NEAR EAST UNIVERSITY

Faculty of Engineering

Department of Computer Engineering

GENETIC ALGORITHM BASE OPTIMIZATION

Graduation Project
COM – 400

Student: Cenay Bulut

Supervisor:Assoc.Prof.Dr.Rahib ABIYEV

Nicosia-2002

ACKNOWLEDGEMENT

First of all I want to thanks Asst.Dr.Prof Rahib ABIYEV whose helped that made the completion of this project.

I want to thanks to my family from beginning until now for supporting me. And I want to thanks to my frirends.

ABSTRACT

Genetic algorithms are a part of evolutionary computing, which is a rapidly growing area of artificial intelligence. As you can guess, genetic algorithms are inspired by Darwin's theory about evolution. Simply said, solution to a problem solved by genetic algorithms is evolved.

Optimization is a branch of mathematics with many real-world applications. Many problems in industry and in scientific fields involve finding the "best" value for a particular function, given a set of restrictions that must not be violated. Formally, the function that has to be "optimized" (generally maximized or minimized) is called the objective function. The factors which affect the value of the function are called the decision variables, and the restrictions (whether on the decision variables or the objective function itself) are called constraints

In the following chapters I will present in detailed information about both GENETIC ALGORITHM and OPTIMIZATION PROBLEM.

CONTENTS

INTRODUCTION

GENETIC ALGORITHMS were invented to mimic some of the processes observed in natural evolution. Many people, biologists included, are astonished that life at the level of complexity that we observe could have evolved in the relatively short time suggested by the fossil record. The father of the original Genetic Algorithm was John Holland who invented it in the early 1970's.

The study of genetic algorithms originated with John Holland in the mid-1970s. His original genetic algorithm is approximately the same as the ``Simple Genetic Algorithm" now found in the literature. Since this GENETIC ALGOTIHM is used as a starting point for almost all new work, it is worth describing it in detail.

A simple GENETIC ALGOTIHM represents solutions using strings of bits. These bits may encode integers, real numbers, sets, or whatever else is appropriate to the problem. Advocates argue that the use of bit-strings as a universal representation allows for a uniform set of simple operators, and simplify the task of analyzing GENETIC ALGOTIHM properties theoretically. Detractors argue that bitwise operators are often not appropriate for particular problems, and that analytic ease is too high a price to pay for performance. Today, most practical GENETIC ALGOTIHM systems use problem-specific representations (integers to represent integers, character strings to represent sets, and so on), and customize operations for these representations.

The operators provided by the simple GENETIC ALGOTIHM were 1-point crossover, mutation, and inversion. These were inspired directly by natural systems. Today, inversion has largely been dropped, and several different forms of crossover and mutation are used. Selection in the simple GENETIC ALGOTIHM was based directly on fitness: given a population of individuals, the probability of a particular individual passing its genes into the next generation was directly proportional to its fitness. Various ranking and selection schemes are now used instead of raw fitness in order to ensure that genetic drift does not occur, i.e. that good genes are less likely to disappear because of a bad accident.

As the previous paragraph hinted, simple GENETIC ALGOTIHM systems use generational update schemes. These are like the life cycles of many plant and insect

species: each generation produces the next and then dies off, so that an individual in generation g never has a chance to breed with one in generation g+1. As we shall see, continuous update schemes are also possible, in which children are gradually mixed into a single, continuously-evolving, population.

CHAPTER ONE

GENETIC ALGORITHMs

1.1. What is Genetic Algorithms?

Genetic Algorithms (GAs) are adaptive heuristic search algorithm based on the evolutionary ideas of natural selection and genetics. As such they represent an intelligent exploitation of a random search used to solve optimization problems. Although randomised, GAs are by no means random, instead they exploit historical information to direct the search into the region of better performance within the search space. The basic techniques of the GAs are designed to simulate processes in natural systems necessary for evolution, specially those follow the principles first laid down by Charles Darwin of "survival of the fittest.". Since in nature, competition among individuals for scanty resources results in the fittest individuals dominating over the weaker ones.

1.2.Why Genetic Algorithms?

It is better than conventional AI in that it is more robust. Unlike older AI systems, they do not break easily even if the inputs changed slightly, or in the presence of reasonable noise. Also, in searching a large state-space, multi-modal state-space, or n-dimensional surface, a genetic algorithm may offer significant benefits over more typical search of optimization techniques. (linear programming, heuristic, depth-first, breath-first, and praxis.)

1.3. Who can benefit from GAs?

Nearly everyone can gain benefits from Genetic Algorithms, once he can encode solutions of a given problem to chromosomes in GA, and compare the relative performance (fitness) of solutions. An effective GA representation and meaningful fitness evaluation are the keys of the success in GA applications. The appeal of GAs comes from their simplicity and elegance as robust search algorithms as well as from their power to discover good solutions rapidly for difficult high-dimensional problems.

GAs are useful and efficient when:

1. The search space is large, complex or poorly understood.
2. Domain knowledge is scarce or expert knowledge is difficult to encode to narrow the search space.
3. No mathematical analysis is available.
4. Traditional search methods fail.

The advantage of the GA approach is the ease with which it can handle arbitrary kinds of constraints and objectives; all such things can be handled as weighted components of the fitness function, making it easy to adapt the GA scheduler to the particular requirements of a very wide range of possible overall objectives.

GAs have been used for problem-solving and for modelling. GAs are applied to many scientific, engineering problems, in business and entertainment, including:

Optimization: GAs have been used in a wide variety of optimisation tasks, including numerical optimisation, and combinatorial optimisation problems such as traveling salesman problem (TSP), circuit design [Louis 1993] , job shop scheduling [Goldstein 1991] and video & sound quality optimisation.

Automatic Programming: GAs have been used to evolve computer programs for specific tasks, and to design other computational structures, for example, cellular automata and sorting networks.

Machine and robot learning: GAs have been used for many machine- learning applications, including classificationa and prediction, and protein structure prediction. GAs have also been used to design neural networks, to evolve rules for learning classifier systems or symbolic production systems, and to design and control robots.

Economic models: GAs have been used to model processes of innovation, the development of bidding strategies, and the emergence of economic markets.

Immune system models: GAs have been used to model various aspects of the natural immune system, including somatic mutation during an individual's lifetime and the discovery of multi-gene families during evolutionary time.

Ecological models: GAs have been used to model ecological phenomena such as biological arms races, host-parasite co-evolutions, symbiosis and resource flow in ecologies.

Population genetics models: GAs have been used to study questions in population genetics, such as "under what conditions will a gene for recombination be evolutionarily viable?"

Interactions between evolution and learning: GAs have been used to study how individual learning and species evolution affect one another.

Models of social systems: GAs have been used to study evolutionary aspects of social systems, such as the evolution of cooperation [Chughtai 1995], the evolution of communication, and trail-following behaviour in ants.

1.4. Difference between GA and traditional methods

Most traditional optimization methods used in science and engineering applications can be divided into two broad classes: direct search methods requiring only the objective function values and gradient search methods requiring gradient information either exactly or numerically.

1. These methods work on point-by-point basis. They start with an initial guess and a new solution is found iteratively.
2. Most of them are not guaranteed to find the global optimal solutions. The termination criterion is the value of gradient of objective function becomes close to zero.
3. They work with coding of the parameter set, not the parameters themselves.
4. Advantage of working with a coding of variable space is that the coding discretizes the search spaces even though the function may be continuous.

5. Since function values at various discrete solutions are required, a discrete or discontinuous function may be tackled using GAs.
6. They search from a population of points, not single point so it is very likely that the expected GA solution maybe a global solution
7. They use objective function values and not derivatives.
8. Probabilistic transition rules are used, not deterministic. The search can proceed in any direction.

1.5. Brief Overview

GAs were introduced as a computational analogy of adaptive systems. They are modelled loosely on the principles of the evolution via natural selection, employing a population of individuals that undergo selection in the presence of variation-inducing operators such as mutation and recombination (crossover). A fitness function is used to evaluate individuals, and reproductive success varies with fitness.

The Algorithms

1. Randomly generate an initial population M(0)
2. Compute and save the fitness u(m) for each individual m in the current population M(t)
3. Define selection probabilities p(m) for each individual m in M(t) so that p(m) is proportional to u(m)
4. Generate M(t+1) by probabilistically selecting individuals from M(t) to produce offspring via genetic operators
5. Repeat step 2 until satisfying solution is obtained.
6. The paradigm of GAs descibed above is usually the one applied to solving most of the problems presented to GAs. Though it might not find the best solution. more often than not, it would come up with a partially optimal solution.

CHAPTER TWO

OPERATIONS OF GENETIC ALGORITHMS

2.1. Chromosome

All living organisms consist of cells. In each cell there is the same set of chromosomes. Chromosomes are strings of DNA and serves as a model for the whole organism. A chromosome consists of genes, blocks of DNA. Each gene encodes a particular protein. Basically can be said, that each gene encodes a trait, for example color of eyes. Possible settings for a trait (e.g. blue, brown) are called alleles. Each gene has its own position in the chromosome. This position is called locus.

Complete set of genetic material (all chromosomes) is called genome. Particular set of genes in genome is called genotype. The genotype is with later development after birth base for the organism's phenotype, its physical and mental characteristics, such as eye color, intelligence etc.

2.1.1. Reproduction

During reproduction, first occurs recombination (or crossover). Genes from parents form in some way the whole new chromosome. The new created offspring can then be mutated. Mutation means, that the elements of DNA are a bit changed. This changes are mainly caused by errors in copying genes from parents.

The fitness of an organism is measured by success of the organism in its life.

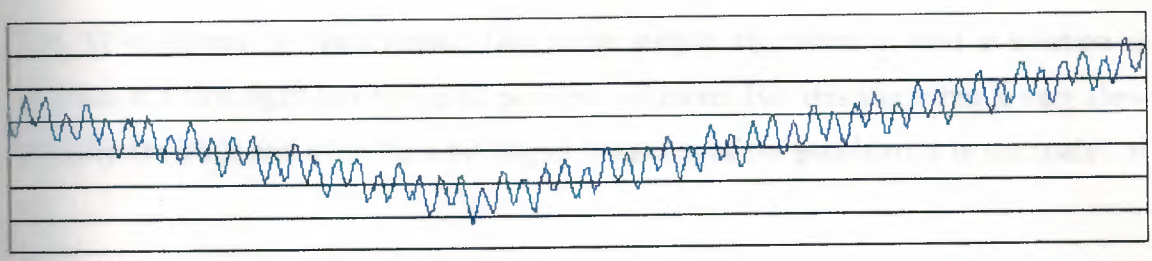
2.2. Search Space

2.2.1. Search Space

If we are solving some problem, we are usually looking for some solution, which will be the best among others. The space of all feasible solutions (it means objects among those the desired solution is) is called search space (also state space). Each point in the search space represents one feasible solution. Each feasible solution

can be "marked" by its value or fitness for the problem. We are looking for our solution, which is one point (or more) among feasible solutions – that is one point in the search space.

The looking for a solution is then equal to a looking for some extreme (minimum or maximum) in the search space. The search space can be whole known by the time of solving a problem, but usually we know only a few points from it and we are generating other points as the process of finding solution continues.



Example of a search space

The problem is that the search can be very complicated. One does not know where to look for the solution and where to start. There are many methods, how to find some suitable solution (i.e. not necessarily the best solution), for example hill climbing, tabu search, simulated annealing and genetic algorithm. The solution found by this method is often considered as a good solution, because it is not often possible to prove what is the real optimum.

2.2.2. NP-hard Problems

Examples of difficult problems, which cannot be solved in the "traditional" way, are NP problems.

There are many tasks for which we know fast (polynomial) algorithms. There are also some problems that are not possible to be solved algorithmically. For some problems was proved that they are not solvable in polynomial time.

But there are many important tasks, for which it is very difficult to find a solution, but once we have it, it is easy to check the solution. This fact led to NP-complete problems. NP stands for no deterministic polynomial and it means that it is possible to "guess" the

solution (by some no deterministic algorithm) and then check it, both in polynomial time. If we had a machine that can guess, we would be able to find a solution in some reasonable time.

Studying of NP-complete problems is for simplicity restricted to the problems, where the answer can be yes or no. Because there are tasks with complicated outputs, a class of problems called NP-hard problems has been introduced. This class is not as limited as class of NP-complete problems.

For NP-problems is characteristic that some simple algorithm to find a solution is obvious at a first sight just trying all possible solutions. But this algorithm is very slow (usually O (2^n)) and even for a bit bigger instances of the problems it is not usable at all.

Today nobody knows if some faster exact algorithm exists. Proving or disproving this remains as a big task for new researchers (and maybe you). Today many people think, that such an algorithm does not exist and so they are looking for some alternative methods, example of these methods are genetic algorithms.

Examples of the NP problems are satisfiability problem, traveling salesman problem or knapsack problem. Compendium of NP problems is available.

2.3. Parameters of Genetic Algorithms

2.3.1. Crossover and Mutation Probability

There are two basic parameters of GA - crossover probability and mutation probability. Crossover probability says how often will be crossover performed. If there is no crossover, offspring is exact copy of parents. If there is a crossover, offspring is made from parts of parents' chromosome. If crossover probability is 100%, then all offspring is made by crossover. If it is 0%, whole new generation is made from exact copies of chromosomes from old population (but this does not mean that the new generation is the same!).Crossover is made in hope that new chromosomes will have good parts of old chromosomes and maybe the new chromosomes will be better. However it is good to leave some part of population survive to next generation.

Mutation probability says how often will be parts of chromosome mutated. If there is no mutation, offspring is taken after crossover (or copy) without any change. If mutation is performed, part of chromosome is changed. If mutation probability is 100%, whole chromosome is changed, if it is 0%, nothing is changed. Mutation is made to prevent falling GA into local extreme, but it should not occur very often, because then GA will in fact change to random search.

