Optimization Techniques by Building Population-based Probabilistic Models: From Algorithms to Applications

基于种群概率模型的优化技术：从算法到应用

姜 群 ● 著

上海交通大学出版社
SHANGHAI JIAO TONG UNIVERSITY PRESS

Optimization Techniques by Building Population-based Probabilistic Models: From Algorithms to Applications

基于种群概率模型的优化技术：从算法到应用

姜 群 著

上海交通大學出版社

内 容 提 要

本书较系统地讨论了遗传算法和分布估计算法的基本理论，并在二进制搜寻空间实验性地比较了几种分布估计算法。在此基础上深入地论述了构建一类新的分布估计算法的思路和实现方法，最后介绍了分布估计算法在计算机科学、资源管理等领域的一些成功应用实例及分布估计算法的几种有效改进方法。

图书在版编目(CIP)数据

基于种群概率模型的优化技术：从算法到应用：英文/姜群著. —上海：上海交通大学出版社，2010
ISBN 978-7-313-06369-4

Ⅰ.①基… Ⅱ.①姜… Ⅲ.①电子计算机—算法理论—英文 Ⅳ.①TP301.6

中国版本图书馆 CIP 数据核字(2010)第 056262 号

基于种群概率模型的优化技术：从算法到应用

姜 群 著

上海交通大学 出版社出版发行
(上海市番禺路 951 号 邮政编码 200030)
电话：64071208 出版人：韩建民
上海交大印务有限公司印刷 全国新华书店经销
开本：787mm×960mm 1/16 印张：10.25 字数：191 千字
2010 年 4 月第 1 版 2010 年 4 月第 1 次印刷
ISBN 978-7-313-06369-4/TP 定价：48.00 元

Preface

The study and use of population-based probabilistic modeling techniques for optimization have been successfully developed during the last decade. Among these techniques, Genetic algorithms (GAs) and Estimation of Distribution Algorithms (EDAs) have been the reference.

This book, comprised of a total of 9 chapters, covers broadly important spectrum subjects ranging from fundamental theories of GAs and EDAs, development of a new type of EDAs and applications of EDAs to efficiency enhancements of EDAs. In chapter 1, GA fundamentals are discussed. We begin with what is usually the most critical decision in any application, namely that of deciding how best a candidate solution is represented to the algorithm. We then describe variation operators suitable for different types of representation, before turning our attention to the selection and replacement mechanisms that are used to manage the populations of solutions. In chapter 2, the EDAs proposed for the solution of combinatorial optimization problems and optimization in continuous domains are reviewed. Different approaches for EDAs have been ordered by the complexity of interrelations so that they are able to express. An empirical comparison of EDAs in binary search spaces is covered in chapter 3. Furthermore, techniques of implementations of a new type of EDAs are studied in chapter 4. The experimental results of applying EDAs to some optimization problems are shown in chapter 5. Chapter 6, 7 and 8 bring together some EDAs approaches to optimization problems in the fields of graph matching and resource management. Finally, chapter 9 provides an overview of different efficiency-enhancement techniques for EDAs.

This book should be of interested to theoreticians and practitioners alike, and is a must – have resources for those interested in optimization in general, and genetics and estimation of distribution algorithms in particular. Also engineers who, in their daily life, face real-world optimization problems can derive benefit from the reading of the

book. Moreover, this book may be used by graduate students in computer science and by people interested in the development of these new techniques that, in the following years, will provide us with interesting and appealing challenges.

Qun Jiang
College of Computer Science and Engineering
Chongqing University of Technology

Contents

Chapter 1
Fundamentals and Literature

1.1 Optimization Problems

In a broader sense, a genetic algorithm is any population-based model that uses selection and recombination operators to generate new sample points in a search space. This type of algorithm is often viewed as function optimizers, and the range of optimizing problems to which genetic algorithms have been applied is quite broad. Usually, considering a parameter optimization problem we must optimize a set of variables either to maximize some target such as profit, or to minimize cost or some measure of error. We might view such a problem as a black box with a series of control dials representing different parameters; the only output of the black box is a value returned by an evaluation function indicating how well a particular combination of parameter settings solves the optimization problem. The goal is to set the various parameters so as to optimize some output. In more traditional terms, we wish to minimize (or maximize) some function $F(X_1, X_2, \cdots, X_M)$. There are many optimization methods that have been developed in mathematics and operations research. What role do genetic algorithms play as an optimization tool? Genetic algorithms are often described as a global search method that does not use gradient information. Thus, non-differentiable functions as well as functions with multiple local optima represent classes of problems to which genetic algorithms might be applied. Genetic algorithms, as a weak method, are robust but very general. If there exists a good specialized optimization method for a specific problem, then genetic algorithm may not be the best optimization tool for that application.

