CHAPTER 5
INTRODUCTION TO DISCRETE VARIABLE OPTIMIZATION

5.1 Introduction

5.1.1 Examples of Discrete Variables

One often encounters problems in which design variables must be selected from among a set of discrete values. Examples of discrete variables include catalog or standard sizes (I beams, motors, springs, fasteners, pipes, etc.), materials, and variables which are naturally integers such as people, gear teeth, number of shells in a heat exchanger and number of distillation trays in a distillation column. Many engineering problems are discrete in nature.

5.1.2 Solving Discrete Optimization Problems

At first glance it might seem solving a discrete variable problem would be easier than a continuous problem. After all, for a variable within a given range, a set of discrete values within the range is finite whereas the number of continuous values is infinite. When searching for an optimum, it seems it would be easier to search from a finite set rather than from an infinite set.

This is not the case, however. Solving discrete problems is harder than continuous problems. This is because of the combinatorial explosion that occurs in all but the smallest problems. For example if we have two variables which can each take 10 values, we have $10*10=10^2=100$ possibilities. If we have 10 variables that can each take 10 values, we have 10^{10} possibilities. Even with the fastest computer, it would take a long time to evaluate all of these. Obviously we need better strategies than just exhaustive search.

5.1.2.1 Example: Standard I Beam Sections

There are 195 standard I beam sections. If we have a structure with 20 different beam sizes, how many combinations of beams are possible?

$$195^{20}=6.3*10^{45}$$

Since the number of grains of sand on the earth is estimated to be around $1*10^{25}$, this is a big number!

5.1.3 Related Discrete Variables

We also need to distinguish between discrete variables and *related discrete variables.* Often two discrete variables are related, i.e. the discrete values are somehow tied to each other. For example, suppose we wish to select a pipe, described by the thickness and diameter, from among a set of standard sizes. This makes thickness and diameter discrete. They will also likely be *related*, because certain values of thickness are matched to certain diameters and vice-versa. In general, as diameters increase, the available values for thickness increase as well. Material properties also represent related discrete variables. When we pick a certain material, the modulus, density, yield strength, etc. are also set. These material properties are

related to each other. We cannot match, for example, the density of aluminum with the modulus for steel. When we have related discrete variables, we have discrete variables that fix the values of several variables at once.

5.2 Discrete Optimization with Branch and Bound

5.2.1 Description of General Branch and Bound Algorithm

A classical method for handling discrete problems is called *Branch and Bound*. The word "branch" refers to a tree structure that is built. The word "bound" refers to an estimate of the objective function which is used to prune the tree. Branch and Bound requires that we have an efficient continuous optimization algorithm (we start by modeling the variables as continuous), which is called many times during the course of developing the discrete solution. Some definitions are in order.

- *Root of the tree:* The starting point where all variables are allowed to be continuous.
- *Node:* Any partial or complete discrete solution. In the example which follows, the number inside the node indicates the order in which it was evaluated.
- *Leaf:* A complete solution in which all discrete variables are fixed at discrete values.
- *Bud Node:* A partial solution that yet might grow further.
- *Branching or expanding a node:* Creating a child node below a bud node. When we do this, we are fixing the values of the next discrete variable.
- *Candidate Discrete Optimum or Incumbent:* The best complete solution we have found so far.
- *Bounding function:* In our case, the value of the objective function with some discrete variables allowed to be continuous. This solution will form a lower bound to the actual solution and to the nodes below it. There is no bounding function for a leaf node, because all values are known and the objective can just be computed.
- *Pruning a node:* Cutting off the branch of the tree for a node whose bound is higher (or even within some percentage) than the best incumbent so far.

The branch and bound strategy works by developing an "upside down" tree structure (see example given Fig. 5.1). Initially, at the root of the tree, we optimize the design to find the continuous solution. This will be our beginning estimate of the objective.

At the next level, only one discrete variable is allowed to take on discrete values: other discrete variables are modeled as continuous. At each level in the tree one more discrete variable is made discrete. The various combinations of values for discrete variables constitute nodes in the tree.

We start by progressing down the tree according to the discrete variable combinations that appear to be the best. At each node, an optimization problem is performed for any continuous variables and those discrete variables modeled as continuous at that node. Assuming we are minimizing, the objective value of this optimization problem becomes a *lower bound* for any branches below that node, i.e. the objective value will underestimate (or, at best, be equal to) the true value of the objective since, until we reach the bottom of the tree, some of the discrete variables are modeled as continuous. Once a solution has been found for which all

discrete variables have discrete values (we reach the bottom of the tree), then any node which has an objective function higher than the solution in hand can be pruned, which means that these nodes don't have to be considered further.

As an example, suppose we have 3 discrete variables: variables 1 and 2 have 3 possible discrete values and variable 3 has 4 possible discrete values. A branch and bound tree would look like Fig. 5.1 given below.

In this tree, “Level 0” represents the root node where all variables are continuous. "Level 1" represents the nodes where variable 1 is allowed to be discrete and variables 2 and 3 are continuous. For "Level 2," variables 1 and 2 are discrete; only variable 3 is continuous. In "Level 3," we are at a leaf node where all variables are discrete. The numbers in the circles show the order in which the nodes were evaluated. The number shown at the upper right of each circle is the optimum objective value for the optimization problem performed at the node. An asterisk means no feasible solution could be found to the optimization problem; a double underscore indicates the node was pruned.

At the first level, three optimizations are performed with variable 1 at its 1st, 2nd and 3rd discrete values respectively (variables 2 and 3 continuous). The best objective value was obtained at node 3. This node is expanded further. Nodes 4, 5, 6 correspond to variable 1 at its 3rd discrete value and variable 2 at its 1st, 2nd and 3rd discrete values respectively, with variable 3 continuous. The best objective value for nodes 1, 2, 4, 5,and 6 is at node 1, so it is expanded into nodes 7, 8, and 9. Now the best objective value among unexpanded nodes is at node 5, so it is expanded. Nodes 10, 11, 12, 13 correspond to variable 1 at its 3rd discrete value, variable 2 at its 2nd discrete value, and variable 3 at its 1st, 2nd, 3rd, and 4th values. The best objective value, 59, is obtained at node 11. This becomes the temporary optimal solution. Any nodes with objectives higher than 59 can automatically be pruned since the objective only increases as we go down the tree from a node and make more and more variables discrete. Thus nodes 2, 7, 8 are pruned. Node 9, however, looks promising, so we expand this node. As is shown, we eventually find at node 16, with variable 1 at its 1st discrete value, variable 2 at its 3rd discrete value, and variable 3 at its 3rd discrete value, a better optimum than we had before. At this point all further nodes are pruned and we can stop. In this example we are using a *best first* strategy, meaning we always expand the node which has the lowest bound.