2.3.2. Other Parameters

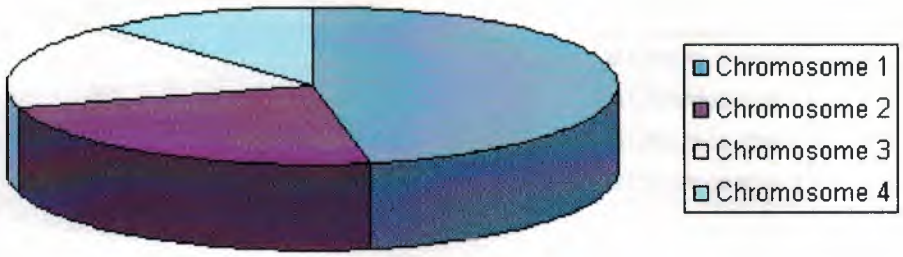
There are also some other parameters of GA. One also important parameter is population size. Population size says how many chromosomes are in population (in one generation). If there are too few chromosomes, GA have a few possibilities to perform crossover and only a small part of search space is explored. On the other hand, if there are too many chromosomes, GA slows down. Research shows that after some limit (which depends mainly on encoding and the problem) it is not useful to increase population size, because it does not make solving the problem faster.

2.4. Selection

As you already know from the Genetic algorithm outline, chromosomes are selected from the population to be parents to crossover. The problem is how to select these chromosomes. According to Darwin's evolution theory the best ones should survive and create new offspring. There are many methods how to select the best chromosomes, for example roulette wheel selection, Boltzman selection, tournament selection, rank selection, steady state selection and some others.

2.4.1. Roulette Wheel Selection

Parents are selected according to their fitness. The better the chromosomes are, the more chances to be selected they have. Imagine a roulette wheel where are placed all chromosomes in the population, every has its place big accordingly to its fitness function, like on the following picture.



Then a marble is thrown there and selects the chromosome. Chromosome with biggest fitness will be selected more times.

Following algorithm can simulate this.

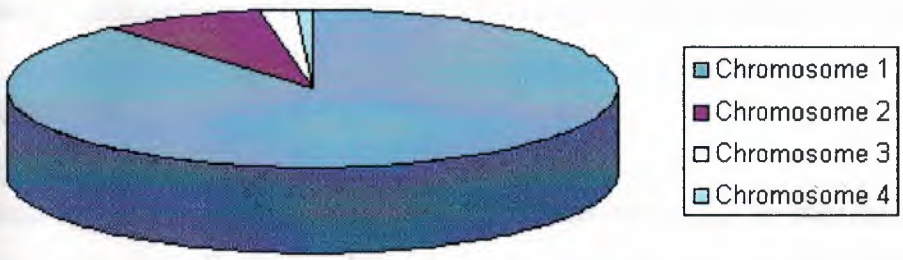
1. [Sum] Calculate sum of all chromosome fitness's in population - sum S.
2. [Select] Generate random number from interval (0,S) - r.
3. [Loop] Go through the population and sum fitness's from 0 - sum s. When the sum s is greater then r, stop and return the chromosome where you are.

Of course, step 1 is performed only once for each population.

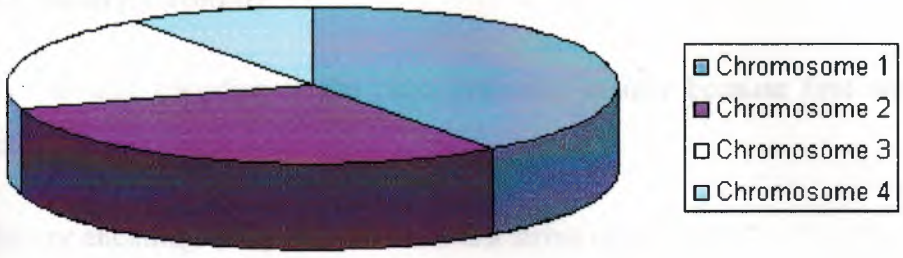
2.4.2. Rank Selection

The previous selection will have problems when the fitness's differs very much. For example, if the best chromosome fitness is 90% of the entire roulette wheel then the other chromosomes will have very few chances to be selected.

Rank selection first ranks the population and then every chromosome receives fitness from this ranking. The worst will have fitness 1, second worst 2 etc. and the best will have fitness N (number of chromosomes in population). You can see in following picture, how the situation changes after changing fitness to order number.



Situation before ranking (graph of fitness's)



Situation after ranking (graph of order numbers)

After this all the chromosomes have a chance to be selected. But this method can lead to slower convergence, because the best chromosomes do not differ so much from other ones.

2.4.3. Steady-State Selection

This is not particular method of selecting parents. Main idea of this selection is that big part of chromosomes should survive to next generation.

GA then works in a following way. In every generation is selected a few (good - with high fitness) chromosomes for creating a new offspring. Then some (bad - with low fitness) chromosomes are removed and the new offspring is placed in their place. The rest of population survives to new generation.

2.4.4. Elitism

Idea of elitism has been already introduced. When creating new population by crossover and mutation, we have a big chance, that we will loose the best chromosome.

Elitism is name of method, which first copies the best chromosome (or a few best chromosomes) to new population. The rest is done in classical way. Elitism can very rapidly increase performance of GA, because it prevents losing the best found solution.

2.5. Encoding

Encoding of chromosomes is one of the problems, when you are starting to solve problem with GA. Encoding very depends on the problem. In this section will be introduced some encodings, which have been already used with some success.

2.5.1. Binary Encoding

Binary encoding is the most common, mainly because first works about GA used this type of encoding.

In binary encoding every chromosome is a string of bits, 0 or 1.

Chromosome A	10110010110010101100101
Chromosome B	11111110000011000001111

Example of chromosomes with binary encoding

Binary encoding gives many possible chromosomes even with a small number of alleles. On the other hand, this encoding is often not natural for many problems and sometimes corrections must be made after crossover and/or mutation.

Example of Problem: Knapsack problem

The problem: There are things with given value and size. The knapsack has given capacity. Select things to maximize the value of things in knapsack, but do not extend knapsack capacity.

Encoding: Each bit says, if the corresponding thing is in knapsack.

2.5.2. Permutation Encoding

Permutation encoding can be used in ordering problems, such as travelling salesman problem or task ordering problem.

In permutation encoding, every chromosome is a string of numbers, which represents number in a sequence.

Chromosome A	1 5 3 2 6 4 7 9 8
Chromosome B	8 5 6 7 2 3 1 4 9

Example of chromosomes with permutation encoding

Permutation encoding is only useful for ordering problems. Even for this problems for some types of crossover and mutation corrections must be made to leave the chromosome consistent (i.e. have real sequence in it).

Example of Problem: Travelling salesman problem (TSP)
The problem: There are cities and given distances between them. Travelling salesman has to visit all of them, but he does not to travel very much. Find a sequence of cities to minimize travelled distance.
Encoding: Chromosome says order of cities, in which salesman will visit them.

2.5.3. Value Encoding

Direct value encoding can be used in problems, where some complicated values, such as real numbers, are used. Use of binary encoding for this type of problems would be very difficult. In value encoding, every chromosome is a string of some values. Values can be anything connected to problem, form numbers, real numbers or chars to some complicated objects.

Chromosome A	1.2324 5.3243 0.4556 2.3293 2.4545
Chromosome B	ABDJEIFJDHDIERJFDLDFLFEGT
Chromosome C	(back), (back), (right), (forward), (left)

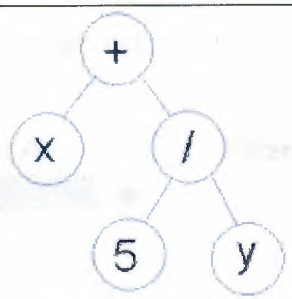
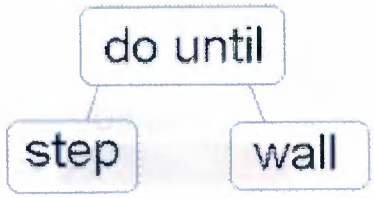
Example of chromosomes with value encoding

Value encoding is very good for some special problems. On the other hand, for this encoding is often necessary to develop some new crossover and mutation specific for the problem.

Example of Problem: Finding weights for neural network **the problem:** There is some neural network with given architecture. Find weights for inputs of neurons to train the network for wanted output. **Encoding**: Real values in chromosomes represent corresponding weights for inputs.

2.5.4 Tree Encoding

Tree encoding is used mainly for evolving programs or expressions, for genetic programming. In tree encoding every chromosome is a tree of some objects, such as functions or commands in programming language.

Chromosome A	Chromosome B
	
(+ x (/ 5 y))	(do until step wall)

Example of chromosomes with tree encoding.

Tree encoding is good for evolving programs. Programing language LISP is often used to this, because programs in it are represented in this form and can be easily parsed as a tree, so the crossover and mutation can be done relatively easily.

Example of Problem: Finding a function from given values **the problem:** Some input and output values are given. Task is to find a function, which will give the best (closest to wanted) output to all inputs.

Encoding: Chromosome is a function represented in a tree.

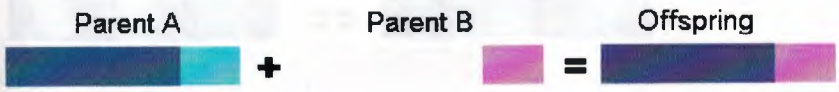
2.6. Crossover and Mutation

Crossover and mutation are two basic operators of GA. Performance of GA very depends on them. Type and implementation of operators depends on encoding and also on a problem. There are many ways how to do crossover and mutation. There are only some examples and suggestions how to do it for several encoding.

2.6.1. Binary Encoding

2.6.1.1. Crossover

Single point crossover - one crossover point is selected, binary string from beginning of chromosome to the crossover point is copied from one parent, the rest is copied from the second parent .



11001011+11011111 = 11001111

Two point crossover - two crossover point are selected, binary string from beginning of chromosome to the first crossover point is copied from one parent, the part from the first to the second crossover point is copied from the second parent and the rest is copied from the first parent

11001011 + 11011111 = 11011111

Uniform crossover - bits are randomly copied from the first or from the second parent

11001011 + 11011101 = 11011111

Arithmetic crossover - some arithmetic operation is performed to make a new offspring

11001011 + 11011111 = 11001001 (AND)

2.6.1.2. Mutation

Bit inversion - selected bits are inverted

11001001 => 10001001

2.6.2. Permutation Encoding

2.6.2.1. Crossover

Single point crossover - one crossover point is selected, till this point the permutation is copied from the first parent, then the second parent is scanned and if the number is not yet in the offspring it is added
Note: there are more ways how to produce the rest after crossover point

(1 2 3 4 5 6 7 8 9) + (4 5 3 6 8 9 7 2 1) = (1 2 3 4 5 6 8 9 7)

2.6.2.2 . Mutation

Order changing - two numbers are selected and exchanged

(1 2 3 4 5 6 8 9 7) => (1 8 3 4 5 6 2 9 7)

2.6.3. Value Encoding

2.6.3.1. Crossover

All crossovers from binary encoding can be used

2.6.3.2. Mutation

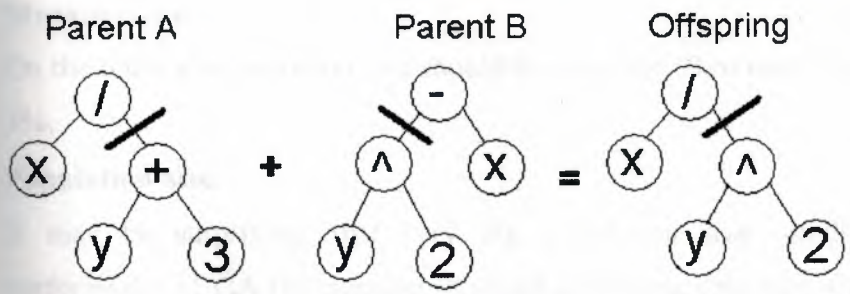
Adding a small number (for real value encoding) - to selected values is added (or subtracted) a small number

(1.29 5.68 2.86 4.11 5.55) => (1.29 5.68 2.73 4.22 5.55)

2.6.4. Tree Encoding

2.6.4.1. Crossover

Tree crossover - in both parent one crossover point is selected, parents are divided in that point and exchange part below crossover point to produce new offspring



2.6.4.2. Mutation

Changing operator, number - selected nodes are changed

2.7. Recommendations

2.7.1. Parameters of GA

This chapter should give you some basic recommendations if you have decided to implement your genetic algorithm. These recommendations are very general. Probably you will want to experiment with your own GA for specific problem, because today there is no general theory which would describe parameters of GA for any problem.

Recommendations are often results of some empiric studies of GAs, which were often performed only on binary encoding.

1. **Cross-over-rate**
   Crossover rate generally should be high, about 80%-95%. (However some results show that for some problems crossover rate about 60% is the best.)
2. **Mutation-rate**
   On the other side, mutation rate should be very low. Best rates reported are about 0.5%-1%.
3. **Population-size**
   It may be surprising, that very big population size usually does not improve performance of GA (in meaning of speed of finding solution). Good population size is about 20-30, however sometimes sizes 50-100 are reported as best. Some research also shows, that best population size depends on encoding, on size of encoded string. It means, if you have chromosome with 32 bits, the population should be say 32, but surely two times more than the best population size for chromosome with 16 bits.
4. **Selection**
   Basic roulette wheel selection can be used, but sometimes rank selection can be better. There are also some more sophisticated method, which changes parameters of selection during run of GA. Basically they behaves like simulated annealing. But surely elitism should be used (if you do not use other method for saving the best found solution). You can also try steady state selection.

5. **Encoding**

Encoding depends on the problem and also on the size of instance of the problem. Operators depend on encoding and on the problem.

2.7.2 - Applications of GA

Genetic algorithms have been used for difficult problems (such as NP-hard problems), for machine learning and also for evolving simple programs. They have been also used for some art, for evolving pictures and music.

Advantage of GAs is in their parallelism. GA is travelling in a search space with more individuals (and with genotype rather than phenotype) so they are less likely to get stuck in a local extreme like some other methods.

They are also easy to implement. Once you have some GA, you just have to write new chromosome (just one object) to solve another problem. With the same encoding you just change the fitness function and it is all. On the other hand, choosing encoding and fitness function can be difficult.