In a strict interpretation, the genetic algorithm refers to a model introduced and

investigated by John Holland (1975) and by students of Holland. It is still the case that most of the existing theory for genetic algorithms applies either solely or primarily to the model introduced by Holland, as well as variations on what will be referred to in this chapter as the canonical genetic algorithm. Recent theoretical advances in modeling genetic algorithms also apply primarily to the canonical genetic algorithm (Vose, 1993).

1.2 Canonical Genetic Algorithm

The canonical genetic algorithm has a binary representation, fitness proportionate selection, a low probability of mutation, and an emphasis on genetically inspired recombination as a means of generating new candidate solutions. This is summarized in Table 1.1

Table 1.1 Sketch of the Canonical GA

Representation	Bit-string
Recombination	1-Point crossover
Mutation	Bit flip
Parent selection	Fitness proportional
Survival selection	Generational

To illustrate this, we show the details of one selection - reproduction cycle on a simple problem [1], namely that of maximizing the values of x^2 for x in the range of 0 ~ 31. Using a simple 5-bit binary encoding, Table 1.2 shows a random initial population of four genotypes, the corresponding phenotype and their fitness values. The column Pr ob_i shows the probability that an individual $i \in \{1, 2, 3, 4\}$ is chosen to be a parent, which for fitness proportionate selection is Pr $ob_i = f_i / \sum f_j$. The number of parents is the same as the size of our population, so the expected number of copies of each individual after selection is $f_i / \bar{f}$. As can be seen, these numbers are not integers; rather they represent a probability distribution, and the mating pool is creaked by making the number of random choices to sample from this distribution. The column "Actual count" stands for the number of copies in the mating pool, i.e., it shows one possible outcome.

Table 1.2 The x^2 example, 1: initialization, evaluation, and parent selection

String No.	Initial population	x Value	Fitnessf $(x) = x^2$	Pr ob_i	Expected	Actual count
1	01101	12	169	0.14	0.58	1
2	11000	24	576	0.49	1.97	2
3	01000	8	64	0.06	0.22	0
4	10011	19	361	0.31	1.23	1
Sum			1,170	1	4	4
Average			293	0.25	1	1
Max			576	0.49	1.97	2

The selection individuals are paired at random, and for each pair a random point along the string is chosen. Table 1.3 shows the results of crossover on the given mating pool for crossover points after the fourth and second genes, respectively, together with the corresponding fitness values.

Table 1.3 The x^2 example, 2: crossover, and offspring evaluation

String No.	Mating pool	Crossover point	Offspring after xover	x Value	Fitness $f(x) = x^2$
1	0110|1	4	01100	12	144
2	1100|0	4	11001	25	625
3	11|000	2	11011	27	729
4	10|011	2	10000	16	256
Sum					1,754
Average					439
Max					729

In canonical GA mutation works by generating a random number in each bite position, and comparing it to affixed low (e.g. 0.001) value, usually called the mutation rate. If the random number is below that rate, the value of the gene in the corresponding position is flipped. In our example we have $4 \times 5 = 20$ genes, and Table 1.4 shows the outcome of mutation when the first and eighteenth values in our sequence of random numbers are below the bitwise mutation probability. In this case, the mutation shown happened to have caused positive changes in fitness, but we should emphasis that in later generations, as the number of "ones" in the population rises, mutation will be on average (but not always) deleterious. Although manually engineered, this example shows a typical progress: the average fitness grows from 293 to 634.5, and the best fitness in the population from 576 to 784 after crossover and

mutation.

Table 1.4 The x^2 example, 3: mutation, and offspring evaluation

String No.	Offspring after xover	Offspring after mutation	x Value	Fitness $f(x) = x^2$
1	01100	11100	28	784
2	11001	11001	25	625
3	11011	11011	27	729
4	10000	10100	20	400
Sum Average Max				2, 538 634. 5 784

1.3 Individual Representations

The first step in the implementation of any genetic algorithm is to decide on a genetic representation of a candidate solution to the problem. This involves defining the genotype and the mapping from genotype to phenotype.

When choosing a representation, it is important to choose the "right" representation for the problem being solved. Getting the representation is one of the most difficult parts of designing a good evolutionary algorithm. Often this only comes with practice and a good knowledge of the application domain.

1.3.1 Binary Representations

The first representation we look at is one of the simplest — the binary. This is one of the earliest representations, and historically many GAs have mistakenly used it almost independently of the problem they were trying to solve. Here the genotype consists simply of a string of binary digits — a bit-string.