It should be clear that Branch and Bound gains efficiency by pruning nodes of the tree which have higher bounds than known discrete solutions. In realistic problems, the majority of nodes are pruned—in the example which follows, less than 0.001% of the nodes were expanded.

However, there is a cost—at each node we have to perform an optimization, which could involve many calls to the analysis program. One way to reduce this cost is by making a linear approximation to the actual problem and optimizing the linear approximation. This greatly reduces the number of analysis calls, but at some expense in accuracy.

Another way to reduce the size of the tree is to select discrete value neighborhoods around the continuous optimum. It is likely the discrete solution for a particular variable is close to

the continuous one. Selecting neighborhoods of the closest discrete values makes it possible to further restrict the size of the problem and reduce computation.

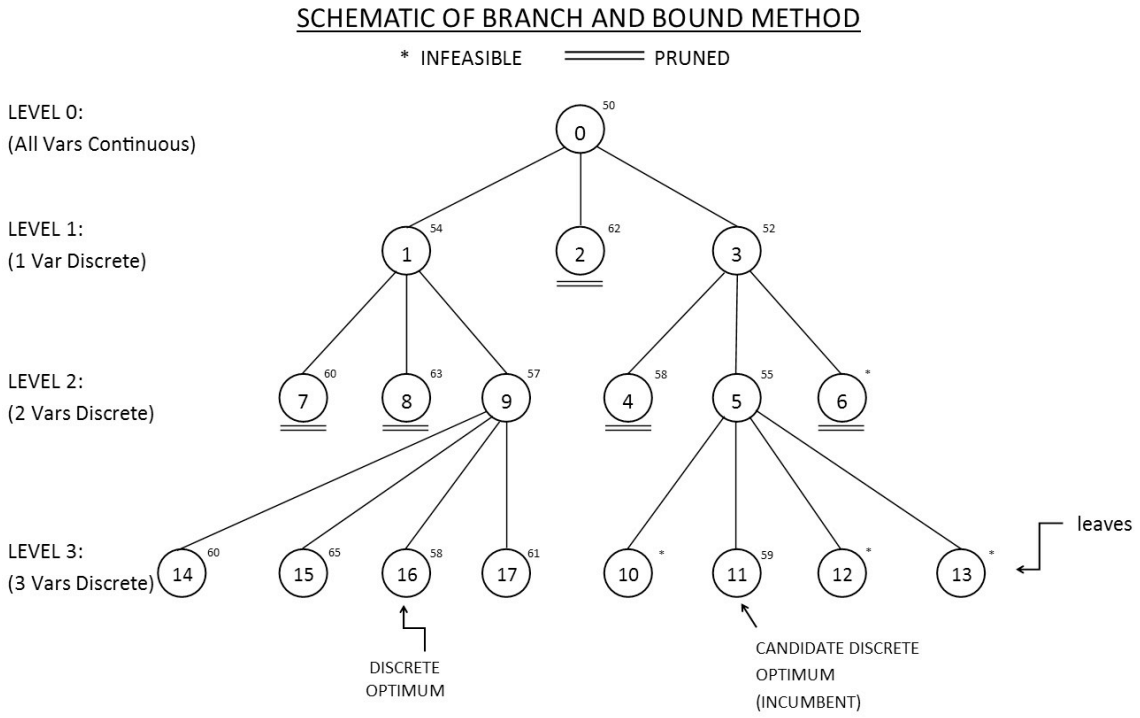

Fig. 5.1 An example Branch and Bound tree expanded with a best first strategy.

5.2.2 Branch and Bound with Integer Constraints

The previous problem description was for general mixed discrete problems. A subset of these is a mixed problem with integer constraints, meaning the discrete variables are constrained to be integers. An approach to this problem with Branch and Bound is show in Fig. 5.2 below.

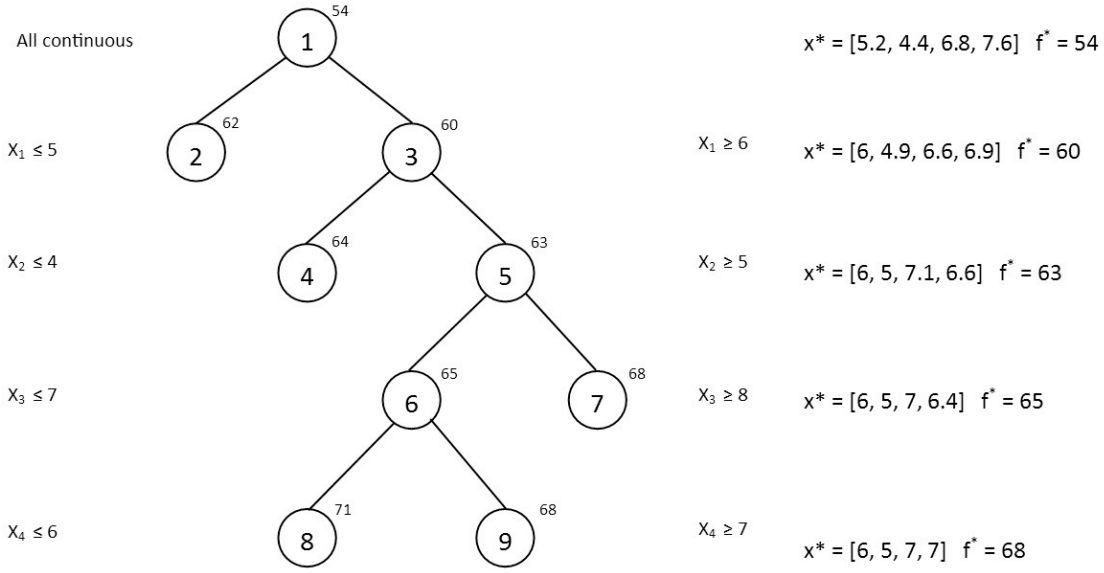

Fig. 5.2. Branch and Bound with integer variables expanded using a depth first strategy.