Disadvantage of GAs is in their computational time. They can be slower than some other methods. But with todays computers it is not so big problem.

to get an idea about problems solved by GA, here is a short list of some applications:

1. Nonlinear dynamical systems - predicting, data analysis
2. Designing neural networks, both architecture and weights
3. Robot trajectory
4. Evolving LISP programs (genetic programming)
5. Strategy planning
6. Finding shape of protein molecules
7. TSP and sequence scheduling
8. Functions for creating images

CHAPTER THREE

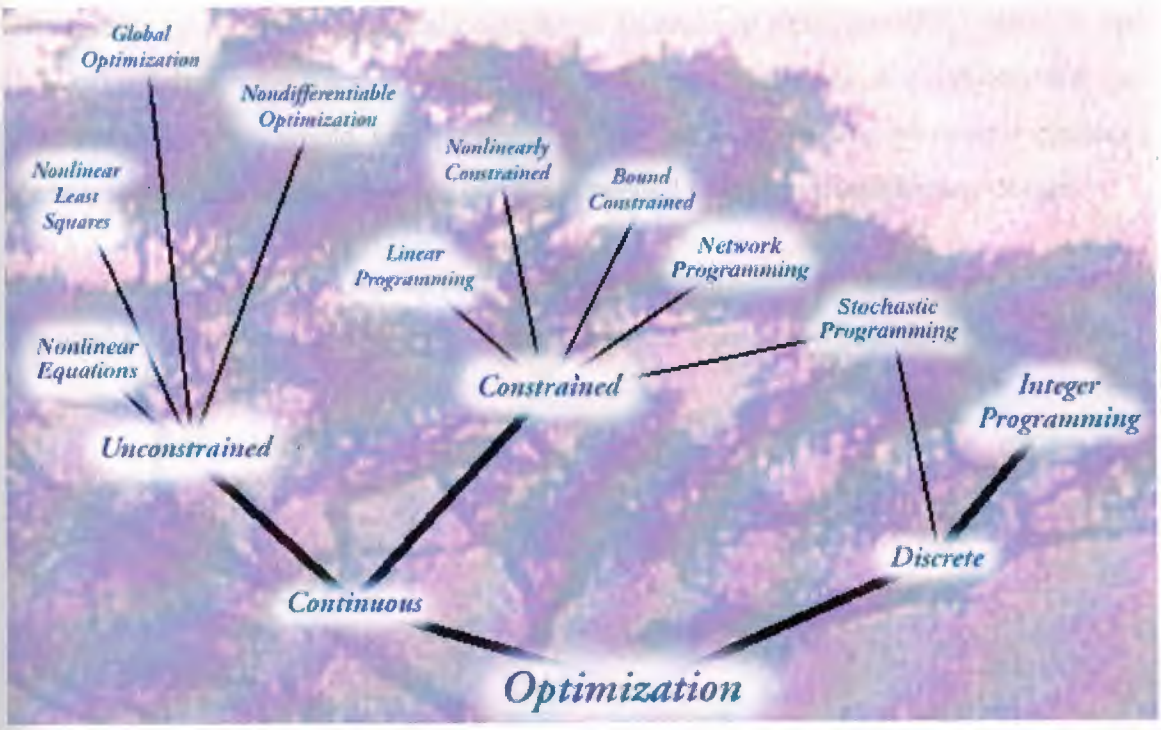
OPTIMIZATION PROBLEM

3 . What Is Optimization

Optimization problems are made up of three basic ingredients:

1. An objective function which we want to minimize or maximize. For instance, in a manufacturing process, we might want to maximize the profit or minimize the cost. In fitting experimental data to a user-defined model, we might minimize the total deviation of observed data from predictions based on the model. In designing an automobile panel, we might want to maximize the strength.
2. A set of unknowns or variables which affect the value of the objective function. In the manufacturing problem, the variables might include the amounts of different resources used or the time spent on each activity. In fitting-the-data problem, the unknowns are the parameters that define the model. In the panel design problem, the variables used define the shape and dimensions of the panel.
3. A set of constraints that allow the unknowns to take on certain values but exclude others. For the manufacturing problem, it does not make sense to spend a negative amount of time on any activity, so we constrain all the "time" variables to be non-negative. In the panel design problem, we would probably want to limit the weight of the product and to constrain its shape.

The Optimization Tree is an online guide to the field of numerical optimization. It introduces the different subfields of optimization and includes outlines of the major algorithms in each area, with pointers to software packages where appropriate. The connections between the Tree's web pages mirrors the relationships between these different areas. Follow the pathways through the tree to see how everything hangs together!



3.1. Multi-Objective Optimization

3.1.1 - Introduction

Most realistic optimization problems, particularly those in design, require the simultaneous optimization of more than one objective function. Some examples:

In bridge construction, a good design is characterized by low total mass and high stiffness.

Aircraft design requires simultaneous optimization of fuel efficiency, payload, and weight.

In chemical plant design, or in design of a groundwater remediation facility, objectives to be considered include total investment and net operating costs.

A good sunroof design in a car could aim to minimize the noise the driver hears and maximize the ventilation.

The traditional portfolio optimization problem attempts to simultaneously minimize the risk and maximize the fiscal return. In these and most other cases, it is unlikely that the different objectives would be optimized by the same alternative parameter choices. Hence, some trade-off between the criteria is needed to ensure a satisfactory design.

Multicriteria optimization has its roots in late-nineteenth-century welfare economics, in the works of Edgeworth and Pareto. A mathematical description is as follows:

$$\min_{x \in C} F(x) = \begin{bmatrix} f_1(x) \\ f_2(x) \\ \vdots \\ f_n(x) \end{bmatrix} \qquad \ldots (MOP)$$

where n >= 2 and

$$C = \{x : h(x) = 0,\ g(x) \leq 0,\ a \leq x \leq b\}$$

denotes the feasible set constrained by equality and inequality constraints and explicit variable bounds. The space in which the objective vector belongs is called the *objective space* and image of the feasible set under F is called the *attained set*.

The scalar concept of ``optimality" does not apply directly in the multiobjective setting. A useful replacement is the notion of *Pareto optimality*. Essentially, a vector $x^* \in C$ is said to be Pareto optimal for (MOP) if all other vectors $x \in C$ have a higher value for at least one of the objective functions $f_i(\cdot)$, or else have the same value for all objectives. Formally speaking, we have the following definition:

A point $x^* \in C$ is said to be (glob ally) *Pareto optimal* or a (globally) efficient solution or a non-dominated or a non-inferior point for (MOP) if and only if there is no $x \in C$ such that $f_i(x) \leq f_i(x^*)$ for all $i \in \{1, 2, \ldots, n\}$, with at least one strict inequality.

Pareto optimal points are also known as *efficient*, *non-dominated*, or *non-inferior* points.

We can also speak of *locally* Pareto optimal points, for which the definition is the same as the one just given, except that we restrict attention to a feasible neighborhood of x^*.

That is, if $B(x^*, \delta)$ denotes a ball of radius δ around the point

We can also speak of *locally* Pareto optimal points, for which the definition is the same as the one just given, except that we restrict attention to a feasible neighborhood of x^*.

That is, if $B(x^*, \delta)$ denotes a ball of radius δ around the point x^*, we require that for some $\delta > 0$, there is *no* $x \in C \cap B(x^*, \delta)$ such that

$$f_i(x) \leq f_i(x^*), \qquad \text{for all } i = \{1, 2, \ldots, n\}$$

with at least one strict inequality.

Typically, there is an entire curve or surface of Pareto points, whose shape indicates the nature of the tradeoff between different objectives.

3.2. Integer Programming

In many applications, the solution of an optimization problem makes sense only if certain of the unknowns are integers. *Integer linear programming* problems have the general form

$$\min\{c^T : A_x = b, \quad x \geq 0, \quad x \in Z^n\} \qquad (1.1)$$

where Z^n is the set of n-dimensional integer vectors. In *mixed-integer linear programs*, some components of x are allowed to be real. We restrict ourselves to the pure integer case, bearing in mind that the software can also handle mixed problems with little additional complication of the underlying algorithm.

Integer programming problems, such as the *fixed-charge network flow* problem and the famous *traveling salesman* problem, are often expressed in terms of binary variables. The fixed-charge network problem modifies the minimum-cost network flow paradigm

by adding a term $f_{ij} y_{ij}$ to the cost, where the binary variable y_{ij} is set to 1 if arc (i, j) carries a nonzero flow χ_{ij}; it is set to zero otherwise.

In other words, there is a fixed overhead cost for using the arc at all. In the traveling salesman problem, we need to find a tour of a number of cities that are connected by directed arcs, so that each city is visited once and the time required to complete the tour is minimized. One binary variable is assigned to each directed arc; a variable χ_{ij} is set to 1 if city i immediately follows city j on the tour, and to zero otherwise.

3.2.1. Branch and Bound

Although a number of algorithms have been proposed for the integer linear programming problem, the *branch-and-bound* technique is used in almost all of the software in our survey. This technique has proven to be reasonably efficient on practical problems, and it has the added advantage that it solves continuous linear programs as subproblems, that is, linear programming problems without integer restrictions.

The branch-and-bound technique can be outlined in simple terms. An *enumeration tree* of continuous linear programs is formed, in which each problem has the same constraints and objective as (1.1) except for some additional bounds on certain components of χ. At the root of this tree is the problem (1.1) with the requirement $\chi \in Z^n$ removed. The solution χ to this root problem will not, in general,have all integer components. We now choose some noninteger solution component χ_i and define I_j to be the integer part of χ_j, that is, $I_j = \lfloor \chi_j \rfloor$. This gives rise to two subproblems. The left-child problem has the additional constraint $\chi_j \leq I_j$, whereas in the right-child problem we impose $\chi_j \geq I_j + 1$. This *branching* process can be carried out recursively; each of the two new problems will give rise to two more problems when we branch on one of the noninteger components of their solution. It follows from this construction that the enumeration tree is binary.

Eventually, after enough new bounds are placed on the variables, integer solutions $x \in Z^n$ are obtained. The value z_{opt} of $c^T x$ for the best integer solution found so far is retained and used as a basis for pruning the tree. If the continuous problem at one of the nodes of the tree has a final objective value greater than z_{opt}, so do all of its descendants, since they have smaller feasible regions and hence even larger optimal objective values. The branch emanating from such a node cannot give rise to a better integer solution than the one obtained so far, so we consider it no further; that is, we prune it. Pruning also occurs when we have added so many new bounds to some continuous problem that its feasible region disappears altogether.

The preferred strategy for solving the node problems in the enumeration tree is of the depth-first type: When two child nodes are generated from the current node, we choose one of these two children as the next problem to solve. One reason for using this strategy is that on practical problems the optimal solution usually occurs deep in the tree. There is also a computational advantage: If the dual simplex algorithm is used to solve the linear program at each node, the solution of the child problem can be found by a simple update to the basis matrix factorization obtained at the parent node. The linear algebra costs are trivial.

Two important questions remain: How do we select the noninteger component on which to branch, and how do we choose the next problem to solve if the branch we are currently working on is pruned? The answer to both questions depends on maintaining a lower bound z^I on the objective value for the continuous linear program at node I, and an estimate $z_{\text{int}}^{\text{I}}$ of the objective value of the best integer solution for the problem at node I. Both values can be calculated when the problem at node I is generated as the child of another problem. After the current branch has reached a dead end, there are two common strategies for selecting the next problem: Choose the one for which z^I is least or choose the one for which $z_{\text{int}}^{\text{I}}$ is least. Other strategies use some criterion function that combines z^I, $z_{\text{int}}^{\text{I}}$ and z_{opt}

In many codes, the user is allowed to specify a *branching order* to guide the choice of components on which to branch. By the nature of the problem, some components of

may be more significant than others; the algorithms can use this knowledge to make branching decisions that lead to the solution more quickly. In the absence of such an ordering, the *degradations* in the objective value are estimated by forcing each component in turn to its next highest or next lowest integer. The branching variable is often chosen to be the one for which the degradation is greatest.

The CPLEX, FortLP, LAMPS, LINDO, MIP III, OSL, and PC-PROG packages use the branch-and-bound technique to solve mixed-integer linear programs. The NAG Fortran Library (Chapter H) contains a branch-and-bound subroutine, and also an earlier implementation of a cutting plane algorithm due to Gomory. (The latter code is scheduled for removal from the library in the near future.) GAMS interfaces with a number of mixed-integer linear programming solvers, and even with a mixed-integer nonlinear programming solver. LINDO has two other front-end systems: LINGO provides a modeling interface to it, while What's*Best!* provides a variety of spreadsheet interfaces.

The CPLEX integer programming system can be used either as a stand-alone system or as a subroutine that is called from the user's code. CPLEX also implements a *branch-and-cut* strategy, in which the bounds on optimal objective values are tightened by adding additional inequality constraints

$$F_\chi \leq f$$

to the problem. The matrix F and vector f are chosen so that all integer vectors χ satisfying the original constraints $A_\chi = b, \chi \geq 0$ also satisfy $F_\chi \leq f$. These extra constraints (*cuts*) have the effect of reducing the size of the set of real vectors that is being considered at each node.

The Q01SUBS package contains routines to solve the quadratic zero-one programming problem

$$\min\left\{\frac{1}{2}\chi^{\mathrm{T}} Q_\chi + C^{\mathrm{T}}\chi : \chi_i \in \{0,1\}, i = 1,\ldots,n\right\}$$

where Q may be indefinite, while the QAPP package solves the assignment problem with a quadratic objective. Both algorithms use a branch-and-bound methodology similar to the techniques described above.

3.3. Bound-constrained optimization

Bound-constrained optimization problems play an important role in the development of software for the general constrained problem because many constrained codes reduce the solution of the general problem to the solution of a sequence of bound-constrained problems. The development of software for this problem, which we state as

$$\min\{f(\chi): l \le \chi \le u\}$$

is also important in applications because parameters that describe physical quantities are often constrained to lie in a given range.