For a particular application we have to decide how long the string should be, and how we will interpret it to produce a phenotype. In choosing the genotype-phenotype mapping for a specific problem, one has to make sure that encoding allows that all possible bit strings denote a valid solution to the given problem and that, vice versa, all possible solutions can be represented.

For some problems, particularly those concerning Boolean decision variables, the genotype-phenotype mapping is natural, but frequently bit-strings are used to encode

other non-binary information. For example, we might interpret a bit-string of length 80 as ten 8 – bit integers. Usually this is a mistake, and better results can be obtained by using the integer or real-valued representations directly.

One of the problems of coding numbers in binary is that different bits have different significance. This can be helped by using Gray coding, which is a variation on the way that integers are mapped on bit strings. The standard method has the disadvantage that the Hamming distance between two consecutive integers is often not equal to one. If the goal is to evolve an integer number, you would like to have the chance of changing a 7 into an 8 equal to that of changing it to a 6. The chance of changing 0111 to 1000 by independent bit-flips is not the same, however, as that of changing it to 0110. Gray coding is a representation which ensures that consecutive integers always have Hamming distance one.

1.3.2 Integer Representations

Binary representations are not always the most suitable if our problem more naturally maps onto a representation where different genes can take one of a set values. One obvious example of when this might occur is the problem of finding the optimal values for a set of variables that all take integer values. These values might be unrestricted, or might be restricted to a finite set: for example, if we are trying to evolve a path on square grid, we might restrict the values to the rest $\{0, 1, 2, 3\}$ representing $\{\text{North, East, South, West}\}$. In either case an integer encoding is probably more suitable than a binary encoding. When designing the encoding and variation operators, it is worth considering whether there are any natural relations between the possible values that an attribute can take. This might be obvious for ordinal attributes such as integers, but for cardinal attributes such as the compass points above, there may not be a natural ordering.

1.3.3 Real-Valued Representations

Often the most sensible way to candidate solution to a problem is to have a string of real values. This occurs when the values that we want to represent as genes come from a continuous rather than a discrete distribution. Of course, on a computer the precision of these real values is actually limited by the implementation, so we will refer to them as floating-point numbers. The genotype for a solution with k genes is

now a vector $(x_1, \cdots, x_k)$ with $x_i \in \mathbf{R}$.

1.3.4 Permutation Representations

Many problems naturally take the form of deciding on the order in which a sequence of events should occur. While other forms do occur (for example decoder functions based on unrestricted integer representation in Ref. [2, 3]) or "floating keys" based on real-valued representations in Ref. [4, 5]), the most natural representation of such problems is as a permutation of a set of integers. One immediate consequence is that while an ordinary GA string allows numbers to occur more than once, such sequences of integers will not represent valid permutations. It is clear that we need new variation operators to preserve the permutation property that each possible allele value occurs exactly once in the solution.

When choosing appropriate variation operators it is also worth bearing in mind that there are actually two classes of problems that are represented by permutations. In the first of these, the order in which events occur is important. This might happen when the events use limited resources or time, and a typical example of this sort of problem is the "job shop scheduling" problem.

An alternative type of order-based problems depends on adjacency, and is typified by the traveling sales person problem. The problem is to find a complete tour of n given cities of minimal length. The search space for this problem is very big: there are $(n-1)!$ different routes possible for n given cities. For $n = 30$ there are approximately 10^{32} different tours. We label the cities $1, 2, \cdots, n$. One complete tour is a permutation of the cities, so that for $n = 4$, the routes [1, 2, 3, 4] and [3, 4, 2, 1] are both valid. The difference from order-based problems can clearly be seen if we consider that the starting point of the tour is not important, thus [1, 2, 3, 4], [2. 3. 4. 1], [3, 4, 1, 2], and [4, 1, 2, 3] are all equivalent. Many examples of this class are also symmetric, so that [4, 3, 2, 1] and so on are also equivalent.

Finally, we should mention that there are two possible ways to encode a permutation. In the first (most commonly used) of these the i^{th} element of the representation denotes the event that happens in that place in the sequence (or the i^{th} destination visited). In the second, the value of the i^{th} element denotes the position in the sequence in which the i^{th} event happens. Thus for the four cities [A, B, C, D], and the permutation [3, 1, 2, 4], the first encoding denotes the tour [C, A, B, D]

and the second [B, C, A, D].

1.4 Mutation

Mutation is the generic name given to those variation operators that use only one parent and create on child by applying some kind of randomized change to the representation. The form taken depends on the choice of encoding used, as does the meaning of the associated parameter, which is often referred to as the mutation rate. In the descriptions below we concentrate on the choice of operators rather than of parameters.