Note that in this example, we are using a *depth first* strategy, meaning we expand the best node at each level, without looking to see if there are nodes with better bounds at previous levels. At each level we have modeled the requirement that the variable be an integer with two constraints (e.g. $x_1 \le 5;\ x_1 \ge 6$), as opposed to just fixing the values of the variables (at either 5 or 6). These constraints allow the variable in question to assume any value except in the interval between two consecutive integers and thus do not artificially restrict the search space. In fact, it is possible that a variable currently at an integer value could assume a continuous value as additional constraints are added for other variables further down the tree, and the method needs to be able to accommodate this situation.

5.3 Exhaustive Search

The exhaustive search strategy examines all possible combinations of discrete variables to locate the discrete optimum. In terms of a branch and bound tree, exhaustive search examines those nodes found at the bottom of an un-pruned branch and bound tree structure. If the branch and bound strategy is performed without much pruning, more nodes could be evaluated and thus more time required than exhaustive search. In the above example, 17 nodes were evaluated with branch and bound; exhaustive search would have required the evaluation of 3*3*4 = 36 nodes. The efficiency of branch and bound relative to exhaustive search therefore depends on how much pruning is done.

5.4 Example: Design of 8 Story 3 Bay Planar Frame

Prof. Richard Balling applied Branch and Bound to the design of a 2D structure, given in Fig. 5.3. The objective was to minimize weight, subject to AISC combined stress constraints.

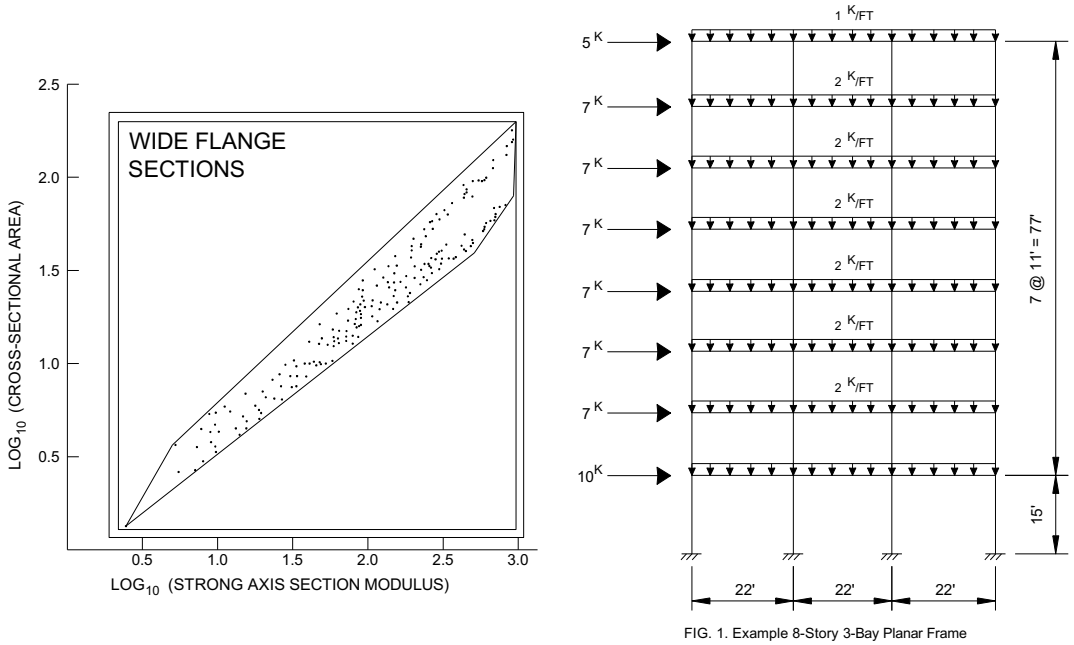

Fig. 5.3 Schematic of eight story 2D frame, and graph showing band of possible discrete values.

Each beam can take one of 195 possible sizes. It was assumed the same size girder continues across each floor. The same size column continues for two stories. Exterior columns were assumed the same size; interior columns were also assumed the same size. These assumptions resulted in 16 separate members; for each member we needed to specify A, I, S, giving a total of 48 design variables (but only 16 related discrete variables). The constraints represented combined stress constraints for each member.

For the starting design all members were W12x65. The weight was 58,243 lbs. There were 13 violated stress constraints.

Step 1: The discrete files were created and the problem set up.

Step 2: A continuous optimization was performed. Using GRG, an optimum was achieved after 9 iterations and 486 analyses. The weight was 36,253 lbs. There were 20 binding stress constraints.

Step 3: Discrete neighborhoods were chosen to reduce problem size. Neighborhoods were sized so that each member had 3 to 5 discrete combinations.

Step 4: Linearization was done to further reduce computation.

Step 5: Branch and bound was executed. The optimum required 3066 linear optimizations which represented only 0.001% of the possible nodes. The discrete optimum weighed 40,337 lbs. and had 5 binding stress constraints.

5.5 Simulated Annealing

5.5.1 Introduction

Branch and Bound and Exhaustive Search both suffer from a serious problem—as the number of variables increases, the number of combinations to be examined explodes. Clearly this is the case for Exhaustive Search, which does nothing to reduce the size of the problem. The same is true for Branch and Bound, although to a lesser degree, because of the pruning and approximations which are employed.

In general then, algorithms which try to search the entire combinatorial space can easily be overwhelmed by the shear size of the problem. In contrast, the *evolutionary algorithms* we study in this and the following sections do not suffer from this problem. These algorithms are so-named because they mimic natural processes that govern how nature evolves. These algorithms do not attempt to examine the entire space. Even so, they have been shown to provide good solutions.