Algorithms for the solution of bound-constrained problems seek a local minimizer χ^* of f. The standard first-order necessary condition for a local minimizer χ^* can be expressed in terms of the *binding* set

$$\mathrm{B}(\chi^*) = \{i: \chi_i^* = l_i, \partial_i f(\chi^*) \ge 0\} \cup \{i: \chi_i^* = u_i, \partial_i f(\chi^*) \le 0\}$$

at χ^* by requiring that

$$\partial_i f(\chi^*) = 0, \qquad i \notin B(\chi^*)$$

There are other ways to express this condition, but this form brings out the importance of the binding constraints. A second-order sufficient condition for χ^* to be a local minimizer of the bound-constrained problem is that the first-order condition hold and that

$$w^T \nabla^2 f(\chi^*) w > 0$$

for all vectors ω with $w \neq 0$ $w_i = 0,$ $i \in B_8(\chi^*)$ where

$$B_8(\chi^*) = B(\chi^*) \cap \{i : \partial_i f(\chi^* \neq 0)\}$$

is the *strictly binding* set at χ^*

Given any set of *free* variables f, we can define the *reduced gradient* and the *reduced Hessian* matrix, respectively, as the gradient of f and the Hessian matrix of f with respect to the free variables. In this terminology, the second-order condition requires that the reduced gradient be zero and that the reduced Hessian matrix be positive definite when the set F of free variables consists of all the variables that are not strictly binding at χ^*. As we shall see, algorithms for the solution of bound-constrained problems use unconstrained minimization techniques to explore the reduced problem defined by a set f_k of free variables. Once this exploration is complete, a new set of free variables is chosen with the aim of driving the reduced gradient to zero.

3.3.1 Newton Methods

IMSL, LANCELOT, MATLAB, NAG (NAG Fortran, or NAG C) OPTIMA, PORT 3, TN/TNBC , and VE08 implement quasi-Newton, truncated Newton, and Newton algorithms for bound-constrained optimization. The NEOS SERVER's bound-constrained minimization facility also implements a quasi-Newton algorithm. All these codes implement line-search and trust-region versions of unconstrained minimization algorithms, so our discussion here is brief, emphasizing the differences between the unconstrained and bound-constrained cases.

A line-search method for bound-constrained problems generates a sequence of iterates by setting

$$\chi_{k+1} = \chi_k + \alpha_k d_k,$$

where χ_k is a feasible approximation to the solution, d_k is a search direction, and $a_k > 0$ is the step. The direction is d_k obtained as an approximate minimizer of the subproblem

$$\min\left\{\nabla f(\chi_k)^{\mathrm{T}} d + \frac{1}{2} d^{\mathrm{T}} B_k d : d_i = 0, \quad i \in W_k\right\}, \qquad (1.1)$$

where W_k is the *working* set and B_k is an approximation to the Hessian matrix of at χ_k. All variables in the working set W_k are fixed during this iteration, while all other variables are in the free set F_k. We can express this subproblem in terms of the free variables by noting that it is equivalent to the unconstrained problem

$$\min\left\{g_k^{\mathrm{T}} w + \frac{1}{2} w^{\mathrm{T}} A_k w : w \in R^{m_k}\right\},$$

where m_k is the number of free variables, A_k is the matrix obtained from B_k by taking those rows and columns whose indices correspond to the free variables, and g_k is obtained from $\nabla f(\chi_k)$ by taking the components whose indices correspond to the free variables,

The main requirement on w_k is that d_k be a feasible direction, that is, $\chi_k + \alpha d_k$ satisfies the constraints for all $\alpha > 0$ sufficiently small. This is certainly the case if $w_k = A(\chi_k)$,

where

$$A(\chi) = \{i : \chi_i = l_i\} \cup \{i : \chi_i = u_i\}$$

is the set of *active* constraints at χ. As long as progress is being made with the current

W_k, the next working set W_{k+1} is obtained by merging $A(\chi_{k+1})$ with W_k. This updating process is continued until the function cannot be reduced much further with the current working set. At this point, the classical strategy is to drop a constraint in W_k for which $a_i f(\chi_k)$ has the wrong sign, that is, $i \in W_k$ but $i \in B\,\chi_k$, where the binding set

$$B(\chi) = \{i : \chi_i = l_i, a_i f(\chi) \geq 0\} \cup \{i : \chi_i = u_i, a_i f(\chi) \leq 0\}$$

is defined as before. In general it is advantageous to drop more than one constraint, in the hope that the algorithm will make more rapid progress towards the optimal binding set. However, all dropping strategies are constrained by the requirement that the solution d_k of the subproblem be a feasible direction.

An implementation of a line-search method based on subproblem (1.1) must cater to the situation in which the reduced Hessian matrix A_k is indefinite, because in this case the subproblem does not have a solution. This situation may arise, for example, if B_k is the Hessian matrix or an approximation obtained by differences of the gradient. Here, it is necessary to specify d_k by other means. For example, we can use the modified Cholesky factorization.

Quasi-Newton methods for bound-constrained problems update an approximation to the reduced Hessian matrix since, as already noted, only the reduced Hessian matrix is likely to be positive definite. The updating process is not entirely satisfactory because there are situations in which a positive definite update that satisfies the quasi-Newton condition does not exist. Moreover, complications arise because the dimension of the reduced matrix changes when the working set W_k changes. Quasi-Newton methods are usually beneficial when the working set remains fixed during consecutive iterations.

The choice of line-search parameter α_k is quite similar to the unconstrained case. If subproblem (1.1) has a solution and $\chi_k + d_k$ violates one of the constraints, then we compute the largest $\mu_k \in (0,1)$ such that

$$\chi_k + \mu_k d_k$$

is feasible. A standard strategy for choosing α_k is to seek an $\alpha_k \in (0, \mu_k)$ that satisfies the sufficient decrease and curvature conditions. We are guaranteed the existence of such an α_k unless μ_k satisfies the sufficient decrease condition and

$$\nabla f(\chi_k + \mu_k d_k)^T d_k < 0$$

This situation is likely to happen if, for example, f is strictly decreasing on the line segment $[\chi_k, \chi_k + \mu_k d_k]$. In this case it is safe to set $\alpha_k = \mu_k$.

3.3.2. Gradient-Projection Methods

Active set methods have been criticized because the working set changes slowly; at each iteration at most one constraint is added to or dropped from the working set. If there are k_0 constraints active at the initial w_0, but k_8 constraints active at the solution, then at least $|k_8 - k_0|$ iterations are required for convergence. This property can be a serious disadvantage in large problems if the working set at the starting point is vastly different from the active set at the solution. Consequently, recent investigations have led to algorithms that allow the working set to undergo radical changes at each iteration and to interior-point algorithms that do not explicitly maintain a working set.

The *gradient-projection* algorithm is the prototypical method that allows large changes in the working set at each iteration. Given χ_k, this algorithm searches along the piecewise linear path

$$p[x_k - \alpha \nabla f(x_k)], \alpha \geq 0$$

where p is the projection onto the feasible set. A new point

$$x_{k+1} = p[x_k - \alpha_k \nabla f(x_k)]$$

is obtained when a suitable $\alpha_k > 0$ is found. For bound-constrained problems, the projection can be easily computed by setting

$$[p(x)_i = mid\{x_i, l_i, u_i\}]$$

where $mid\{\}$ is the middle (median) element of a set. The search for α_k has to be done carefully since the function

$$\Phi(\alpha) = f(p[x_k - \alpha \nabla f(x_k)])$$

is only piecewise differentiable.

If properly implemented, the gradient-projection method is guaranteed to identify the active set at a solution in a finite number of iterations. After it has identified the correct active set, the gradient-projection algorithm reduces to the steepest-descent algorithm on the subspace of free variables. As a result, this method is invariably used in conjunction with other methods with faster rates of convergence.

Trust-region algorithms can be extended to bound-constrained problems. The main difference between the unconstrained and the bound-constrained version is that we now require the step s_k to be an approximate solution of the subproblem

$$\min\{q_k(s) : \|D_k s\| \le \Delta_k, l \le x_k + s \le u\}$$

where

$$q_k(s) = \nabla f(x_k)^T s + \frac{1}{2} s^T B_k s$$

An accurate solution to this subproblem is not necessary, at least on early iterations. Instead, we use the gradient-projection algorithm to predict a step s_k^c (the *Cauchy step*) and then require merely that our step, s_k, satisfies the constraints in the trust-region subproblem with $q_k(s_k) \le q_k(s_k^c)$. An approach along these lines is used by VE08 and

PORT 3. In the bound-constrained code in LANCELOT the trust region is defined by the l_∞- norm and $D_k = \mathrm{I}$, yielding the equivalent subproblem

$$\min\left\{q_k(8) : \max\left(l - x_k, \Delta_k e\right) \le 8 \le \min\left(u - x_k\right)\Delta_k e\right\}$$

where e is the vector of all ones.

The advantage of strategies that combine the gradient-projection method with trust-region methods is that the working set is allowed to change rapidly, and yet eventually settle into the working set for the solution. LANCELOT uses this approach, together with special data structures that exploit the (group) partially separable structure of f, to solve large bound-constrained problems.

3.4. Stochastic Programming

3.4.1. Introduction

All of the model formulations that you have encountered thus far in the Optimization Tree have assumed that the data for the given problem are known accurately. However, for many actual problems, the problem data cannot be known accurately for a variety of reasons. The first reason is due to simple measurement error The second and more fundamental reason is that some data represent information about the future (e.g., product demand or price for a future time period) and simply cannot be known with certainty. We will discuss a few ways of taking this uncertainty into account and, specifically, illustrate how stochastic programming can be used to make some optimal decisions.

3.4.2. Recourse

The fundamental idea behind stochastic linear programming is the concept of *recourse*. Recourse is the ability to take corrective action after a random event has taken place. A simple example of *two-stage recourse* is the following:

Choose some variables, ***x,*** to control what happens today.

Overnight, a random event happens.

Tomorrow, take some recourse action, y, to correct what may have gotten messed up by the random event.

We can formulate optimization problems to choose x and y in an optimal way. In this example, there are two periods; the data for the first period are known with certainty and some data for the future periods are stochastic, that is, random.

3.4.3. Example

You are in charge of a local gas company. When you buy gas, you typically deliver some to your customers right away and put the rest in storage. When you sell gas, you take it either from storage or from newly-arrived supplies. Hence, your decision variables are 1) how much gas to purchase and deliver, 2) how much gas to purchase and store, and 3) how much gas to take from storage and deliver to customers. Your decision will depend on the price of gas both now and in future time periods, the storage cost, the size of your storage facility, and the demand in each period. You will decide these variables for each time period considered in the problem. This problem can be modelled as a simple linear program with the objective to minimize overall cost. The solution is valid if the problem data are known with certainty, that is, if the future events unfold as planned.

More than likely, the future will not be precisely as you have planned; you don't know for sure what the price or demand will be in future periods though you can make good guesses. For example, if you deliver gas to your customers for heating purposes, the demand for gas and its purchase price will be strongly dependent on the weather. Predicting the weather is rarely an exact science; therefore, not taking this uncertainty into account may invalidate the results from your model. Your ``optimal" decision for one set of data may not be optimal for the actual situation.

3.4.4. Scenarios

Suppose in our example that we are experiencing a normal winter and that the next winter can be one of three scenarios: normal, cold, or very cold. To formulate this problem as a stochastic linear program, we must first characterize the uncertainty in the

model. The most common method is to formulate scenarios and assign a probability to each scenario. Each of these scenarios has different data as shown in the following table:

Scenario	Probability	Gas Cost ($)	Demand (units)
Normal	1/3	5.0	100
Cold	1/3	6.0	150
Very Cold	1/3	7.5	180

Both the demand for gas and its cost increase as the the weather becomes colder. The storage cost is constant, say, 1 unit of gas is $1per year. If we solve the linear program for each scenario separately, we arrive at three purchase/storage strategies:

Normal - Normal

Year	Purchase to Use	Purchase to Store	Storage	Cost
1	100	0	0	500
2	100	0	0	500

Total Cost = $1000

Normal - Cold

Year	Purchase to Use	Purchase to Store	Storage	Cost
1	100	0	0	500
2	150	0	0	900

Total Cost = $1400

Normal - Very Cold

Year	Purchase to Use	Purchase to Store	Storage	Cost
1	100	180	180	1580
2	0	0	0	0

Total Cost = $1580

We do not know which of the three scenarios will actually occur next year, but we would like our current purchasing decision to put is in the best position to minimize our expected cost. Bear in mind that by the time we make our second purchasing decision, we will know which of the three scenarios has actually happened.

3.5. Formulating a Stochastic Linear Program

Stochastic programs seek to minimize the cost of the first-period decision plus the expected cost of the second-period recourse decision.

$$\min \quad subject \quad to \quad e^{T}\chi + \mathrm{E}_{\omega} Q(\chi,\omega)$$
$$A_{\chi} = b$$
$$\chi \geq 0 \qquad where$$

$$Q(\chi,\omega) = \min \quad subject \quad to$$
$$d(\omega)^{T} y$$
$$T(\omega)\chi + W(\omega)y(\omega) = h(y)$$

The first linear program minimizes the first-period direct costs, $c^{T}x$ plus the expected recourse cost, over all of the possible scenarios while meeting the first-period constraints, Ax = b

The recourse cost Q depends both on x, the first-period decision, and on the random event, ω. The second LP describes how to choose $y(\omega)$ (a different decision for each random scenario ω). It minimizes the cost dTy subject to some recourse function, Tx + Wy = h. This constraint can be thought of as requiring some action to correct the system after the random event occurs. In our example, this constraint would require the purchase of enough gas to supplement the original amount on hand in order to meet the demand.

One important thing to notice in stochastic programs is that the first-period decision, x, is independent of which second-period scenario actually occurs. This is called the *nonanticipativity property.* The future is uncertain and so today's decision cannot take advantage of knowledge of the future.

3.6. Determistic Equivalet

The formulation above looks a lot messier than the deterministic LP formulation that we discuss elsewhere. However, we can express this problem in a deterministic for by introducing a different second-period y variable for each scenario. This formulation is called the *deterministic equivalent.*

$$\min \quad e^{\mathrm{T}} x + \sum_{i}^{N} = 1 \, p_i d_i^{\mathrm{T}} y_i$$

subject to

$$A_x = b$$

$$\mathrm{T}_i + W_i y_i = h_i, i = 1,\ldots,N$$

$$x \geq 0$$

$$y_i \geq 0$$

where N is the number of scenarios and is the probability of the scenario's occurrence.

