Genetic Algorithms

Let's remind ourselves of the simple table-driven agent that we designed for walking anticlockwise around the walls of grid-based rooms. The agent has eight touch sensors mounted around its body, each of which returns 1 if there's an object in the corresponding cell and 0 otherwise. In fact, we realised we need only three of the sensors to be switched on. But in this lecture we will assume all eight are operative.

We used three actions to implement our agent: `Move`, which moves the agent one pace forward in the direction that it is facing; `Turn(RIGHT, 2)`, which turns the agent 90^o (2 lots of 45^o) to the right; and `Turn(LEFT, 2)`, which turns the agent 90^o to the left. But in this lecture, we will assume the agent has a total of four actions:

- `Move`;
- `Turn(RIGHT, 2)`;
- `Turn(RIGHT, 4)`;
- `Turn(LEFT, 2)`.

We worked hard to design our agent in the previous lecture: it took ingenuity to realise that only three of the four sensors and three of the eight actions were needed; and it took ingenuity to fill in the table.

Since all the computation was done by us, the designers, in advance, we agreed we could not really regard the agent as being autonomous.

In this lecture, we are going to consider an alternative way of obtaining the agent's table. We are going to look at *genetic algorithms* (GAs). We will *evolve* agents with good tables. We stress from the outset that evolving tables for table-driven agents is only one use of GAs. There are lots of other applications, some of which are mentioned at the end of these notes.

1 Evolution by Natural Selection

The theory of evolution by natural selection can be formulated in many ways, but the following encapsulates its essential elements:

- Successive generations can differ from previous ones. Children inherit characteristics of their parents. But combining and mutating these characteristics introduces variation from generation to generation.
- Less fit individuals are selectively eliminated ('survival of the fittest').

Thus it is possible for individuals in successive generations to perform better than those in previous ones.

Programs that emulate this process are referred to as *Genetic Algorithms (GAs)*. You might also encounter the phrase *Evolutionary Algorithms*, but this phrase is sometimes used in a broader way.

2 GAs

GAs iteratively update a *population* of *individuals*. On each iteration, the individuals are evaluated using a *fitness function*. A new generation of the population is obtained by probabilistically selecting fitter individuals from the current generation. Some of these individuals are admitted to the next generation unchanged. Others are subjected to *genetic operators* such as crossover and mutation to create new offspring.

Here's a sketch of a typical GA in pseudocode:

Algorithm: GA(n, χ, μ)

// Initialise generation 0:
$k := 0$;
$P_k :=$ a population of n randomly-generated individuals;
// Evaluate P_k:
Compute *fitness*(i) for each $i \in P_k$;
do
{ **// Create generation $k+1$:**
  // 1. Copy:
  Select $(1-\chi) \times n$ members of P_k and insert into P_{k+1};
  // 2. Crossover:
  Select $\chi \times n$ members of P_k; pair them up; produce offspring; insert the offspring into P_{k+1};
  // 3. Mutate:
  Select $\mu \times n$ members of P_{k+1}; invert a randomly-selected bit in each;
  // Evaluate P_{k+1}:
  Compute *fitness*(i) for each $i \in P_k$;
  // Increment:
  $k := k+1$;
}
while fitness of fittest individual in P_k is not high enough;
return the fittest individual from P_k;

n is the number of individuals in the population; χ is the fraction of the population to be replaced by crossover in each iteration; and μ is the mutation rate.

We'll look at the parts of this in more detail and exemplify them by considering how to evolve a table-driven agent for wall-following.

3 Representation of individuals

Individuals in the population are often represented by *bit strings*. Then, crossover and mutation are easy to implement and efficient. But it does imply that we need to implement ways of *encoding* an individual as a bit string and *decoding* again.

Let's consider encoding and decoding our table-driven agents from earlier.

We have eight sensors, each returning 0 or 1. This gives us $2^8 = 256$ different percepts and so the table for our table-driven agent will have 256 entries. (In the earlier lecture, when we restricted attention to just three sensors, the table was only 8 entries.) The percepts are already bit strings, which is handy.

We have four actions. With two bits, we can assign a unique bit pattern to each action. For example:

Action	Bit string
`Move`	00
`Turn(RIGHT, 2)`	01
`Turn(RIGHT, 4)`	10
`Turn(LEFT, 2)`	11

So imagine these are the first few entries in an agent's table:

Percept	Action
00000000	`Move`
00000001	`Turn(LEFT, 2)`
00000010	`Turn(RIGHT, 4)`
⋮	⋮

We can concatenate all the entries to form one long bit string:

$$\overbrace{00000000}^{\text{percept}}\ \overbrace{00}^{\texttt{Move}}\ \overbrace{00000001}^{\text{percept}}\ \overbrace{11}^{\texttt{Turn(LEFT, 2)}}\ \overbrace{00000010}^{\text{percept}}\ \overbrace{10}^{\texttt{Turn(RIGHT, 4)}}\ \ldots$$

Class Exercise How long will the bit string be?