Simulated annealing copies a phenomenon in nature—the annealing of solids—to optimize a complex system. Annealing refers to heating a solid to a liquid state and then cooling it slowly so that thermal equilibrium is maintained. Atoms then assume a nearly globally minimum energy state. In 1953 Metropolis et al. [22] created an algorithm to simulate the annealing process. The algorithm simulates a small random displacement of an atom that

results in a change in energy. If the change in energy is negative, the energy state of the new configuration is lower and the new configuration is accepted. If the change in energy is positive, the new configuration has a higher energy state; however, it may still be accepted according to the Boltzmann probability factor:

$$P = \exp\left(\frac{-\Delta E}{k_b T}\right) = e^{\left(-\Delta E / k_b T\right)} \tag{5.1}$$

where k_b is the Boltzmann constant and T is the current temperature. By examining this equation we should note two things: the probability is proportional to temperature--as the solid cools, the probability gets smaller; and inversely proportional to ΔE --as the change in energy is larger the probability of accepting the change gets smaller.

When applied to engineering design, an analogy is made between energy and the objective function. The design is started at a high "temperature," where it has a high objective (we assume we are minimizing). Random perturbations are then made to the design. If the objective is lower, the new design is made the current design; if it is higher, it may still be accepted according the probability given by the Boltzmann factor. The Boltzmann probability is compared to a random number drawn from a uniform distribution between 0 and 1; if the random number is smaller than the Boltzmann probability, the configuration is accepted. This allows the algorithm to escape local minima.

As the temperature is gradually lowered, the probability that a worse design is accepted becomes smaller. Typically at high temperatures the gross structure of the design emerges which is then refined at lower temperatures.

Although it can be used for continuous problems, simulated annealing is especially effective when applied to combinatorial or discrete problems. Although the algorithm is not guaranteed to find the best optimum, it will often find near optimum designs with many fewer design evaluations than other algorithms. (It can still be computationally expensive, however.) It is also an easy algorithm to implement.

Fig. 5.4 below shows how the weight of a structure changed as simulated annealing was used to select from among discrete structural members. Each cycle represents a temperature. It can be seen that in earlier cycles worse designs were accepted more often than in later cycles.

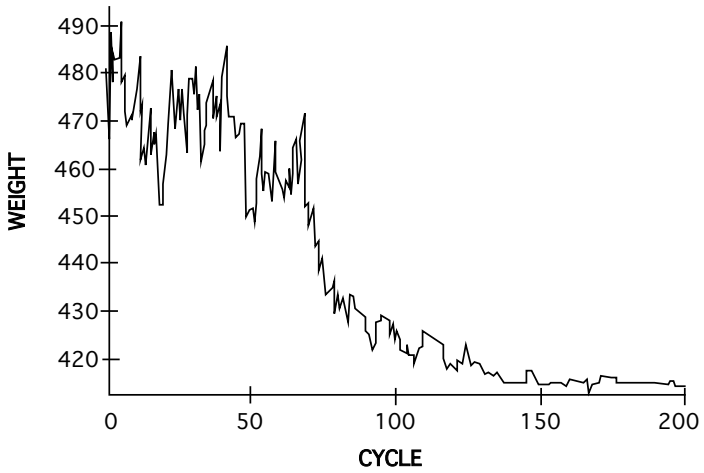

Fig. 5.4 Change in weight of structure during simulated annealing.

The analogy between annealing and simulated annealing is summarized in the table below.

Annealing objective – minimum energy configuration	Simulated annealing objective - minimum cost
Annealing Boltzmann equation $P = \exp\left(\frac{-\Delta E}{k_B T}\right) = e^{\left(-\Delta E / k_b T\right)}$	Simulated annealing Boltzmann equation $P = \exp\left(\frac{-\Delta E}{\Delta E_{avg} T}\right) = e^{\left(-\Delta E / \Delta E_{avg} T\right)}$
P is the probability that an atom will move from a lower to a higher energy state	P is the probability that a higher cost design will be accepted
ΔE is the change in energy to go from a lower energy state to a higher one	ΔE is the cost difference between the current design and the previous one
T is the absolute current annealing temperature; it correlates to the amount of mobility of the molecules or atoms	T is a unitless value; it correlates to the mobility of the optimization process to accept a higher cost design
k_B is the Boltzmann constant	ΔE_{avg} is the running average value of the ΔE; it normalizes the change in the objective (ΔE)

5.5.2 Algorithm Description

5.5.2.1 Selecting Algorithm Parameters

In the above table, notice that instead of (5.1), we use the following to estimate Boltzmann probability:

$$P = \exp\left(\frac{-\Delta E}{\Delta E_{avg} T}\right) = e^{\left(-\Delta E / \Delta E_{avg} T\right)} \tag{5.2}$$

We see that this equation includes ΔE_{avg} instead of *k.* The constant ΔE_{avg} is a running average of all of the "accepted" ΔE (objective changes) to this point in the optimization. It normalizes the change in the objective, ΔE , in (5.2).

Equation (5.2) is also a function of a "temperature," T. How is this chosen? We recommend you set the probability level, P_s , that you would like to have at the beginning of the optimization that a worse design could be accepted. Do the same for the probability level at the end of the optimization, P_f . Then, if we assume $\Delta E = \Delta E_{avg}$, (which is clearly true at the start of the optimization),

$$T_s = \frac{-1}{\ln\left(P_s\right)} \qquad T_f = \frac{-1}{\ln\left(P_f\right)} \tag{5.3}$$

Select the total number of cycles, N, you would like to run. Each cycle corresponds to a temperature. Decrease temperature according to the expression,

$$T_{n+1} = F \cdot T_n \qquad F = \left(\frac{T_f}{T_s}\right)^{1/(N-1)} \tag{5.4}$$

where T_{n+1} is the temperature for the next cycle, T_n is the current temperature, and F is the reduction factor (< 1). Note that the design should be perturbed at each temperature until "steady state" is reached. Since it is not clear when steady state is reached for a design, we recommend perturbing the design at least *n* (*n* = no. of design variables) or more if computationally feasible.

5.5.2.2 Example: Choosing Parameters for Simulated Annealing

We pick the following:

$$P_s = 0.5 \qquad P_f = 10^{-8} \qquad N = 100$$

and using (4.3) and (4.4) we calculate,

$$T_s = 1.4426 \qquad T_f = 0.054278 \qquad F = 0.9674$$

5.5.2.3 Algorithm Steps

1. Choose a starting design.
2. Select P_s, P_f, N, and calculate T_s, T_f and F.
3. Randomly perturb the design to different discrete values close to the current design.
4. If the new design is better, accept it as the current design.

