Chapter 14: Heuristics for Discrete Search: Genetic Algorithms and Simulated Annealing

The branch and bound algorithms that we have studied thus far have one very nice property: they guarantee that the optimum solution will be found. But branch and bound also has one fatal flaw: it is combinatorially explosive, and hence will take excessive time (and possibly computer memory) for problems that are larger than medium scale. Further, discrete problems of large scale are very common in practice, e.g. scheduling (shift workers, exams, airline flights, etc.). But these problems still need to be solved, so we have to give up on finding the optimum solution and instead concentrate on finding a pretty good solution within the limits of time and computer memory available.

This means that we need to employ *heuristic* methods. A heuristic is a method that is not guaranteed to find the optimum, but usually gives a very good solution, though it cannot guarantee to do even that every time. Heuristics are "quick and dirty" methods, generally relatively fast and relatively good. We have actually studied a couple of heuristic methods already in Chapter 12: beam search, and stopping branch and bound with a guarantee of closeness to optimality. Here is a rough guide to when to use various discrete search methods:

Problem Size	**Methods**
small	Enumeration
medium	Branch and bound Dynamic programming A* search
large	Branch and bound variants: • Beam search • Guarantee of closeness to optimality Problem-specific heuristics Controlled random search: • Genetic algorithms • Simulated annealing • Tabu search Pure random search

In the rest of this chapter we will look at two popular heuristic methods that are applicable to a very wide range of practical problems.

Genetic Algorithms

These are fascinating algorithms. The name derives from the way in which they loosely mimic the process of evolution of organisms, where a problem solution stands in for the organism's genetic string. Features include a survival of the fittest mechanism in which potential solutions in a population are pitted against each other, as well as recombination of solutions in a mating process and random variations. The incredible part is that this heuristic can "evolve" better and better solutions without any deep understanding of the problem itself! Genetic algorithms can be applied to any problem that has these two characteristics: (i) a solution can be expressed as a string, and (ii) a value representing the worth of the string can be calculated.

Genetic algorithms have a couple of important advantages. They are simple to program and they work directly with complete solutions: unlike branch and bound, there is no need for estimates or for bounding functions.

As an example, let's look again at a variation of the person-job assignment problem. Let me stress that in practice the best way to solve this problem is actually by the exact and fast assignment problem linear program. However this is an easy-to-understand problem that we have worked with before, so we will see how it can be solved via a genetic algorithm. In this example we are assigning salespeople to regions, and the table below shows the expected number of units sold if a salesperson is assigned to a region.

		Region			
		1	**2**	**3**	**4**
Salesperson	**A**	20	37	15	28
	B	25	24	18	29
	C	18	30	14	24
	D	21	33	16	20
	E	23	31	19	23

Our objective is to maximize the number of units sold. Further, since there are only 4 regions to cover, we must assign just 4 of the 5 salespeople (each salesperson can handle only one region). Which of the 4 salespeople should be chosen, and how should they be assigned to the regions to maximize the total number of units sold?

Let's first check that a genetic algorithm can be applied to this problem. Can a solution be expressed as a string? Yes: a solution such as CDAB can represent the assignment of salesperson C to region 1, salesperson D to region 2, salesperson A to region 3 and salesperson B to region 4. Can a value be assigned to a string to represent its value? Yes: simply add up the expect units sold for the solution; for example the value associated with string CDAB would be 18 + 33 + 15 + 29 = 95.

Now we can use this example to explore a very basic genetic algorithm approach to solving this problem. At all times we will have a *population* consisting of numerous solution strings. Each string is analogous to a genetic string of chromosomes. The solutions will compete with each other in a survival of the fittest contest where their chances of survival are proportional to the relative "goodness" of their solution string value. Parts of surviving strings are then combined in various ways through a process similar to male-female reproduction to create a population of new child strings. Some of these may be randomly changed as happens in real life through e.g. bombardment via cosmic rays. Now we have a new population, and the process repeats. Amazingly, after this cycle repeats a number of times, there are usually much better solutions in the current population than in the original. Note however that the process is not entirely random: good solutions have a better chance of survival, and a better chance or reproduction, and reproduction tends to combine parts of stronger solutions into even better ones. Good characteristics tend to persist in the population and to combine in useful ways.

There are three main *operators* in a basic genetic algorithm: reproduction, crossover, and mutation. We will examine each of these in turn. First, however, it is necessary to establish an initial population of solutions. The simplest (but probably not the best) way to create an initial population is generate it randomly. We will discuss better ways later. The size of the population (i.e. how many solutions there should be) is also an important parameter: it must be large enough that it can support sufficient genetic variation, but not so large that calculations take an inordinate amount of time. In practice, the population size is often determined by experimentation.

The Reproduction Operator

The reproduction is equivalent to the "survival of the fittest" contest. It determines not only which solutions survive, but how many copies of each of the survivors to make. This will be important later during the crossover operation. The probability of survival of a solution is proportional to its solution value; also known as its *fitness* (the function that assigns values to solution strings is also known as the *fitness function*).