For our three-scenario problem, we have

$$
\begin{aligned}
\min \quad & e^{T}x + p_1 d_1^{T} y_1 + p_2 d_2^{T} y_1 + p_3 d_3^{T} y_3 \\
s.\ t. \quad & A_x = b \\
& T_1 x + W_1 y_1 = h_1 \\
& T_2 x + W_2 y_2 = h_2 \\
& T_3 x + W_3 y_3 = h_3 \\
& x, y_i \geq 0
\end{aligned}
$$

Notice that the nonanticipativity constraint is met. There is only one first-period decision, x, whereas there are N second-period decisions, one for each scenario. The first-period decision cannot anticipate one scenario over another and must be feasible for each scenario. That is, Ax = b and Ti x + Wi yi = hi for i = 1,...,N Because we solve for all the decisions, x and yi simultaneously, we are choosing x to be (in some sense) optimal over all the scenarios.

Another feature of the deterministic equivalent is worth noting. Because the T and W matrices are repeated for every scenario in the model, the size of the problem increases linearly with the number of scenarios. Since the structure of the matrices remains the same and because the constraint matrix has a special shape, solution algorithms can take advantage of these properties. Taking uncertainty into account leads to more robust solutions but also requires more computational effort to obtain the solution.

3.7. Comparisons with Other Formulations

Because stochastic programs require more data and computation to solve, most people have opted for simpler solution strategies. One method requires the solution of the problem for each scenario. The solutions to these problems are then examined to find where the solutions are similar and where they are different. Based on this information, subjective decisions can be made to decide the best strategy.

3.7.1. Expected-Value Formulation

A more quantifiable approach is to solve the original LP where all the random data have been replaced with their expected values. Hopefully in this approach we will do all right on average. For our example then, we consider the (expected value) problem data to be

Year	Gas cost ($)	Demand
1	5.0	100
2	6.167	143.33

Solving this problem gives the following result:

Year	Purchase to Use	Purchase to Store	Storage	Cost
1	100	143.33	143.33	1360
2	0	0	0	0

Cost = $1360.00

Let's compute what happens in each scenario if we implement the expected value solution:

Scenario	Recourse Action	Recourse Cost	Total Cost
Normal	Store 43.33 excess @ $1 per unit	43.33	1403.33
Cold	Buy 6.67 units @ $6 per unit	40	1400
Very Cold	Buy 36.67 units @ $7.5 per unit	275	1635

The expected total cost over all scenarios is $\frac{1}{3}1403.33+\frac{1}{3}1400+\frac{1}{3}1635=\$1479,44$

3.7.2. Stochastic Programming Solution

Forming and solving the stochastic linear program gives the following solution:

Year	Purchase to Use	Purchase to Store	Storage	Cost
1 Normal	100	100	100	1100
2 Normal	0	0	0	0
2 Cold	50	0	0	300
2 Normal	80	0	0	600

$$\text{Cost} = 1100 + \frac{1}{3}300 + \frac{1}{3}600 = 1400\ \$$$

Similarly we can compute the costs for the stochastic programming solution for each scenario:

Scenario	Recourse Action	Recourse Cost	Total Cost
Normal	None	0	1100
Cold	Buy 50 units @ \$6 per unit	300	1400
Very Cold	Buy 80 units @ \$7.5 per unit	600	1700

The expected total cost over all scenarios is $\frac{1}{3}1100 + \frac{1}{3}1400 + \frac{1}{3}1700 = \$1400$

The difference in these average costs (\$79.44) is the value of the stochastic solution over the expected-value solution. Also notice that the cost of the stochastic solution is greater than or equal to the optimal solution for each scenario solved separately $1100 \geq 1100, 1400 \geq 1400$ and $1635 \geq 1580$. By solving each scenario alone, one assumes perfect information about the future to obtain a minimum cost. The stochastic solution is minimizing over a number of scenarios and, as a result, sacrifices the minimum cost for each scenario in order to obtain a robust solution over all the scenarios.

Conclusion

Randomness in problem data poses a serious challenge for solving many linear programming problems. The solutions obtained are optimal for the specific problem but may not be optimal for the situation that actually occurs. Being able to take this randomness into account is critical for many problems where the essence of the problem is dealing with the randomness in some optimal way. Stochastic programming enables the modeller to create a solution that is optimal over a set of scenarios.

3.7.3. Solution Techniques

The multiobjective problem is almost always solved by combining the multiple objectives into one scalar objective whose solution is a Pareto optimal point for the original MOP. Most algorithms have been developed in the linear framework (i.e. linear objectives and linear constraints), but the techniques described below are also applicable to nonlinear problems.

3.8 - Minimizing Weighted Sums of Functions

A standard technique for MOP is to minimize a positively weighted convex sum of the objectives, that is,

$$\sum_{i=1}^{n} \alpha_i f_i(x), \qquad \alpha_i > 0, \quad i = 1, 2, \ldots, n.$$

It is easy to prove that the minimizer of this combined function is Pareto optimal. It is up to the user to choose appropriate weights. Until recently, considerations of computational expense forced users to restrict themselves to performing only one such minimization. Newer, more ambitious approaches aim to minimize convex sums of the objectives for various settings of the convex weights, therefore generating various points in the Pareto set. Though computationally more expensive, this approach gives an idea of the shape of the Pareto surface and provides the user with more information about the trade-off among the various objectives. However, this method suffers from two drawbacks. First, the relationship between the vector of weights and the Pareto curve is such that a uniform spread of weight parameters rarely produces a uniform spread of points on the Pareto set. Often, all the points found are clustered in certain

parts of the Pareto set with no point in the interesting ``middle part'' of the set, thereby providing little insight into the shape of the trade-off curve. The second drawback is that non-convex parts of the Pareto set cannot be obtained by minimizing convex combinations of the objectives (note though that non-convex Pareto sets are seldom found in actual applications).

3.9. Homotopy Techniques

Homotopy techniques aim to trace the complete Pareto curve in the bi-objective case (n=2). By tracing the full curve, they overcome the sampling deficiencies of the weighted-sum approach. The main drawback is that this approach does not generalize to the case of more than two objectives. For more information, see Rao and Papalambros and Rakowska, Haftka, and Watson .

3.10. Goal Programming

In the goal programming approach, we minimize one objective while constraining the remaining objectives to be less than given target values. This method is especially useful if the user can afford to solve just one optimization problem. However, it is not always easy to choose appropriate ``goals'' for the constraints. Goal programming cannot be used to generate the Pareto set effectively, particularly if the number of objectives is greater than two.

3.11. Normal-Boundary Intersection (NBI)

The normal-boundary intersection method uses a geometrically intuitive parametrization to produce an even spread of points on the Pareto surface, giving an accurate picture of the whole surface. Even for poorly scaled problems (for which the relative scalings on the objectives are vastly different), the spread of Pareto points remains uniform. Given any point generated by NBI, it is usually possible to find a set of weights such that this point minimizes a weighted sum of objectives, as described above. Similarly, it is usually possible to define a goal programming problem for which the NBI point is a solution. NBI can also handle problems where the Pareto surface is discontinuous or non-smooth, unlike homotopy techniques. Unfortunately, a point generated by NBI may not be a Pareto point if the boundary of the attained set in the

objective space containing the Pareto points is nonconvex or `folded' (which happens rarely in problems arising from actual applications).

NBI requires the individual minimizers of the individual functions at the outset, which can also be viewed as a drawback.

3.12. Multilevel Programming

Multilevel programming is a one-shot optimization technique and is intended to find just one ``optimal" point as opposed to the entire Pareto surface. The first step in multilevel programming involves ordering the objectives in terms of importance. Next, we find the set of points $x \in C$ for which the minimum value of the first objective function is attained. We then find the points in this set that minimize the second most important objective. The method proceeds recursively until all objectives have been optimized on successively smaller sets.

Multilevel programming is a useful approach if the hierarchical order among the objectives is of prime importance and the user is not interested in the continuous trade-off among the functions. However, problems lower down in the hierarchy become very tightly constrained and often become numerically infeasible, so that the less important objectives have no influence on the final result. Hence, multilevel programming should surely be avoided by users who desire a sensible compromise solution among the various objectives.

3.13. Multi-Objective Optimization

3.13.1. Introduction

Most realistic optimization problems, particularly those in design, require the simultaneous optimization of more than one objective function. Some examples:

In bridge construction, a good design is characterized by low total mass and high stiffness.

Aircraft design requires simultaneous optimization of fuel efficiency, payload, and weight.

In chemical plant design, or in design of a groundwater remediation facility, objectives to be considered include total investment and net operating costs.

A good sunroof design in a car could aim to minimize the noise the driver hears and maximize the ventilation.

The traditional portfolio optimization problem attempts to simultaneously minimize the risk and maximize the fiscal return.

In these and most other cases, it is unlikely that the different objectives would be optimized by the same alternative parameter choices. Hence, some trade-off between the criteria is needed to ensure a satisfactory design.

Multicriteria optimization has its roots in late-nineteenth-century welfare economics, in the works of Edgeworth and Pareto. A mathematical description is as follows:

$$\min_{x \in C} F(x) = \begin{bmatrix} f_1(x) \\ f_2(x) \\ \vdots \\ f_n(x) \end{bmatrix} \qquad \ldots(MOP)$$

where n >= 2 and

$$C = \{x : h(x) = 0,\ g(x) \leq 0,\ a \leq x \leq b\}$$

denotes the feasible set constrained by equality and inequality constraints and explicit variable bounds. The space in which the objective vector belongs is called the objective space and image of the feasible set under F is called the attained set.

The scalar concept of ``optimality" does not apply directly in the multiobjective setting. A useful replacement is the notion of Pareto optimality. Essentially, a vector $x^* \in C$ is said to be Pareto optimal for (MOP) if all other vectors $x \in C$ have a higher value for at least one of the objective functions $f_i(\cdot)$, or else have the same value for all objectives.

Formally speaking, we have the following definition:

A point $x^* \in C$ is said to be (glob ally) Pareto optimal or a (globally) efficient solution or a non-dominated or a non-inferior point for (MOP) if and only if there is no $x \in C$ such that $f_i(x) \leq f_i(x^*)$ for all $i \in \{1, 2, \ldots, n\}$, with at least one strict inequality.

Pareto optimal points are also known as efficient, non-dominated, or non-inferior points.

We can also speak of locally Pareto optimal points, for which the definition is the same as the one just given, except that we restrict attention to a feasible neighborhood of x^*.

That is, if $B(x^*, \delta)$ denotes a ball of radius δ around the point x^*, we require that for some $\delta > 0$, there is no $x \in C \cap B(x^*, \delta)$ such that

$$f_i(x) \leq f_i(x^*), \qquad \text{for all } i = \{1, 2, \ldots, n\}$$

with at least one strict inequality.

Typically, there is an entire curve or surface of Pareto points, whose shape indicates the nature of the tradeoff between different objectives.

The optimization problem is then:

Find values of the variables that minimize or maximize the objective function while satisfying the constraints.

Are All these ingredients necessary?

3.13.2. Objective Function

Almost all optimization problems have a single objective function. (When they don't they can often be reformulated so that they do!) The two interesting exceptions are:

1. No objective function. In some cases (for example, design of integrated circuit layouts), the goal is to find a set of variables that satisfies the constraints of the model. The user does not particularly want to optimize anything so there is no reason to define an objective function. This type of problems is usually called a feasibility problem.
2. Multiple objective functions. Often, the user would actually like to optimize a number of different objectives at once. For instance, in the panel design problem, it would be nice to minimize weight and maximize strength simultaneously. Usually, the different objectives are not compatible; the variables that optimize one objective may be far from optimal for the others. In practice, problems with multiple objectives are reformulated as single-objective problems by either forming a weighted combination of the different objectives or else replacing some of the objectives by constraints. These approaches and others are described in our section on multi-objective optimization.

3.14. Unconstrained Optimization

The unconstrained optimization problem is central to the development of optimization software. Constrained optimization algorithms are often extensions of unconstrained algorithms, while nonlinear least squares and nonlinear equation algorithms tend to be specializations. In the unconstrained optimization problem, we seek a local minimizer of a real-valued function, f(x), where x is a vector of real variables. In other words, we seek a vector, x*, such that f(x*) <= f(x) for all x close to x*.

Global optimization algorithms try to find an x* that minimizes f over all possible vectors x. This is a much harder problem to solve. We do not discuss it here because, at present, no efficient algorithm is known for performing this task. For many applications, local minima are good enough, particularly when the user can draw on his/her own experience and provide a good starting point for the algorithm.

Newton's method gives rise to a wide and important class of algorithms that require computation of the gradient vector

$$\Delta f(\chi) = \begin{pmatrix} a_1 f(\chi) \\ \vdots \\ a_n f(\chi) \end{pmatrix}$$

and the Hessian matrix,

$$\nabla^2 f(\chi) = \left(a_j a_i f(\chi)\right)$$

Although the computation or approximation of the Hessian can be a time-consuming operation, there are many problems for which this computation is justified. We describe algorithms in which the user supplies the Hessian explicitly before moving on to a discussion of algorithms that don't require the Hessian.

Newton's method forms a quadratic model of the objective function around the current iterate χ_k. The model function is defined by

$$q_k(8) = f(\chi_k) + \nabla f(\chi_k)^{\mathrm{T}} 8 + \frac{1}{2} 8^{\mathrm{T}} \nabla^2 f(\chi_k) 8.$$

In the basic Newton method, the next iterate is obtained from the minimizer of q_k. When the Hessian matrix, $\nabla^2 f(\chi_k)$, is positive definite, the quadratic model has a unique minimizer that can be obtained by solving the symmetric $n \times n$ linear system:

$$\nabla^2 f(\chi_k) 8_k = -\nabla f(\chi_k)$$

The next iterate is then

$$\chi_{k+1} = \chi_k + 8_k$$

Convergence is guaranteed if the starting point is sufficiently close to a local minimizer x* at which the Hessian is positive definite. Moreover, the rate of convergence is quadratic, that is,

$$\| x_{k+1} - x^* \| \leq \beta \| x_k - x^* \|^2,$$

for some positive constant β.

In most circumstances, however, the basic Newton method has to be modified to achieve convergence.

These codes obtain convergence when the starting point is not close to a minimizer by using either a **line-search** or a **trust-region** approach.

The **line-search variant** modifies the search direction to obtain another a downhill, or descent direction for f. It then tries different step lengths along this direction until it finds a step that not only decreases f, but also achieves at least a small fraction of this direction's potential.