5. If the new design is worse, generate a random number between 0 and 1 using a uniform distribution. Compare this number to the Boltzmann probability. If the random number is lower than the Boltzmann probability, accept the worse design as the current design.
6. Continue perturbing the design randomly at the current temperature until "steady state" is reached.
7. Decrease temperature according to $T_{n+1} = F \cdot T_n$
8. Go to step 3.
9. Continue the process until T_f is reached.

In the early stages, when the temperature is high, the algorithm has the freedom to "wander" around design space. Accepting worse designs allows it to escape local minima. As temperature is decreased the design becomes more and more "frozen" and confined to a particular region of design space.

A diagram of the algorithm is given in Fig. 5.4:

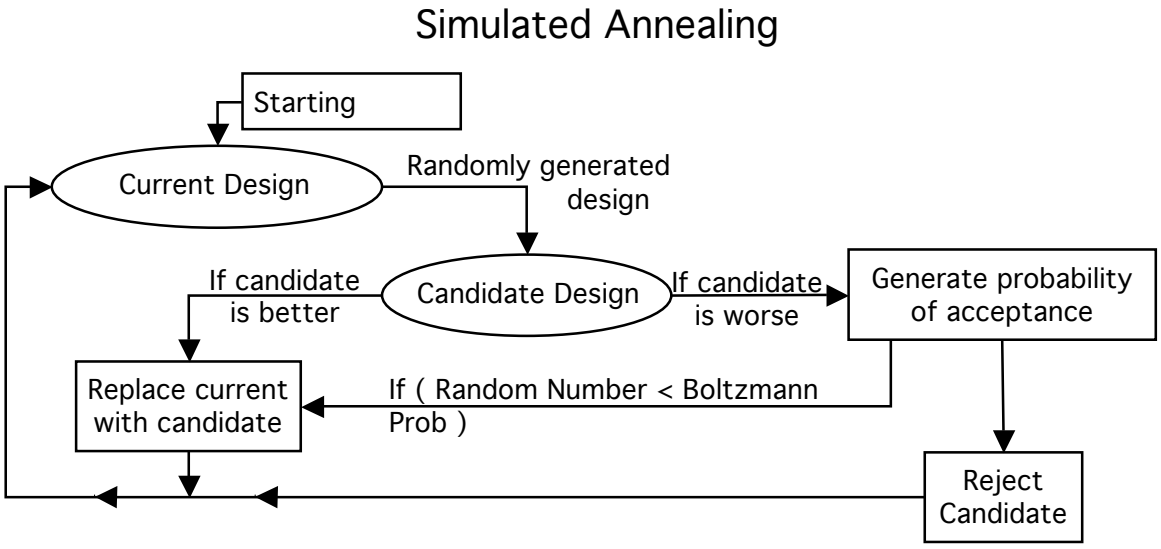

Fig. 5.4. The Simulated Annealing algorithm.

5.5.2.4 Limitations of Simulated Annealing

Simulated annealing is really developed for unconstrained problems. Questions arise when applied to constrained problems--if the perturbed design is infeasible, should it still be accepted? Some implementations automatically reject a design if it is infeasible; others use a penalty function method so the algorithm "naturally" wants to stay away from infeasible designs.

Simulated annealing does not use any gradient information. Thus it is well suited for discrete problems. However, for continuous problems, if gradient information is available, a gradient-based algorithm will be much (>100 times) faster.

5.5.3 Examples of Simulated Annealing

Balling describes the optimization of a 3D, un-symmetric 6 story frame, shown below.

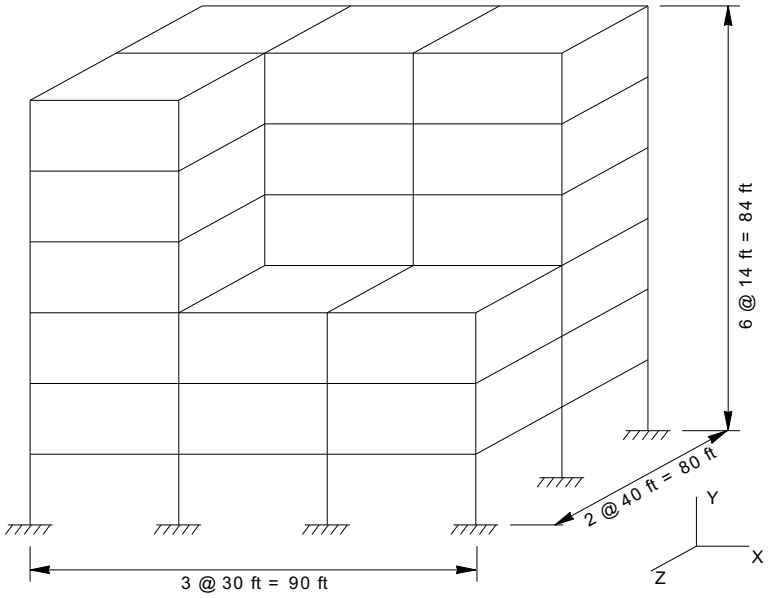

Fig. 5.5. Six story frame.

The 156 members were grouped into 11 member groups--7 column groups and 4 beam groups. Beams and columns must be selected from a set of 47 economy sections for beams and columns respectively. The starting design had a weight of 434,600 lbs. Eleven perturbations were examined at each temperature, and with $N = 100$, an optimization required 1100 analyses. Two iterations of simulated annealing were performed, with the starting design of the second iteration being the optimum from the first. The results were as follows:

Iteration	Optimal Weight	Execution Time
1	416,630 lbs.	1:24:09
2	414,450 lbs.	1:26:24
Total		2:50:33

The change in weight observed as temperature was decreased for the first iteration was very similar to the diagram given Fig. 5.3.

Simulated annealing was compared to the branch and bound strategy. First a continuous optimization was performed. Each design variable was then limited to the nearest 4 discrete variables. To reduce computation, a linear approximation of the structure was made using information at the continuous optimum. Because the neighborhoods were so small, the algorithm was run 4 iterations, where the starting point for the next iteration was the optimum from the current iteration.

Iteration	Optimal Weight	Execution Time
1	420,410 lbs.	0:13:44
2	418,180 lbs.	1:09:44
3	414,450 lbs.	0:12:24
Total		1:35:52

Liu [43] used simulated annealing for the discrete optimization of pipe network systems. Pipes, like wide flange sections, are only manufactured in certain sizes. For this work, each pipe could be selected from 30 possible values.