In fact, if you think about it, we don't need to include the percepts in the bit string:

$$\overbrace{00}^{\texttt{Move}}\ \overbrace{11}^{\texttt{Turn(LEFT, 2)}}\ \overbrace{10}^{\texttt{Turn(RIGHT, 4)}}\ \ldots$$

We know that the first two bits are the action the agent will execute when the percept is 00000000; the next two are the action for percept 00000001; the next two are for 00000010; and so on.

Class Exercise How long will the bit string be? How many different bit strings are there?

With this representation, it is easy for us to encode one of our table-driven agents as a bit string and just as easy to decode a bit string to recreate a table-driven agent.

4 Fitness

The GA must be equipped with a *fitness function*, enabling it to score and rank the individuals. This will be task-specific: the fitness function used to evolve a wall-following agent will be different from one used to evolve a timetabling agent, for example.

The fitness function evaluates the individual (usually in its decoded state, not when it is a bit string) on some task. Typically, *average* performance on a number of randomly-generated tasks is used.

In the case of our wall-following agent, the obvious way to measure fitness is to place it into the room at some random position and then count the proportion of grid cells that lie next to the walls that it manages to visit. (This is relatively

inexact, since it doesn't penalise the agent for visiting cells that do not lie next to the walls, neither does it insist on an anticlockwise circuit, but it is adequate for our simple example.) We can stop the agent after a fixed time, and repeat the task several times, restarting the agent in a different random position each time. The fitness score is then the average performance.

5 Copy

If χ is the fraction of the next generation that will be created by crossover, then $(1-\chi)$ is the fraction that will be copied intact from this generation to the next. In total then, $(1-\chi) \times n$ individuals will be copied over. The question is: how will the GA *select* which individuals to copy? Obviously, this is influenced by their fitness. You might think we just rank them by fitness and copy the top $(1-\chi) \times n$ individuals, but you'd be wrong. Selection works probabilistically. Various options have been proposed.

5.1 Roulette wheel selection

In roulette wheel selection, the probability that individual i is selected, $P(\mathit{choice} = i)$, is computed as follows:

$$P(\mathit{choice} = i) =_{\text{def}} \frac{\mathit{fitness}(i)}{\Sigma_{j=1}^{n} \mathit{fitness}(j)}$$

Think of a roulette wheel in which the segments are of possibly different sizes, based on each individual's relative fitness.

In case you can't see how this is implemented, here's some pseudocode:

```
Algorithm: ROULETTEWHEELSELECTION()
 r := random number, where 0 ≤ r < 1;
 sum := 0;
 for each individual i
 {    sum := sum + P(choice = i);
      if r < sum
      {    return i;
      }
 }
```

5.2 Rank selection

In rank selection, the individuals are sorted by fitness. The probability that individual i is selected is then inversely proportional to its position in this sorted list, i.e. the individual at the head of the list is more likely to be selected than the next individual, and so on through the sorted list.

5.3 Tournament selection

Tournament selection is best explained with a concrete example. Suppose you want to pick 20 individuals from 100. Randomly choose (with uniform probability) a small number of individuals (typically fewer than 10) from the 100 (with replacement). Keep the fittest one. Do this again and again until you have got your 20.

Tournament selection, while slower and more complicated, can create more diverse populations (see later).

6 Crossover

In crossover, portions of two parents from the current generation are combined to create two offspring: a random subpart of the father's bit string is swapped with a random subpart of the mother's bit string.

We need $\chi \times n$ individuals, who will form $\frac{\chi \times n}{2}$ pairs of parents. How will these individuals be selected? Answer: in the same way as we selected individuals for copying, i.e. by roulette wheel selection, rank selection or tournament selection.

How will offspring be created from parents? Various options have been proposed. They are summarised in this diagram.[1]

	Parents				*Offspring*
Single-point crossover:	$\underline{11101}001000$				11101010101
		$\searrow$		$\nearrow$	
			$\longrightarrow$		
		$\nearrow$		$\searrow$	
	$00001\underline{010101}$				00001001000
Two-point crossover:	$11\underline{10100}1000$				00101000101
		$\searrow$		$\nearrow$	
			$\longrightarrow$		
		$\nearrow$		$\searrow$	
	$\underline{00}00101\underline{0101}$				11001011000
Uniform crossover:	$\underline{1}11\underline{01}0\underline{0}10\underline{00}$				10001000100
		$\searrow$		$\nearrow$	
			$\longrightarrow$		
		$\nearrow$		$\searrow$	
	$0\underline{00}01\underline{0}1\underline{01}01$				01101011001

In single-point crossover, a random position in the bit string is selected at which swapping occurs; in two-point crossover, two positions are selected and segments are swapped; in uniform crossover, individual bits are chosen at random for swapping.