The **trust-region variant** uses the original quadratic model function, but they constrain the new iterate to stay in a local neighborhood of the current iterate. To find the step, then, we have to minimize the quadratic subject to staying in this neighborhood, which is generally ellipsoidal in shape.

Line-search and trust-region techniques are suitable if the number of variables n is not too large, because the cost per iteration is of order n^x. Codes for problems with a large number of variables tend to use truncated Newton methods , which usually settle for an approximate minimizer of the quadratic model.

So far, we have assumed that the Hessian matrix is available, but the algorithms are unchanged if the Hessian matrix is replaced by a reasonable approximation. Two kinds of methods use approximate Hessians in place of the real thing:

The first possibility is to use difference approximations to the exact Hessian. We exploit the fact that each column of the Hessian can be approximated by taking the difference between two instances of the gradient vector evaluated at two nearby points. For sparse Hessians, we can often approximate many columns of the Hessian with a single gradient evaluation by choosing the evaluation points judiciously.

Quasi-Newton Methods build up an approximation to the Hessian by keeping track of the gradient differences along each step taken by the algorithm. Various conditions are imposed on the approximate Hessian. For example, its behavior along the step just taken is forced to mimic the behavior of the exact Hessian, and it is usually kept positive definite.

Finally, we mention two other approaches for unconstrained problems that are not so closely related to Newton's method:

Nonlinear conjugate gradient methods are motivated by the success of the linear conjugate gradient method in minimizing quadratic functions with positive definite Hessians. They use search directions that combine the negative gradient direction with another direction, chosen so that the search will take place along a direction not previously explored by the algorithm. At least, this property holds for the quadratic case, for which the minimizer is found exactly within just n iterations. For nonlinear problems, performace is problematic, but these methods do have the advantage that they require only gradient evaluations and do not use much storage.

The nonlinear Simplex method (not to be confused with the simplex method for linear programming) requires neither gradient nor Hessian evaluations. Instead, it performs a pattern search based only on function values. Because it makes little use of information about f, it typically requires a great many iterations to find a solution that is even in the ballpark. It can be useful when f is nonsmooth or when derivatives are impossible to find, but it is unfortunately often used when one of the algorithms above would be more appropriate.

3.15. NON-LINEAR OPTIMIZATION

3.15.1. Non-linear optimization

The general constrained optimization problem is to minimize a nonlinear function subject to nonlinear constraints. Two equivalent formulations of this problem are useful for describing algorithms. They are

$$\min\{f(x): c_i(x) \le 0, i \in \mathrm{T}, c_i(x) = 0, i \in \varepsilon\} \qquad (3.1)$$

Where each c_i is a mapping from $\Re^n$ to $\Re$, and T and ε are index sets for inequality and equality constraints, respectively; and

$$\min\{f(x); c(x) = 0, l \le x \le u\} \qquad (3.2)$$

Where c maps $\Re^n$ to $\Re^m$, and the lower- and upper-bound vectors, l and u , may contain some infinite components. The main techniques that have been proposed for solving constrained optimization problems are reduced-gradient methods, sequential linear and quadratic programming methods, and methods based on augmented Lagrangians and exact penalty functions. Fundamental to the understanding of these algorithms is the Lagrangian function, which for formulation (3.1) is defined as:

$$(x, \lambda)\ L = f(x) + \sum_{i \in \pi \varepsilon} \lambda_i c_i(x)$$

The Lagrange is used to express first-order and second-order conditions for a local minimizer. We simplify matters by stating just first-order necessary and second-order sufficiency conditions without trying to make the weakest possible assumptions. The first-order necessary conditions for the existence of a local minimizer x^* of the constrained optimization problem (3.1) require the existence of Lagrange multipliers λ_i^*, such that

$$\nabla_x \mathrm{L}(x^*, \lambda^*) = \nabla f(x^*) + \sum_{i \in A^*} \lambda_i^* \nabla c_i(x^*) = 0$$

Where, $A^* = \{i \in T : c_i(x^*) = 0\} \cup \in$

Is the active set at x^* , and $\lambda_i^* \ge 0$ if $i \in A^* \cap I$. This result requires a constraint qualification to ensure that the geometry of the feasible set is adequately captured by a linearization of the constraints about $x*$. A standard constraint qualification requires the constraint normal, $\nabla c_i(x^*)$ for $i \in A^*$, to be linearly independent.

The second-order sufficiency condition requires that (x^*, λ^*) satisfies the first-order condition and that the Hessian of the Lagrangian

$$\nabla^2{}_{xx}L(x^*, \lambda^*) = \nabla^2 f(x^*) + \sum_{i \in A^*} \lambda_I^* \nabla^2 c_i(x^*)$$

Satisfies the condition

$$w^T \nabla^2{}_{XX} L(x^*, \lambda^*) \omega \rangle 0$$

For all nonzero ω in the set

$$\left\{ w \in \Re^n : \nabla c_i(x^*)^T \omega = 0, i \varepsilon T^* + \cup \varepsilon_1 \nabla c_i(x^*)^T \omega \leq 0, i \in T_0^* \right\}$$

Where

$$T_+^* = \left\{ i \in A^* \cap T : \lambda_i^* \rangle 0 \right\} T_0 = \left\{ i \in A^* \cap T : \lambda_I^* = 0 \right\}$$

The previous condition guarantees that the optimization problem is well behaved near x^*; in particular, if the second-order sufficiency condition holds, then x^* is a strict local minimizer of the constrained problem. An important ingredient in the convergence analysis of a constrained algorithm is its behavior in the vicinity of a point (x^*, λ^*) that satisfies the second-order sufficiency condition.

3.15.2. The sequential quadratic programming algorithm

It is a generalization of Newton's method for unconstrained optimization in that it finds a step away from the current point by minimizing a quadratic model of the problem. A number of packages, including NPSOL, NLPQL, OPSYC, OPTIMA, MATLAB, and SQP, are founded on this approach. In its purest form, the sequential QP algorithm replaces the objective function with the quadratic approximation

$$q_k(d) = \nabla f(x_k)^T d + \frac{1}{2} d^T \nabla^2{}_{xx} L(x_k, \lambda_k)^d$$

and replaces the constraint functions by linear approximations. For the formulation (3.1), the step d_k is calculated by solving the quadratic subprogram

$$\min\{q_k(d): c_i(x_k)+\nabla c_i(x_k)^T d \le 0, i \in T$$

$$c_i(x_k)+\nabla(x_k)^T d = 0, i \in \varepsilon \} \qquad (3.3)$$

The local convergence properties of the sequential QP approach are well understood when (x^*, λ^*) satisfies the second-order sufficiency conditions. If the starting point x_0 is sufficiently close to x^*, and the Lagrange multiplier estimates $\{\lambda_k\}$ remain sufficiently close to λ^*, then the sequence generated by setting $x_{k+1} = x_k + d_k$ converges to x^* at a second-order rate. These assurances cannot be made in other cases. Indeed, codes based on this approach must modify the sub-problem (3.3) when the quadratic q_k is unbounded below on the feasible set or when the feasible region is empty.

The Lagrange multiplier estimates that are needed to set up the second-order term in q_k can be obtained by solving an auxiliary problem or by simply using the optimal multipliers for the quadratic sub-problem at the previous iteration. Although the first approach can lead to more accurate estimates, most codes use the second approach.

The strategy based on (3.3) makes the decision about which of the inequality constraints appear to be active at the solution internally during the solution of the quadratic program. A somewhat different algorithm is obtained by making this decision prior to formulating the quadratic program. This variant explicitly maintains a working set W_k of apparently active indices and solves the quadratic programming problem

$$\min\{q_k(d): c_i(x_k)+\nabla c_i(x_k)^T d = 0, i \in W_k \} \quad (3.4)$$

To find the step d_k. The contents of W_k updated at each iteration by examining the Lagrange multipliers for the sub-problem (3.4) and by examining the values of $c_i(x_k+1)$ at the new iterate x_k+1 for $i \notin W_k$. This approach is usually called the EQP (equality-based QP) variant of sequential QP, to distinguish it from the IQP (inequality-based QP) variant described above.

The sequential QP approach outlined above requires the computation of $\nabla^2_{xx}L(x_k, \lambda_k)$. Most codes replace this matrix with the BFGS approximation B_k, which is updated at each iteration. An obvious update strategy (consistent with the BFGS update for unconstrained optimization) would be to define

$$S_K = x_k + 1 - x_k, \qquad Y_k = \nabla_x L(x_k + 1, \lambda_k) - \nabla_x L(x_k, \lambda_k)$$

and update the matrix B_k by using the BFGS formula

$$B_{k+1} = B_k - \frac{B_k 8_k 8_k^T B_k}{8_k^T B_k 8_k} + \frac{Y_k Y_k^T}{Y_k^T 8_k}$$

However, one of the properties that make Broyden-class methods appealing for unconstrained problems-its maintenance of positive definiteness in B_k is no longer assured, since $\nabla^2_{xx}L(x^*, \lambda^*)$ is usually positive definite only in a subspace. This difficulty may be overcome by modifying Y_k. Whenever $Y_k^T 8_k$ is not sufficiently positive, Y_k is reset to

$$Y_k \leftarrow \theta_k Y_k + (1 - \theta_k) B_K 8_K,$$

Where $\theta_k \in [0,1)$ is the number closest to 1 such that $Y_k^T 8_k \geq \sigma 8_k^T B_k 8_k$ for some $\sigma \in (0,1)$. The SQP and NLPQL codes use an approach of this type.

The convergence properties of the basic sequential QP algorithm can be improved by using a line search. The choice of distance to move along the direction generated by the sub-problem is not as clear as in the unconstrained case, where we simply choose a step length that approximately minimizes f along the search direction. For constrained problems we would like the next iterate not only to decrease f but also to come closer to satisfying the constraints. Often these two aims conflict, so it is necessary to weigh their relative importance and define a merit or penalty function, which we can use as a

criterion for determining whether or not one point is better than another. The l_1 merit function

$$P_1(x;v) = f(x) + \sum_{i\in\varepsilon} v_i |c_i(x)| + \sum_{i\in T} v_i \max(c_i(x),0), \qquad (3.5)$$

Where $v_i \rangle 0$ are penalty parameters, is used in the NLPQL, MATLAB, and SQP codes, while the augmented Lagrangian merit function

$$L_A(x,\lambda;v) = f(x) + \sum_{i\in\varepsilon} \lambda_i c_i(x) + \frac{1}{2}\sum_{i\in\varepsilon} v_i c_i^2(x) + \frac{1}{2}\sum_{i\in T} \psi_i(x,\lambda;v),$$

Where

$$\psi_i(x,\lambda;v) = \frac{1}{v_i}\{\max\{0, \lambda_i + v_i c_i(x)\}^2 - \lambda_i^2\},$$

Is used in the NLPQL, NPSOL, and OPTIMA codes. The OPSYC code for equality-constrained problems (for which $T = \Phi$) uses the merit function

$$f(x) + \sum_{i\in\varepsilon} \lambda_i c_{i(x)} + \Big(\sum_{i\in\varepsilon} v_i c_i^2(x) \Big)^{\frac{1}{2}}$$

Which combines features of P_1 and L_A.

An important property of the l_1 merit function is that if (x^*, λ^*) satisfies the second-order sufficiency condition, then x^* is a local minimizer of P_1, provided the penalty parameters are chosen so that $v_i \rangle |\lambda_i^*|$. Although this is an attractive property, the use of P_1 requires care. The main difficulty is that P_1 is not differentiable at any x with $c_i(x) = 0$. Another difficulty is that although x^* is a local minimizer of P_1, it is still possible for the function to be unbounded below. Thus, minimizing P_1 does not always lead to a solution of the constrained problem.

The merit function L_A has similar properties. If (x^*, λ^*) satisfies the second-order sufficiency condition and $\lambda = \lambda^*$, then x is a local minimizer of P_1, provided the

penalty parameters v_i are sufficiently large. If $\lambda \neq \lambda^*$, then we can say only that L_A has a minimizer $x(\lambda)$ near x^* and that $x(\lambda)$ approaches x^* as λ converges to λ^*. Note that in contrast to P_1, the merit function L_A is differentiable. The Hessian matrix of L_A is discontinuous at any x with $\lambda_i + v_i c_i(x) = 0$ for $i \in \mathrm{T}$, but, at least in the case $\mathrm{T}_0^* = \Phi$, these points tend to occur far from the solution.

The use of these merit functions by NLPQL is typical of other codes. Given an iterate x_k and the search direction d_k, NLPQL sets $x_{k+1} = x_k + \alpha_k d_k$, where the step length α_k approximately minimizes $(x_k + \alpha d_k; v)$. If the merit function L_A is selected, the step length α_k is chosen to approximately minimize $L_A(x_k + \alpha d_k, \lambda_k + \alpha(\lambda_{k+1} - \lambda_k); v)$, where d_k is a solution of the quadratic programming sub-problem (3.3) and λ_{k+1} is the associated Lagrange multiplier.

3.15.3. Augmented Lagrangian algorithms

These are based on successive minimization of the augmented Lagrangian L_A with respect to x, with updates of λ and possibly v occurring between iterations. An augmented Lagrangian algorithm for the constrained optimization problem computes x_{k+1} as an approximate minimizer of the sub-problem

$$\min\{L_A(x, \lambda_k; v_k) : l \leq x \leq u\},$$

Where $L_A(x, \lambda; v) = f(x) + \sum_{i \in \varepsilon} \lambda_i c_i(x) + \frac{1}{2} \sum_{i \in \varepsilon} v_i c_i^2(x)$

Includes only the equality constraints. Updating of the multipliers usually takes the form

$$\lambda_i \leftarrow \lambda_i + v_i c_i(x_k)$$

This approach is relatively easy to implement because the main computational operation at each iteration is minimization of the smooth function L_A with respect to x, subject only to bound constraints. A large-scale implementation of the augmented Lagrangian approach can be found in the LANCELOT package, which solves the bound-constrained sub-problem by using special data structures to exploit the (group partially

separable) structure of the underlying problem. The OPTIMA and OPTPACK libraries also contain augmented Lagrangian codes.