An example network is shown in Fig. 5.6. This network has 22 pipe sizes to be chosen.

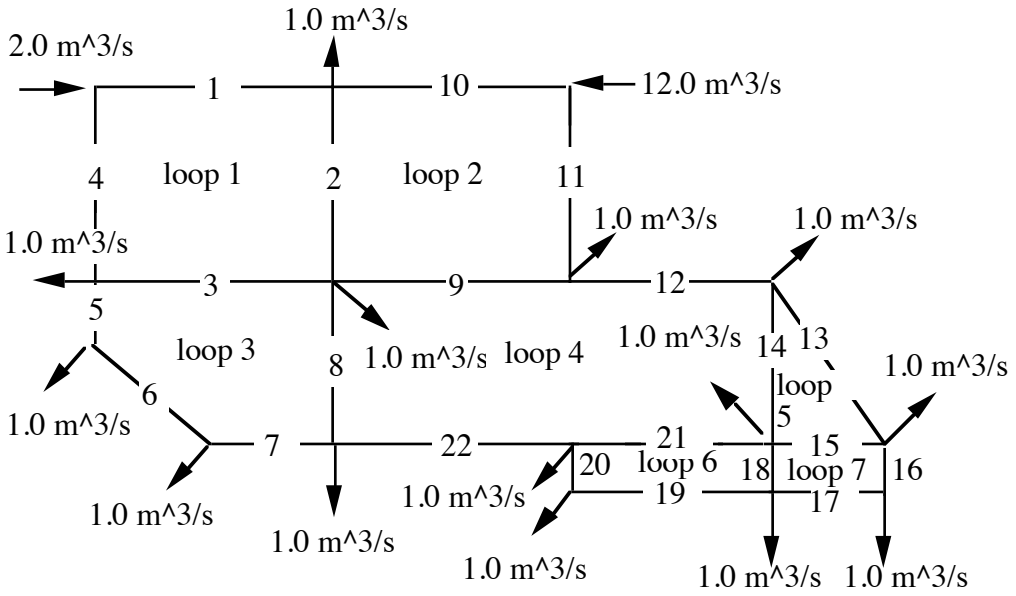

Fig. 5.6 Pipe network optimized with simulated annealing

Arrows give required flows that must enter or leave the network. Simulated annealing was used to find the network with the least cost that could meet these demands for this problem, P_s = 0.9, F=0.9, N=65, and 5 perturbed designs were evaluated at each temperature. For this optimization 7221 analysis calls to the network flow solver were required.

The change in cost during optimization is given below.

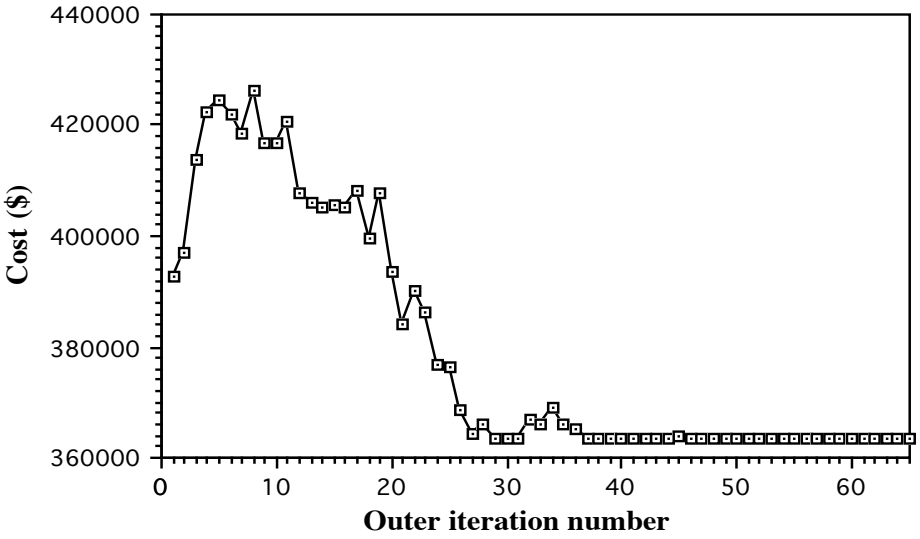

Fig. 5.7 Simulated annealing history for pipe network.

Additional information about Simulated Annealing can be found in Aarts E. and J. Korst [34], Bohachevsky et al. [35], Kirkpatrick et al. [36] and Press et al. [37].

5.6 Classical Genetic Algorithms

5.6.1 Introduction

Genetic Algorithms (GA) are based on the Darwinian theory of natural selection and survival of the fittest. The search algorithm mimics reproduction, crossover, and mutations in nature. The roots of genetic algorithms are traced to work done by Holland [38]. Taking a quote from Davis [39]:

"In nature species are searching for beneficial adaptations to a complicated and changing environment. The "knowledge" that each species gains with a new generation is embodied in the makeup of chromosomes. The operations that alter the chromosomal makeup are applied when parents reproduce; among them are random mutation, inversion of chromosomal material and crossover--exchange of chromosomal material between two parents' chromosomes. Random mutation provides background variation and occasionally introduces beneficial material into a species' chromosomes. Inversion alters the location of genes on a chromosome, allowing genes that are co-adapted to cluster on a chromosome, increasing their probability of moving together during crossover. Crossover exchanges corresponding genetic material from two parent chromosomes, allowing beneficial genes on different parents to be combined in their offspring."

Goldberg [40] has suggested four ways that genetic algorithms are different from traditional derivative-based algorithms:

- GA's work with a coding of the variables, not the variables themselves.
- GA's search from a population of points, not a single point.
- GA's use objective function information, not derivatives or other auxiliary knowledge.
- GA's use probabilistic transition rules, not deterministic rules.

As given in Gen [41], there are five basic components to a genetic algorithm:

1. A genetic representation of solutions to the problem.
2. A way to create an initial population of solutions.
3. An evaluation function rating solutions in terms of the fitness.
4. Genetic operators that alter the genetic composition of children during reproduction.
5. Values for parameters of genetic algorithms.

In the next section all of the components will be specified as we step through the algorithm.