The offspring may or may not be fitter than their parents. But the hope is that roulette wheel selection or rank selection or tournament selection will have supplied reasonably fit parents, and the offspring might have some fitness advantage if they incorporate parts of these fit parents.

Crossover can be implemented extremely efficiently. We will illustrate this for single-point crossover, using the same example as given above.

1. Choose crossover point: Suppose your bit strings are of length n. Randomly generate a number between 1 and $n-1$ inclusive. Call it p.

   In the example above, the bit strings were length 11. Suppose we generate $p = 5$.

2. Create masks: mask$_1$ will be the binary representation of $2^n - 2^p$; mask$_2$ will be the binary representation of $2^p - 1$.

   In the example, mask$_1$ will be $2^{11} - 2^5 = 2016$, which in binary is 11111100000; mask$_2$ will be $2^5 - 1 = 31$, which in binary is 11111, and if we pad it out to get 11 bits it becomes 00000011111.

[1] The diagram is based on one in T.M. Mitchell: *Machine Learning*, McGraw-Hill, 1997.

3. Create offspring$_1$: offspring$_1$:= (parent$_1$ AND mask$_1$) OR (parent$_2$ AND mask$_2$).

   For the example:

parent$_1$	11101001000		parent$_2$	00001010101	
mask$_1$	11111100000		mask$_2$	00000011111	
AND	11101000000		AND	00000010101	
		OR			= 11101010101

4. Create offspring$_2$: offspring$_2$:= (parent$_1$ AND mask$_2$) OR (parent$_2$ AND mask$_1$).

   For the example:

parent$_1$	11101001000		parent$_2$	00001010101	
mask$_2$	00000011111		mask$_1$	11111100000	
AND	00000001000		AND	00001000000	
		OR			= 00001001000

7 Mutate

At this point the new generation of the population is complete: $(1 - \chi) \times n$ individuals have been copied across, and $\chi \times n$ individuals have been produced by a crossover operator. Now a certain proportion μ of the *new* generation are chosen at random. This selection is done with uniform probability: it is not based on fitness. In each of the chosen individuals a bit is chosen at random and this bit is inverted (flipped to the other value):

Mutation: $111010\underline{0}1000 \longrightarrow 111010\underline{1}1000$

Mutation too has an efficient implementation using bit operations.

1. Choose mutation point: Suppose your bit strings are of length n. Randomly generate a number between 0 and $n-1$ inclusive. Call it p.

   In the example above, suppose we generate $p = 4$.

2. Create a mask: The mask will be the binary representation of 2^p.

   In the example, the mask will be $2^4 = 16$, which in binary (padded to 11 bits is) 00000010000.

3. Invert: the new individual := individual XOR mask.

   For the example:

individual	11101001000
mask	00000010000
XOR	11101011000

Mutation is a more random process than crossover and is used sparingly (i.e. μ will have a low value). But it is worth including because it might help to produce an essential feature that is missing in the current generation.

8 Discussion

Crowding. GAs can be very successful at evolving fit individuals. However, a form of stagnation can occur when there is *crowding*. Crowding occurs when some individual that is fitter than others gets to reproduce a lot, so that copies of this individual and minor variants of this individual take over a large part of the population. This reduces the diversity of the population, making it hard for progress to be made.

There are numerous ways of reducing crowding:

- Mutation is, of course, one way of increasing diversity.
- Rank selection and tournament selection tend to be better than roulette wheel selection at maintaining diversity.
- More sophisticated is fitness sharing, whereby an individual's fitness is reduced if the population already contains similar individuals.
- More sophisticated also is to try to encourage the formation of niches within the population (which you can think of as clusters or subspecies), e.g. by restricting crossover so that it is only applied to similar individuals.

Parameters. You can see that to run a GA, you will need to decide on the values of several parameters. The obvious ones are: n, χ and μ. But you also need: a parameter to terminate the algorithm (e.g. a desired fitness level or a maximum number of generations or a maximum time); if fitness if averaged over several tasks, you will have to decide how many tasks; and if tournament selection is used, you will have to decide the size of each tournament. Finding values for these parameters may require experimentation. In general, large numbers are needed to get enough diversity in the populations; but large numbers bring very high computational costs.

Illegal bit strings. Suppose our agent only had 3 actions. We'd still need to use 2 bits so that each action would have a unique bit string associated with it. But one pattern of bits would go unused. There is a risk that this pattern will be created by the crossover or mutation operators. How do we handle this? One option is to reject the individual immediately and run crossover or mutation again until a legal individual is produced. Another option is to allow the individual to enter the new population but to assign it zero fitness.

Applications. GAs have been used throughout AI to evolve all sorts of individuals: digital circuits, factory schedules, university timetables, neural network architectures/topologies, distance measures for kNN, etc., etc. In the case of timetabling, for example, it is easy to imagine how different timetables can be encoded as bit strings and how fitness can be measured by counting violated constraints and preferences.