3.15.3. Reduced-gradient algorithms

These avoid the use of penalty parameters by searching along curves that stay near the feasible set. Essentially, these methods take the formulation (3.2) and use the equality constraints to eliminate a subset of the variables, thereby reducing the original problem to a bound-constrained problem in the space of the remaining variables. If x_B is the vector of eliminated or basic variables, and x_N is the vector of no basic variables, then

$$x_N = h(x_N),$$

Where the mapping h is defined implicitly by the equation

$$c[h(x_N), x_N] = 0$$

(We have assumed that the components of have been arranged so that the basic variables come first.) In practice, $x_B = h(x_N)$

Can be recalculated using Newton's method whenever x_N changes. Each Newton iteration has the form

$$x_B \leftarrow x_B - \partial_B c(x_B, x_N)^{-1} c(x_B, x_N),$$

Where $\partial_B c$ is the Jacobian matrix of c with respect to the basic variables. The original constrained problem is now transformed into the bound-constrained problem.

$$\min\{f(h(x_N), x_N) : l_N \le x_N \le u_N\}$$

Algorithms for this reduced sub-problem subdivide the no basic variables into two categories. These are the fixed variables x_F, which usually include most of the variables that are at either their upper or lower bounds and that are to be held constant on the current iteration, and the super basic variables x_S , which are free to move on

this iteration. The standard reduced-gradient algorithm, implemented in CONOPT, searches along the steepest-descent direction in the super basic variables. The generalized reduced-gradient codes GRG2 and LSGRG2 use more sophisticated approaches. They either maintain a dense BFGS approximation of the Hessian of f with respect to x_S or use limited-memory conjugate gradient techniques. MINOS also uses a dense approximation to the super basic Hessian matrix. The main difference between MINOS and the other three codes is that MINOS does not apply the reduced-gradient algorithm directly to problem (3.1), but rather uses it to solve a linearly constrained sub-problem to find the next step. The overall technique is known as a projected augmented Lagrangian algorithm.

Operations involving the inverse of $\partial_B c(x_B, x_N)$ are frequently required in reduced-gradient algorithms. These operations are facilitated by an LU factorization of the matrix. GRG2 performs a dense factorization, while CONOPT, MINOS, and LSGRG2 use sparse factorization techniques, making them more suitable for large-scale problems.

When some of the components of the constraint functions are linear, most algorithms aim to retain feasibility of all iterates with respect to these constraints. The optimization problem becomes easier in the sense that there is no curvature term corresponding to these constraints that must be accounted for and, because of feasibility; these constraints make no contribution to the merit function. Numerous codes, such as NPSOL, MINOS and some routines from the NAG (NAG Fortran or NAG C) library, are able to take advantage of linearity in the constraint set. Other codes, such as those in the IMSL, PORT 3, and PROC NLP libraries, are specifically designed for linearly constrained problems. The IMSL codes are based on a sequential quadratic programming algorithm that combines features of the EQP and IQP variants. At each iteration, this algorithm determines a set N_k of near-active indices defined by:

$$N_K = \{i \in \mathrm{T} : c_i(x_K) \geq -r_i\},$$

Where the tolerances r_i tend to decrease on later iterations. The step d_K is obtained by solving the sub-problem.

$$\min\{q_K(d) : c_i(x_K + d_K) = 0, i \in \varepsilon, c_i(x_K + d_K) \leq c_i(x_K), i \in N_K\},$$

Where $$q_K(d) = \nabla f(x_K)^T d + \frac{1}{2} d^T B_K d,$$

And B_K is a BFGS approximation to $\nabla^2 f(x_K)$. This algorithm is designed to avoid the short steps that EQP methods sometimes produce, without taking many unnecessary constraints into account, as IQP methods do.

3.13.4 - Feasible sequential quadratic programming algorithms

Finally, we mention feasible sequential quadratic programming algorithms, which, as their name suggests, constrain all iterates to be feasible. They are more expensive than standard sequential QP algorithms, but they are useful when the objective function f is difficult or impossible to calculate outside the feasible set, or when termination of the algorithm at an infeasible point (which may happen with most algorithms) is undesirable. The code FSQP solves problems of the form

$$\min\{f(x) : c(x) \leq 0, Ax = b\}$$

In this algorithm, the step is defined as a combination of the sequential QP direction, a strictly feasible direction (which points into the interior of the feasible set) and, possibly, a second-order correction direction. This mix of directions is adjusted to ensure feasibility while retaining fast local convergence properties. Feasible algorithms have the additional advantage that the objective function f can be used as a merit function, since, by definition, the constraints are always satisfied. FSQP also solves problems in which f is not itself smooth, but is rather the maximum of a finite set of smooth functions

$$f_i : \Re^n \to \Re$$

CHAPTER FOUR

GENETIC ALGORITHMS BASED ON OPTIMIZATION

4.1 Optimization based on Genetic Algorithms

Genetic algorithms were formally introduced in the United States in the 1970s by John Holland at University of Michigan. The continuing price/performance improvements of computational systems have made them attractive for some types of optimization. In particular, genetic algorithms work very well on mixed (continuous and discrete), combinatorial problems. They are less susceptible to getting 'stuck' at local optima than gradient search methods. But they tend to be computationally expensive.

To use a genetic algorithm, you must represent a solution to your problem as a genome (or chromosome). The genetic algorithm then creates a population of solutions and applies genetic operators such as mutation and crossover to evolve the solutions in order to find the best one.

This presentation outlines some of the basics of genetic algorithms. The three most important aspects of using genetic algorithms are:

Definition of the objective function.

Definition and implementation of the genetic representation.

Definition and implementation of the genetic operators, Once these three have been defined, the generic genetic algorithm should work fairly well. Beyond that you can try many different variations to improve performance, find multiple optima (species - if they exist), or parallelism the algorithms,

Genetic algorithm (GA) uses the principles of evolution, natural selection, and genetics from natural biological systems in a computer algorithm to simulate evolution. Essentially, the genetic algorithm is an optimization technique that performs a parallel, stochastic, but directed search to evolve the most fit population. In this section we will introduce the genetic algorithm and explain how it can be used for design of fuzzy systems. The genetic algorithm borrows ideas from and attempts to simulate Darwin's

theory on natural selection and Mendel's work in genetics on inheritance. The genetic algorithm is an optimization technique that evaluates more than one area of the search space and can discover more than one solution to a problem. In particular, it provides a stochastic optimization method where if it "gets stuck" at a local optimum, it tries to simultaneously find other parts of the search space and jump out" of the local optimum to a global one.

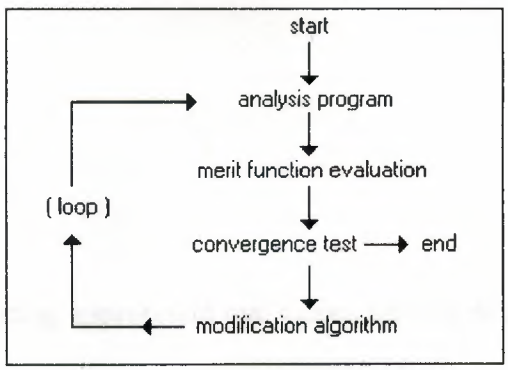
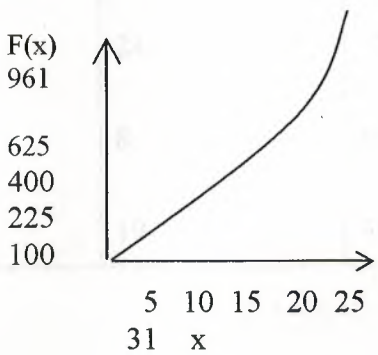


Figure 4.1 Characteristics common to all optimizers

4.2. Main Features for Optimization

1. probabilistic algorithms for searching and optimization
2. mimic natural evolution process
3. capable of handling nonlinear, non-convex problem
4. optimization procedure does not require differentiation operation
5. capable of locating global and local optima within search domain
6. Optimization is based on population instead of a single point.

let's consider a simple problem of maximization of the function F(x)=x2 , where x∈[0,31] (see Fig).



In order to use GA we should first code variables in to bit strings as any integer number between 0 to 31 may be represented in binary number 5 symbols between (00000)=0 and (11111)=31 the length of chromosomes will be five.

Lets take 4 chromosomes with random set of genes as an initial population as:

01101

11000

01000

10011

then define their fitness inserting appropriate real values into the function as:

f(01000)2 = f(8=64

thus the fitness of all individuals in calculated then accordance with the formula

Pi5 =fi/ΣIj=1Fi5, i = 1..4

Survival probability for each individual is calculated where as cumulatives

Picum =ΣIj=1 Pi 5 i = 1..4

Let's enter all the calculated values in table

Initial Population	Their integer values	F(x)=x2	P5	Pcum	Num after Selection
01101	13	169	0.14	0.14	1
11000	24	576	0.49	0.63	2
01000	8	64	0.06	0.69	0
10011	19	361	0.31	1	1

Average	293	0.25	1
Maximum	576	0.49	2
Sum	1170	1	4

For the selection process we generate 4 random numbers from the range [0,1]. Suppose, that we generated 0.1; 0.25; 0.5 and 0.8.

Comparing these values with cumulative probabilities, we obtaining the following

$0.1 < Picum$

$PIcum < 0.25 < P2cum$

$P1cum < 0.5 < P2cum$

$P3cum < 0.8 < P4cum$

Looking at the right sides of these inequalities, one can easily see, that the first and the fourth chromosomes have passed the selection, each four taking a place in the new generation, the second chromosome-the most highly fitted has got 2 copies, while the third one did not survive at all.These indices are written in the right column of table 3-1.

Then crossover operation is applied. If the probability of crossover Pc=1 is given, it means that, 4.1=4 chromosomes will participate in crossover process.

Let's choose them at random. Suppose that the first and the second strings mate from crossover point 4, and the third with the forth-from crossover point 2.

0110|1

1100|0

11|000

00|011

Population after selection	Crossover	New Population	Value of X	F(x)=X2
01101	4	01100	12	144
11000	4	11001	25	625
11000	2	11011	27	729
10011	2	10000	16	256

Table 3-1

Average	439
Maximum	729
Sum	1754

Having compared both of the tables we see, for ourselves, that the population fitness is improved and we have come close to the solution. The next recombination operator, mutation, is performed on a bit-by-bit basis. If Pm=0.05 is given, it means that only one of twenty bits in population will be changed: 20· 0.05=1. Suppose, that the third bit of the fourth string undergoes mutation. I.e. x4=10100. Repeating these operations in a finite number of generations we will get the chromosome (11111) corresponding to

problem optimal solution. It is necessary to mention, that GA is especially effective for multi-extreme problems in the global solution search process. For example, if the junction is of the type shown . it is rather difficult to find its global maximum by means of traditional methods. Suppose that the junction is defined as [1]

$$f(x)=x\cdot \sin(10\pi\cdot x)+1.$$

The problem is to find x from the range [-1,2], which maximizes the function f, i.e., to find x0 such that

$$f(x0) \geq f(x), \text{ for all } x\in[-1,2].$$

It is relatively easy to analyse the junction f. The zeros of the first derivative f' should be determined

$$F(x)=\sin(10\pi\cdot x)+ 10\pi\cdot x\cdot \cos(10\pi\cdot x)=0$$

The formula is equivalent to

$$\tan(10\pi\cdot x)= (10\pi\cdot x)$$

It is clear that the above equation has an infinite number of solutions,

$$xi=\frac{2i-1}{20}+\xi_{i,} \qquad \text{for i=1, 2, ...,}$$

$$x0 = 0,$$

$$xi=\frac{2i+1}{20}+\xi_{i,} \qquad \text{for i= 1, -2, ...,}$$

where terms $\xi 1$ represent decreasing sequences of real numbers (for i= 1, 2, ..., and i=-1, -2, ...) approaching zero. Note also that the function f reaches its local maxima for x, it i is an odd integer, and its local minima for x, if i is an even integer .Since the domain of the problem is $x\in[-1,2]$, the function reaches its maximum for

$$X19=\frac{37}{20}+\xi_{19}=1.85+\xi_{19},$$

Where f(x19) is slightly larger than

$$F(1.85)=1.85 \cdot \sin(18\pi+\pi/2)=1.0=2.85.$$

Assume that we wish to construct a genetic algorithm to solve the above problem, I.e., to maximize the function f.

4.2.1. Representation

We use a binary vector as a chromosome to represent real values of the variable x. The length of the vector depends on the required precision, which, in this example, is six places after the decimal point. The domain of the variable x has lingth 3; the precision requirement implies that the range [-1,2] should be divided into at least 3 · 1000000 equal size ranges. This means that 22 bits are required as a binary vector (chromosome):

$$2097152=231 < 3000000 \leq 233 = 4194304$$

the mapping from a binary (b21 b20 b0) string into a real number x from the range [-1,2] is straightforward and is completed in two steps:

Convert the binary string (b21 b20 b0) from the base 2 to base 10:

$$(<b21\ b20\ b0>)2 = \left(\sum_{i=0}^{21} b_i \cdot 2^i\right)_{10} = x$$

Find a corresponding real number x:

$$x = -10 + x \cdot \frac{3}{2^{22}-1},$$

where –1,0 is the left boundary of the domain and 3 is the length of the domain.

For example, a chromosome

(1000101110110101000111)

represents the number 0.637197, since

x'=(1000101110110101000111)3 = 2288967 and

$$x=1.0+2288967 \cdot \frac{3}{4194303} = 0.637197.$$

of course, the chromosomes

(00000000000000000000) and (11111111111111111111)

represent boundaries of the domain-1.0 and 2.0, respectively.

Initial population. The initialization process is very simple; we create a population of chromosomes, where each chromosome is a binary vector of 22 bits. All 22 bits for each chromosome are initialized randomly.

Evaluation function. Evaluation function eval for binary vectors v is equivalent to the function f:

Eval(v)=f(x),

Where the chromosome v represents the real value x.

As noted earlier, the evaluation junction plays the role of the environment, rating potential solutions in terms of their fitness. For example, three chromosomes:

V1=(1000101110110101000111)

V2=(0000001110000000010000)

V3=(1110000000111111000101)

Correspond to values x=0.637197, x=0.958973 and x3=1.627888, respectively.

Consequently, the evaluation function would rate them as follows:

$$Eval(v1)=f(x1)=1.586345$$

$$Eval(v2)=f(x2)=0.078878$$

$$Eval(v3)=f(x3)=2.250650$$

Clearly, the chromosome v3 is the best of the three chromosomes, since its evalution returns the highest value.

