\mathcal{X}^*|$ of $\mathcal{X}^*$.

It turns out that, typically, it is hard to solve any of the above three problems and, in particular, the counting one, which is the hardest one. However, we would like to point out that there are problems for which decision making is easy (polynomial time) but counting is hard [90]. As an example, finding a feasible path (and also the shortest path) between two fixed nodes in a network is easy, whereas counting the total number of paths between the two nodes is difficult. Some other examples of hard counting and easy decision-making problems include:

- How many different variable assignments will satisfy a given satisfiability formula in disjunctive normal form?
- How many different variable assignments will satisfy a given 2-satisfiability formula (constraints on pairs of variables)?
- How many perfect matchings are there for a given bipartite graph?

In Chapter 5, we follow the saying "counting is hard, but decision making is easy" and employ relevant decision-making algorithms, also called oracles, to derive fast Monte Carlo algorithms for counting.

Below is a detailed list of interesting hard counting problems.

- *The Hamiltonian cycle problem.* How many Hamiltonian cycles does a graph have? That is, how many tours contains a graph in which every node is visited exactly once (except for the beginning/end node)?
- *The permanent problem.* Calculate the permanent of a matrix $\boldsymbol{A}$, or equivalently, the number of perfect matchings in a bipartite balanced graph with $\boldsymbol{A}$ as its biadjacency matrix.
- *The self-avoiding walk problem.* How many self-avoiding random walks of length n exist, when we are allowed to move at each grid point in any neighboring direction with equal probability?
- *The connectivity problem.* Given two different nodes in a directed or undirected graph, say v and w, how many paths exist from v to w that do not traverse the same edge more than once?
- *The satisfiability problem.* Let $\mathcal{X}$ be a collection of all sets of n Boolean variables $\{x_1, \ldots, x_n\}$. Thus, $\mathcal{X}$ has cardinality $|\mathcal{X}| = 2^n$. Let $\mathcal{C}$ be a set of m Boolean disjunctive clauses. Examples of such clauses are $C_1 = x_1 \vee \bar{x}_2 \vee x_4$, $C_2 = \bar{x}_2 \vee \bar{x}_3$, etc. How many (if any) satisfying truth assignments for $\mathcal{C}$ exist, that is, how many ways are there to set the variables $x_1, \ldots, x_n$ either true or false so that all clauses $C_i \in \mathcal{C}$ are true?
- *The k-coloring problem.* Given $k \geq 3$ distinct colors, in how many different ways can one color the nodes (or the edges) of a graph, so that each two adjacent nodes (edges, respectively) in the graph have different colors?
- *The spanning tree problem.* How many unlabeled spanning trees has a graph G? Note that this counting problem is easy for labeled graphs.
- *The isomorphism problem.* How many isomorphisms exist between two given graphs G and H? In other words, in an isomorphism problem one needs to

find all mappings ϕ between the nodes of G and H such that (v, w) is an edge of G if and only if $\phi(v)\phi(w)$ is an edge of H.

- *The clique problem.* How many cliques of fixed size k exist in a graph G? Recall that a clique is a complete subgraph of G.

The decision versions of these problems are all examples of NP-complete problems. Clearly, counting all feasible solutions, denoted by #P, is an even harder problem.

Generally, the complexity class #P consists of the counting problems associated with the decision problems in NP. Completeness is defined similarly to the decision problems: a problem is #P-complete if it is in #P, and if every #P problem can be reduced to it in polynomial counting reduction. Hence, the counting problems that we presented above are all #P-complete. For more details we refer the reader to the classic monograph by Papadimitrou and Stieglitz [92].