5.6.2 Steps of the Classical Algorithm

1. Determine a coding for the design. The classical algorithm uses binary coding. A design is coded as a "chromosome."
2. Develop the initial population. This can be done by randomly creating a set of designs which are evenly spread through the design space. A population of 20 to 100 designs often works well.

3. Pick a crossover and mutation rate. Typical values are 0.8 and 0.01, respectively. These are problem dependent, however, and are often determined experimentally.
4. Select a way to measure the "fitness" or goodness of a design. Often we will just use the objective value. (In Chapter 6, we will learn other ways of measuring fitness.)
5. Select the mating pool. These will be the designs which will be chosen to become parents and produce the next generation. This selection can be done several ways. Two of the most popular are roulette wheel selection and tournament selection. In roulette wheel selection, we select a parent based on spinning a "roulette wheel." The size of the slot on the wheel is proportional to the fitness. In tournament selection, a subset of the population is randomly selected and then the best design from the subset (the tournament) is taken to be the parent. We will illustrate both of these.
6. Perform "crossover." This requires that we select a crossover site and then "swap" strings at the crossover point, creating two new children.
7. Perform "mutation." The check for mutation is done on a bit by bit basis. The mutation rate is usually kept low (0.005, for example). If a bit mutates, change it from 1 to 0 or vice-versa.
8. The new population has now been created. Decode the population strings to get the normal variable values and evaluate the fitness of each design. We now have a new generation of designs, and so we go back to step 2.
9. Continue for a specific number of generations or until the average change in the fitness value falls below a specified tolerance.

The above steps can be modified by several changes to enhance performance. We will discuss these more in Chapter 6.

There are several parts to the above steps; these will be explained further.

5.6.3 Binary Coded Chromosomes

5.6.3.1 Precision with Binary Strings

The original GA algorithm worked with binary coded strings. If we first consider continuous variables, we will need to convert them to binary, this requires that we establish an acceptable level of precision. This is determined from,

$$\text{Precision} = \frac{(U_i - L_i)}{2^p - 1} \tag{5.5}$$

where U_i = upper bound for ith variable
L_i = lower bound for ith variable
p = length of binary string

The precision determines the smallest change we can make in a variable and have it reflected in the binary string.

5.6.3.2 Example: Determining the Precision of Binary Coding

We decide to have a binary string length of 8, and a variable has an upper bound of 10 and a lower bound of zero. The precision is,

$$\frac{(10-0)}{2^8-1} = \frac{10}{255} = 0.0392$$

This is the smallest change in a variable we will be able to distinguish using a binary coding with a string length of 8.

5.6.3.3 Converting from Real to Binary

To convert a real number value to a binary string, first convert it to a base 10 integer value, using the formula,

$$x_{\text{int10}} = \frac{(x_{real} - L) * J}{(U - L)} \tag{5.6}$$

where x_{real} = real number value
x_{int10} = base 10 integer
J = $2^p - 1$

Then convert the integer to a binary string using $2^0 = 1, \quad 2^1 = 2, \quad 2^2 = 4$, etc.

5.6.3.4 Example: Converting from Real to Binary

We have a variable value of 3.567, with a string of length 8, and an upper bound of 10 and a lower bound of zero. The base 10 integer value is,

$$x_{\text{int10}} = \frac{(3.567 - 0)\ 255}{10 - 0} = 90.95 = 91$$

In binary, this value is 01011011 = $((2^0 = 1) + (2^1 = 2) + (2^3 = 8) + (2^4 = 16) + (2^6 = 64) = 91$

To go from binary back to real, just solve (4.9) for the real value:

$$x_{real} = x_{\text{int10}} * \frac{(U - L)}{J} + L \tag{5.7}$$

5.6.3.5 Creating Chromosomes

A chromosome is created by combining the binary strings of all variables together. If we had two variables, which in binary form were `01011011` and `10001101`, the chromosome would be:

```
0101101110001101
```

5.6.4 Genetic Operators: Crossover and Mutation

5.6.4.1 Crossover

Crossover is the exchange of genetic material between two parent chromosomes to form two children. We randomly pick the crossover point and then swap the binary strings which follow after the crossover point. For example if before crossover we have the following parents with the indicated crossover point,

Parent 1: `00111|010`
Parent 2: `11100|111`

Then after crossover we have:

Child 1: `00111111`
Child 2: `11100010`

5.6.4.2 Mutation

It is sometimes beneficial to introduce random "mutations" into our design. This is usually done bit by bit. If we have a mutation rate of 0.001 (0.1%), we draw random numbers from a uniform distribution. If the number is less than 0.001, we switch the bit from 0 to 1 or vice versa. Mutation helps introduce unexpected changes into the population. However, the mutation rate cannot be too high, or a high fraction of designs are essentially changed randomly. However, some practitioners have used mutation rates as high as 10%.

5.6.5 Example: Converting a Chromosome Back to Variables

Suppose the chromosome for two variables (x_1, x_2), each of string length 10, is given by:

`0010110111011011100`

We partition the chromosome into:

x_1: `0010110111` x_2: `1011011100`

Minimum and maximum values:

$5 \le x_1 \le 10$ $1 \le x_2 \le 25$

Base10 integer values:

$x_{1,int10} = 183$ $x_{2,int10} = 732$

Continuous real values:

$x_{1,real} = 5.894$ $x_{2,real} = 18.17$

5.6.6 Example: Classical Genetic Algorithm for One Generation

In this problem we have the objective function $f = x_2^1 + x_2^2$ which we wish to maximize. The variables range from –2 to +5. We will have a string length of 7. We assume we have available a random number generator that generates uniformly distributed numbers between

0 and 1. We will have a population size of 6 (which is small, but is used here for illustration purposes), a crossover probability of 0.8 and a mutation probability of 0.001. We will use roulette wheel selection to choose parents.