During the reproduction phase of the genetic algorithm we would use two classical genetic operators; mutation and crossover.

As mentioned earlier, mutation alters one or more genes (positions in a chromosome) with a probability equal to the mutation rate. Assume that the fifth gene from the v3 chromosome was selected for a mutation. Since the fifth gene in this chromosome is 0, it would be flipped into 1. So the chromosome v3 after this mutation would be

V3=(11101000001111111000101)

This chromosome represents the value x3=1.721638 and f(x3)=-0.082257. This means that this particular mutation resulted in a significant decrease of the value of the chromosome v3 . On the other hand, if the 10th gene was selected for mutation in the chromosome v3 them

V3=(11100000011111111000101)

The corresponding value x3=1.630818 and f(x3)=2.343555, an improve-ment over the original value of f(x3)=2.250650.

Let us illustrate the crossover operator on chromosomes v2 and v3 . Assume that the crossover point was (randomly) selected after the 5th gene:

V2=(00000 | 01110000000010000)

V3=(11100 | 00000111111000101)

The two resulting offspring are

V'2=(00000 | 00000111111000101)

V'3=(11100 | 01110000000010000)

These offspring evaluate to

F(v'2)=f(-0.998113)=0.940865.

F(v'3)=f(1.666028)=2.459245

Note that the second offspring has a better evaluation than both of its parents.

Parameters. For this particular problem we have used the following parameters population size ps=50, probability of crossover pc =0.25, probability of mutation pm =0.01. The following section presents some experimental results for such a genetic system.

Experimental results. we provide the generation number for which we noted an improvement in the evaluation function together with the value of the function. The best chromosome after 150 generations was

Vmax=(11110011010001000000101),

Which corresponds to a value xmax=1.850773

AS exprected, xmax=1.85+ξ and f(xmax) is slightly larger than 2.85.

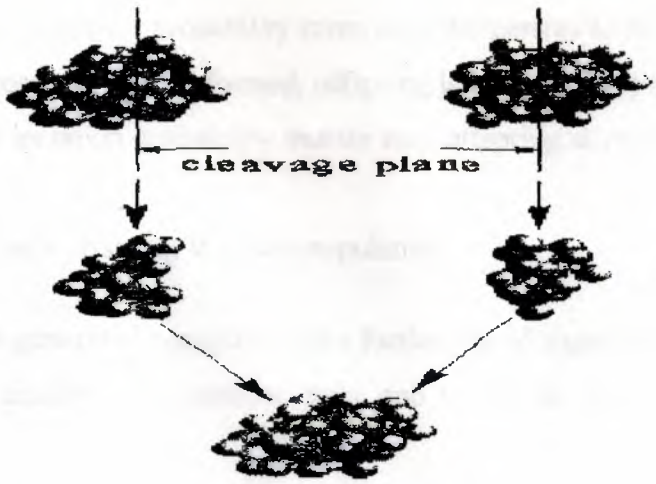
{} → {f(x)} x1t [1] | ∈<≥ · ξ π

Generation Number	Fitness Function
1	1.441942
5	2.250003
8	2.250283
9	2.250284
10	2.250363
12	2.328077
36	2.344251
40	2.345087
51	2.738930
99	2.849246
137	2.850217
145	2.850227

4.3. Genetic Algorithm Structural Optimization

Atomistic models of materials can provide accurate total energies. For problems where the structures are not known, however, discovering the lowest energy geometry is difficult. This is particularly true for atomic clusters, whose structure may vary dramatically with a small change in the number of atoms. For this type of problem, the number of possible stable structures increases exponentially fast with the number of atoms. Furthermore, there is considerable experimental difficulty in determining the structure of an atomic cluster. We have been able to address this problem using a novel approach to applying genetic algorithms. The Darwinian evolution process inspires these algorithms. A population of structures is maintained, and "mating" structures and selecting out the lowest energy geometries produce new generations.

The key to a successful genetic algorithm is to design a mating process that allows for the good parts of the parent structures to be inherited by the next generation. Such a process allows for efficient searching of the possible stable structures. A poor mating algorithm is no better than a random search. We have designed a new mating process, depicted at left. Two structures are chosen as "parent" structures, Each one is divided into two halves by a cleavage plane. A new structure is generated by connecting half of each parent into a new cluster, followed by atomic relaxation to a local minimum. We have successfully applied our "cut and paste" approach to a number of challenging problems, including: (12).



4.4. Genetic Algorithm

4.4.1. Basic Description

Genetic algorithms are inspired by Darwin's theory about evolution. Solution to a problem solved by genetic algorithms is evolved.

Algorithm is started with a set of solutions (represented by chromosomes) called population. Solutions from one population are taken and used to form a new population. This is motivated by a hope, that the new population will be better than the old one. Solutions which are selected to form new solutions (offspring) are selected according to their fitness ,the more suitable they are the more chances they have to reproduce.

This is repeated until some condition (for example number of populations or improvement of the best solution) is satisfied.

4.4.2. Outline of the Basic Genetic Algorithm

1. [Start] Generate random population of n chromosomes (suitable solutions for the problem)
2. [Fitness] Evaluate the fitness f(x) of each chromosome x in the population
3. [New population] Create a new population by repeating following steps until the new population is complete

a. [Selection] Select two parent chromosomes from a population according to their fitness (the better fitness, the bigger chance to be selected)

b. [Crossover] With a crossover probability cross over the parents to form a new offspring (children). If no crossover was performed, offspring is an exact copy of parents.
c. [Mutation] With a mutation probability mutate new offspring at each locus (position in chromosome).
d. [Accepting] Place new offspring in a new population

4. [Replace] Use new generated population for a further run of algorithm
5. [Test] If the end condition is satisfied, stop, and return the best solution in current population
6. [Loop] Go to step 2

Some Comments:

As you can see, the outline of Basic GA is very general. There are many things that can be implemented differently in various problems.

First question is how to create chromosomes, what type of encoding choose. With this is connected crossover and mutation; the two basic operators of GA. Encoding, crossover and mutation are introduced in next chapter.

Next questions are how to select parents for crossover. This can be done in many ways, but the main idea is to select the better parents (in hope that the better parents will produce better offspring). Also you may think, that making new population only by new offspring can cause lost of the best chromosome from the last population. This is true, so so called elitism is often used. This means, that at least one best solution is copied without changes to a new population, so the best solution found can survive to end of run. Maybe you are wandering, why genetic algorithms do work. It can be partially explained by Schema Theorem (Holland), however, this theorem has been criticized in recent time. If you want to know more, check other resources.

CHAPTER FIVE

TRAVELLING SALESMAN PROBLEM

5.1 History Of Travelling Salesman Problem

The traveling salesman problem, or TSP for short, is this: given a finite number of ``cities" along with the cost of travel between each pair of them, find the cheapest way of visiting all the cities and returning to your starting point. .

Mathematical problems related to the traveling salesman problem were treated in the 1800s by the Irish mathematician Sir William Rowan Hamilton and by the British mathematician Thomas Penyngton Kirkman. The picture below is a photograph of Hamilton's Icosian Game that requires players to complete tours through the 20 points using only the specified connections. A nice discussion of the early work of Hamilton and Kirkman can be found in the book Graph Theory 1736-1936 by N. L. Biggs, E. K. LLoyd, and R. J. Wilson, Clarendon Press, Oxford, 1976.

The general form of the TSP appears to be have been first studied by mathematicians starting in the 1930s by Karl Menger in Vienna and Harvard. The problem was later promoted by Hassler Whitney and Merrill Flood at Princeton. A detailed treatment of the connection between Menger and Whitney, and the growth of the TSP as a topic of study can be found in Alexander Schrijver's paper ``On the history of combinatorial optimization (till 1960)".

5.2 Travellin Versus Travelling

Julia Robinson's 1949 paper "On the Hamiltonian Game (A Traveling Salesman Problem)" begins with the sentence: "The purpose of this note is to give a method for solving a problem related to the traveling salesman problem." Although the problem was apparently well known at that time, there does not appear to be any earlier reference in the literature. Solution methods began to appear in papers in the mid-1950s; these early papers used a variity of minor variations of the name traveling salesman problem.

Dantzig, Fulkerson, and Johnson (1954) referred to the "traveling-salesman problem", Heller (1954) used "travelling salesman's problem", and Morton and Land (1955) preferred "the `travelling salesman' problem" (and write that they orginally called it the "laundry van problem"). We will follow Robinson and use "traveling salesman problem". The sixth edition of The Concise Oxford Dictionary offers some support, writing "travelling-bag", "travelling-cap", and "travelling clock" all with two l's, but "traveling salesman" with a single l. The second edition of The Oxford English Dictionary does make this distinction, however, writing "travelling salesman problem" (despite a reference to Dantzig, Fulkerson, and Johnson (1954).

5.3. Traveling Salesman Problem

In the Travelling Salesman Problem, the goal is to find the shortest distance between N different cities. The path that the salesman takes is called a tour.

Testing every possibility for an N city tour would be N! math additions. A 30 city tour would be 2.65 X 10^{32} additions. Assuming 1 billion additions per second, this would take over 8,000,000,000,000,000 years. Adding one more city would cause the number of additions to increase by a factor of 31. Obviously, this is an impossible solution.

A genetic algorithm can be used to find a solution is much less time. Although it probably will not find the best solution, it can find a near perfect solution in less than a minute. There are two basic steps to solving the travelling salesman problem using a GA. First, create a group of many random tours in what is called a population. These tours are stored as a sequence of numbers. Second, pick 2 of the better (shorter) tours parents in the population and combine them, using crossover, to create 2 new solutions children in the hope that they create an even better solution. Crossover is performed by picking a random point in the parent's sequences and switching every number in the sequence after that point.

The idea of Genetic Algorithms is to simulate the way nature uses evolution. The GA uses Survival of the Fittest with the different solutions in the population. The good solutions reproduce to form new and hopefully better solutions in the population, while the bad solutions are removed.

Eventually, the GA will make every solution look identical. This is not ideal. There are two ways around this. The first is to use a very large initial population so that it takes the GA longer to make all of the solutions the same. The second method is mutation. Mutation is when the GA randomly changes one of the solutions. Sometimes a mutation can lead to a better solution that a crossover would not have found.

The difficulty in the TSP using a GA is encoding the solutions. The encoding cannot simply be the list of cities in the order they are travelled. As shown below, the crossover operation will not work. The crossover point is the 3rd number. Every number in parent 1 before the crossover point is copied into the same position in child 1. Then, every number after the crossover point in parent 2 is put into child 1. The opposite is done for child 2.

Parent 1	1 2 3 4 5
Parent 2	3 5 2 1 4
Child 1	1 2 3 1 4
Child 1	3 5 2 4 5

As you can see, the city 1 is used twice and city 5 is missing in child 1. A more complicated form of encoding (or a more complicated crossover) must be used. This form of encoding should ensure that that city adjacencies in the list are preserved from the parents to the children. This method should also realize that the tours 1 2 3 4 5 and 3 2 1 5 4 are the same. The method I have used does both.

Time for some technical jargon.

In my GA, the tours are encoded as a 2-dimensional array (a NxN matrix) of bits that store city adjacencies in both directions. A set bit indicates a city connection. If element [X, Y] is set, then city X connects to city Y. Only 2 bits will be set in every row and column.

Every iteration, a number of tours are chosen from the list of tours. This is the tournament set. The best 2 of these tours from the tournament will be combined to form

2 new tours using crossover. These two new solutions will replace the worst 2 tours from the tournament.

A greedy crossover operation combines the 2 tours to hopefully form 2 better tours. All adjacencies that are shared by the parent are placed in both children. This is done by performing a binary AND on the two parent matrices. When the parents disagree, the children alternate which parent they will get an adjacency from. If an adjacency produces a conflict (city used twice or incomplete tour), then a random city is used instead.

Due to the complexity of the encoding method, no mutation is done.

There is also a graphical display with the program where the current best solution is displayed as a bunch of lines connecting different cities.

CONCLUSION

In this project I researched GENETIC ALGORITHM BASE OPTIMIZATION.

Because GENETIC ALGORITHM is much wide an deep suject. But I only research the small points that is BASED OPTIMIZATION of GENETIC ALGORITHM. By this way I also touched GENETIC ALGORITHM little bit.

I hope that this project will be usefull for both future life and the other people who are interested in GENETIC ALGORITH BASE OPTIMIZATION.

REFERENCES

[1] Wolfgang Banzhaf , Colin Reeves , col Reeves (1999) Foundations of Genetic Algorithms ,New York ,Morgan Kaufmann Publishers.

[2] David E. Goldberg Addison,(January 1989), Optimization and Machine Learning Wesley Pub Co; ISBN: 0201157675

[3] Randy L. Haupt, Sue Ellen Haupt (January 1998) John Wiley & Sons; ISBN: 0471188735

[4] Lawrence Davis, (1991),Handbook of Genetic Algorithms, Van Nostrand Reinhold New York.

[5] Mitsuo Gen, Runwei Cheng (2000) Genetic Algorithms and Engineering Optimization John Wiley & Sons,NewYork: ISBN:0471315311

[6] Nirwan Ansari, Edwin Hou (April 1997) Computational Intelligence for Optimization Kluwer Academic Publishers; ISBN: 0792398386

[7] David E. Goldberg and J. Richardson, (1987), Genetic Algorithms with sharing formultimodal function optimization In J. J. Hillsdale, NJ.

[8] David E. Goldberg, K. Deb, H. Kargupta, and G. Harik, (1993),Rapid, Accurate Optimization of Difficult Problems Using Fast Messy Genetic Algorithms, In J.D. Morgan Kaufmann Publishers.

WEB REFRENCES :

[1] http://www.ieee.org

[2] http://www.computer.org/abstracts

[3] http://www.csc.fi/math_topics/opt/

[4] http://www.csc.fi/math_topics/opt/ohj/

[5] http://www.csc.fi/oppaat/gams/

[6] http://wwwfp.mcs.anl.gov/otc/Guide/OptWeb/continuous/constrained/stochastic/

[7] http://www.doc.ic.ac.uk/~nd/surprise_96/journal/vol1/hmw/article1.html