As an example, consider a population of 4 solution strings from our small salesperson assignment problem, and the relative fitness of each string:

String	Fitness (solution value)	Fitness as % of total
CDAB	95	95/373 = 25.5%
BADC	102	102/373 = 27.3%
BCDA	99	99/373 = 26.5%
CBAD	77	77/373 = 20.7%
fitness total	*373*	*373/373 = 100.0%*

The first 3 solutions are relatively evenly matched, though the fourth solution is a bit weaker. How will we decide which ones survive? Conceptually, we construct a virtual weighted roulette wheel, as shown in Figure 14.1, where the weight of any solution is proportional to the "fitness as % of total" shown in the table above. "Spinning the wheel" by generating a random number selects a solution string to reproduce a copy of itself into a new intermediate population known as the *mating pool* for reasons that will be clear soon. If we chose a population of size *n*, then the wheel is spun *n* times to create a mating pool of size n. In our small example since the population size is 4, then the wheel is spun 4 times.

In reality we “spin the roulette wheel” by generating a uniformly distributed random number between 0 and 100. The solution is then selected based on the cumulative sum of the fitness relative weights. For the example in the table and in Figure 14.1, we spin the wheel and select as follows:

- If the random number is between 0 and 25.5, then select CDAB,
- If the random number is between 25.6 and 25.5+27.3=52.8, then select BADC,
- If the random number is between 52.9 and 25.5+27.3+26.5=79.3, then select BCDA,
- If the random number is between 79.4 and 100.0, then select CBAD.

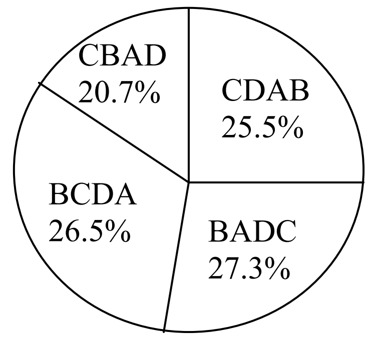

Figure 14.1: A virtual roulette wheel.

Note that it is entirely possible for one of the solutions to be selected more than once, and for some solutions not be chosen at all. In general it is most likely that the stronger (most fit) solutions will be chosen (i.e. survive) most often, and that the weaker (most unfit) solutions will not be chosen (i.e. die). However, due to the random nature of the process, it is also possible for a weak solution to be chosen multiple times and for a strong solution to die, but this is unlikely.

After the reproduction operation, we have an intermediate population known as the mating pool that is ready to mix and mingle, akin to the process of mating and reproducing children that share some of the genetic material of each parent. This is the function of the *crossover* operator.

The Crossover Operator

During crossover, two parent solution strings from the mating pool combine to create two new child solution strings. This happens as follows:

1. Randomly select two parent strings from the mating pool.
2. Randomly select a *crossover point* in the solution string. This is the point between any two positions in the solution string.
3. Swap the ends of the two parent strings, from the crossover point to the end of the string, to create two new child strings.

parent strings		*child strings*
XXXX|XX	→	XXXXOO
OOOO|OO		OOOOXX
↑ *crossover point*		

Figure 14.2: Illustration of crossover.

This process is illustrated in Figure 14.2, where X and O represent values in the two solution strings. In our example we might see a crossover such as:

BC|DA → BCAD
CB|AD   CBDA

There are numerous variations on the basic crossover operator, for example randomly choosing *two* crossover points and swapping the string contents between those two crossover points.

Of course, it is entirely possible that crossover will produce infeasible children, as for example:

CDA|B → CDAC
BAD|C BADB

In this case, both children are infeasible because they both contain repeated salespeople, and each salesperson can handle just one region.

How are we to handle the problem of infeasible child strings? The best way is to use a different variant of crossover that does not allow infeasible children to be created at all: we will describe one such variant (*partially-matched crossover)* later. If infeasible children are relatively infrequent, they can be handled by simply rejecting the infeasible child and applying the crossover operator again. Finally, if there is no better crossover operator and infeasibility is relatively frequent then you can accept the infeasible child, but penalize its fitness. In our example, we could adjust the fitness downwards, e.g. by 10 points for every repeated salesperson in a solution string (or by a squared factor, or many other ways).

The new population is now almost ready. There is one last operator to apply.

The Mutation Operator

The mutation operator is used to randomly alter the values of some of the positions in some of the strings based on a parameter that determines the level of mutation. One common choice is a 1 in 1000 chance of mutation. This can be implemented as follows. For each position in each string, generate a random integer between 1 and 1000. If this number is 1, then the position is chosen for mutation, and is randomly switched to any other possible value. In our example, the second position in the string CBAD might be chosen for mutation and might randomly switched from a value of B to a value of E. This is an improvement: CBAD has a fitness of 77, while CEAD has a fitness of 84.

Of course it is just as possible that the mutation could worsen the fitness function or even generate an infeasible solution. Given this downside, why do we bother with mutation at all? There is a very good reason. For a clue take a look at the set of solutions that comprised the original population in our example (see table on page 3). What do you notice about that set of solutions?

Salesperson E is not present in *any* of the solutions in that initial population! And there is no way that salesperson E will be introduced by either the reproduction or crossover operators. The *only* way that salesperson E might appear in a solution is via mutation. Now we see the motivation behind mutation: to sample the solution space widely. So where reproduction and crossover try to concentrate the solutions that we already have into better solutions, mutation works instead to sample the solution space and to broaden the search.

Mutation is a vital part of the solution process, and the mutation rate can have a big impact on the quality of the final solution. It is even possible (though vastly more inefficient) to solve problems using *only* the mutation operator.