1.4.1 Mutation for Binary Encodings

Although a few other schemes have been occasionally used, the most common mutation operator used for binary encodings considers each gene separately and allows each bit to flip with a small probability p_m. The actual number of values changed is thus not fixed, but depends on the sequence of random numbers drawn, so for an encoding of length L on average LP_m values will be changed. In Fig. 1.1. this is illustrated for the case where the third, fourth, and eighth random values generated are less than the bitwise mutation rate p_m.

Fig. 1.1 Bitwise mutation for binary encodings

A number of studies and recommendations have been made for the choice of suitable values for the bitwise mutation rate p_m, and it is worth noticing at the outset that the most suitable choice to use depends on the desired outcome. For example, does the application require a population in which all members have high fitness, or simply that one highly fit individual is found? However, most binary coded GAs use mutation rates in a range such that on average between one gene per generation and one gene per offspring is mutated.

1.4.2 Mutation for Integer Encodings

For integer encodings there are two principal forms of mutation used, both of which mutate each gene independently with user-defined probability p_m.

1.4.2.1 Random Resetting

Here the "bit-flipping" mutation of binary encodings is extended to "random resetting", so that with probability p_m a new value is chosen at random from the set of permissible values in each position. This is the most suitable operator to use when the genes encode for cardinal attributes, since all other gene values are equally likely to be chosen.

1.4.2.2 Creep Mutation

This schema was designed for ordinal attributes and works by adding a small value to each gene with probability p. Usually these values are sampled randomly for each position, from a distribution that is symmetric about zero, and is more likely to generate small changes than large ones. It should be noted that creep mutation requires a number of parameters controlling the distribution from which the random numbers are drawn, and hence the size of the steps that mutation takes in the search space. Finding appropriate settings for these parameters may not be easy, and it is sometimes common to use more than one mutation operator in tandem from integer based problems. For example, in Ref [6] both a "big creep" and a "little creep" operators are used. Alternatively, random resetting might be used with low probability, in conjunction with a creep operator that tended to make small changes relative to the range of permissible values.

1.4.3 Mutation for Floating-Point Encodings

For floating-point representations, it is normal to ignore the discretization imposed by hardware and consider the allele values as coming from a continuous rather than a discrete distribution, so the forms of mutation described above are no longer applicable. Instead it is common to change the allele value of each gene randomly within its domain given by a lower L_i and upper U_i bound, resulting in the following transformation:

$\langle x_1, \cdots, x_n \rangle \rightarrow \langle x_1', \cdots, x_n' \rangle$, where $x_i, x_i' \in [L_i, U_i]$ two types can be distinguished according to the probability distribution, from which the new gene values are drawn: uniform and non-uniform mutation.

1.4.3.1 Uniform Mutation

For this operator the values of x_i' are drawn uniformly randomly from $[L_i, U_i]$. This is the most straightforward option, analogous to bit-flipping for binary encodings and the random resetting sketched above for integer encodings. It is normally used

with a position-wise mutation probability.

1.4.3.2 Non-Uniform Mutation with a Fixed Distribution

Perhaps the most common form of non-uniform mutation used with floating-point representations takes a form analogous to the creep mutation for integers. It is designed so that usually, but not always, the amount of change introduced is small. This is achieved by adding to the current gene value an amount drawn randomly from a Gaussian distribution with mean zero and user-specified standard deviation, and then curtailing the resulting value to the range $[L_i, U_i]$ if necessary. The Gaussian (or normal) distribution has the property that approximately two thirds of the samples drawn lie within one standard deviation. This means that most of the changes made will be small, but there is nonzero probability of generating very large changes since the tail of the distribution never reaches zero. It is normal practice to apply this operator with the probability of one per gene, and instead the mutation parameter is used to control the standard deviation of the Gaussian and hence the probability distribution of the step sizes taken.

An alternative to the Gaussian distribution is use for a Cauchy distribution, which has a "fatter" tail. That is, the probabilities of generating larger values are slightly higher than for a Gaussian with the same standard deviation.

1.4.4 Mutation for Permutation Representations

For permutation representations, it is no longer possible to consider each gene independently; rather finding legal mutations is a matter of moving alleles around in the genome. This has the immediate consequence that the mutation parameter is interpreted as the probability that the string undergoes mutation, rather than that a single gene in the string is altered. The three most common forms of mutation used for order-based problems were first described in Ref. [7].

1.4.4.1 Swap Mutation

This operator works by randomly picking two positions (genes) in the string and swapping their allele values. This is illustrated in Fig. 1.2, where the values in positions two and five have been swapped.

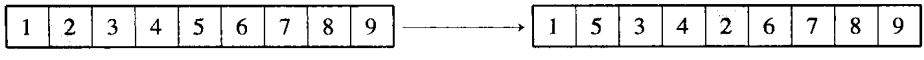

Fig. 1.2 Swap mutation