We randomly generate the following six designs as our starting population:

Design	x_1	x_2	Fitness
1	1.521	–0.824	2.9924
2	3.922	–1.006	16.394
3	2.179	–0.033	4.7491
4	–0.292	4.405	19.489
5	–0.523	–1.636	2.95
6	2.956	–1.951	12.544

We then convert the designs to binary strings and form chromosomes:

Design	x_1	Base 10 Int	Binary	x_2	Base 10 Int	Binary	Chromosome
1	1.521	64	1000000	–0.824	21	0010101	10000000010101
2	3.922	107	1101011	–1.006	18	0010010	11010110010010
3	2.179	76	1001100	–0.033	36	0100100	10011000100100
4	–0.292	31	0011111	4.405	116	1110100	00111111110100
5	–0.523	27	0011011	–1.636	7	0000111	00110110000111
6	2.956	90	1011010	–1.951	1	0000001	10110100000001

There is some additional information we need to compute for the roulette wheel selection:

Design	Fitness	Fitness/Sum	Cumulative Probability
1	2.992	0.0506	0.0506
2	16.39	0.2773	0.328
3	4.749	0.0803	0.408
4	19.49	0.3297	0.738
5	2.950	0.0499	0.788
6	12.54	0.2122	1.00
Sum	59.12	1.00	
Average	9.85		

The cumulative probability will be used to set the size of the slots on the roulette wheel. For example, Design 2 has a relatively high fitness of 16.39; this represents 27.7% of the total fitness for all designs. Thus it has a slot which represents 27.7% of the roulette wheel. This slot is the distance from 0.0506 to 0.328 under the cumulative probability column. If a random number falls within this interval, Design 2 is chosen as a parent. In like manner, Design 5, which has a low fitness, only gets 5% of the roulette wheel, represented by the interval from 0.738 to 0.788.

We draw out six random numbers:
0.219, 0.480, 0.902, 0.764, 0.540, 0.297

The first random number is in the interval of Design 2—so Design 2 is chosen as a parent. The second number is within the interval of Design 4—so Design 4 is chosen as a parent. Proceeding in this fashion we find the parents chosen to mate are 2,4,6,5,4,2.

We will mate the parents in the order they were selected. Thus we will mate 2 with 4, 6 with 5, and 4 with 2.

Should we perform crossover for the first set of parents? We draw a random number, 0.422, which is less than 0.8 so we do. We determine the crossover point by selecting a random number, multiplying by 13 (the length of the chromosome minus 1) and taking the interval the number lies within for the crossover point (i.e., 0-1 gives crossover at point 1, 10-11 gives crossover at point 11, etc.) , since there are 1–13 crossover points in a 14 bit string. Crossover occurs at: 0.659 * 13 = 8.56 = 9th place.

Parent 1: 001111111|10100
Parent 2: 110101100|10010

Child 1: 00111111110010
Child 2: 11010110010100

Do we perform mutation on any of the children? We check random numbers bit by bit--none are less than 0.001.

Do we do crossover for the second set of parents? We draw a random number of 0.749, less than 0.8, so we do. Crossover for second mating pair: 0.067*13 = 0.871 = 1st place

Parent 3: 1|0110100000001
Parent 4: 0|0110110000111

Child 3: 10110110000111
Child 4: 00110100000001

Again, no mutation is performed.

Do we do crossover for third set of parents? Random number = 0.352 ≤ 0.8, so we do. Crossover for third mating pair: 0.260*13 = 3.38 = 4th place

Parent 5: 1101|0110010010
Parent 6: 0111|1111110100

Child 5: 11011111110100
Child 6: 00110110010010

As we check mutation, we draw a random number less than 0.001 for the last bit of Child 5. We switch this bit. Thus this child becomes,

Child 5: `11011111110101`

We now have a new generation.

We decode the binary strings to Base 10 integers which are converted to real values, using (4.10). Information on this new generation is given below.

Design	Chromosome	Binary x_1	Base 10 Int	x_1	Binary x_2	Base 10 Int	x_2	Fitness
1	00111111110010	0011111	31	-0.291	1110010	114	4.283	18.43
2	11010110010100	1101011	107	3.898	0010100	20	-0.898	16.00
3	10110110000111	1011011	91	3.016	0000111	7	-1.614	11.7
4	00110100000001	0011010	26	-0.567	0000001	1	-1.945	4.10
5	11011111110101	1101111	111	4.118	1110101	117	4.394	36.26
6	00110110010010	0011011	27	-0.5118	0010010	18	-1.008	1.278
							Sum	87.78
							Average	14.63

We see that the average fitness has increased from 9.85 to 14.63.

This completes the process for one generation. We continue the process for as many generations as we desire.

5.6.7 Example: Genetic Algorithm with Tournament Selection

The previous example used roulette wheel selection. The roulette wheel selection process is dependent upon the scaling we choose for the objective. It must also be modified if we wish to minimize instead of maximize.

Another way to select parents which does not have these drawbacks is *tournament selection*. This involves randomly selecting a subset of the population and then taking the best design of the subset. For example, with a tournament size of two, two designs are randomly chosen and the best of the two is selected as a parent.

Tournament selection can be partly understood by considering the extremes. With a tournament size of one you would have random selection of parents, and with a tournament size equal to the population size, you would always be picking the best design as the parent.

We will have a tournament size of two. We generate two random numbers and then multiply them by the population size:

0.219* 6 = 1.31. Values between 1 and 2 give Design 2
0.812* 6 = 4.87. Values between 4 and 5 give Design 5

The best design of these two is Design 2, so design 2 is chosen to be Parent 1. (We are still working with the starting generation, not the second generation given above.) We then

conduct another tournament to find Parent 2. We continue in a similar fashion until six parents are chosen.

See also Michalewicz [42] for additional information.

5.7 Comparison of Algorithms

This chapter has introduced a few of the most common or classical methods for discrete or continuous/discrete optimization.

Some time ago I came across this comparison of gradient-based algorithms, simulated annealing and genetic algorithms. I regret I cannot give credit to the author. The author assumes we are trying to find the top of a mountain using kangaroo(s).

"Notice that in all hill climbing [gradient-based] methods discussed so far, the kangaroo can hope at best to find the top of a mountain close to where he starts. There's no guarantee that this mountain will be Everest, or even a very high mountain. Various methods are used to try to find the actual global optimum.

In simulated annealing, the kangaroo is drunk and hops around randomly for a long time. However, he gradually sobers up and tends to hop up hill.

In genetic algorithms, there are lots of kangaroos that are parachuted into the Himalayas at random places. These kangaroos do not know that they are supposed to be looking for the top of Mt. Everest. However, every few years, you shoot the kangaroos at low altitudes and hope the ones that are left will be fruitful and multiply."