Genetic Algorithms

1 Background

Where do the rules of a production system come from? There are several options. The option we're going to look at here is: an agent might have obtained them through a process of *evolution*.

There are two observations:

- 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)* or *Evolutionary Algorithms*. In this lecture, we use GAs to evolve programs (in our case, production systems). This is referred to as *Genetic Programming (GP)* or *Evolutionary Programming*.

2 Genetic Algorithms

Generation 0 is a population of randomly generated individuals.

Repeat a large number of times:

Evaluate each individual by using it on the task for which it is being evolved and scoring its performance on this task using a *fitness function*. (Typically, *average* performance on a number of randomly generated tasks is used.)

Generation $(i + 1)$ is created from generation i.

- A number of individuals in generation $(i + 1)$ will be generated by *copying*. Specifically, a proportion of generation i are picked using *tournament selection* (see below) and are copied directly into generation $(i + 1)$.
- A number of individuals in generation $(i+1)$ will be generated by *crossover operations*. Specifically, to generate each of these individuals, a mother and father are chosen (using tournament selection) from generation i. A random subpart of the mother is switched with a random subpart of the father, hence producing two new children.
- A number of individuals in generation $(i + 1)$ will be generated by *mutation operations*. Specifically, a parent is chosen (using tournament selection) from generation i. A random subpart is replaced by some new random subpart.

Pick the fittest individual from the final generation.

In crossover, the children may or may not be fitter than their parents. But the hope is that tournament selection will supply reasonably fit parents, and the children might have some fitness advantage if they incorporate parts of these fit parents.

Mutation is a more random process and is used sparingly. But it is worth including because it might help to produce an essential feature that was missing in the original population.

Tournament selection is used in GAs to pick individuals from a population. Suppose you want to pick 20 individuals from 100. Pick a small number of individuals (typically fewer than 10) randomly (with replacement) from the 100. Keep the fittest one. Do this again and again until you have got your 20. (There are alternatives to tournament selection which we won't cover here. Tournament selection may have the advantage that it reduces the likelihood of *crowding*. Crowding occurs when some individual that is fitter than others gets to reproduce a lot, so that copies of this individual and variants of this individual take over the population. Tournament selection, on the other hand, tends to maintain diversity.)

You can see that to run a GA, you will need to decide on the values of a large number of parameters.

3 Genetic Programming

Genetic Programming is an application of Genetic Algorithms to the task of evolving software. In Genetic Programming, the individuals will be programs; and, in our case, they will be production systems.

How will we represent programs in a way that supports crossover and mutation? We need compact representations that can be operated on efficiently. But we also prefer it if each randomly generated individual in generation 0 and each individual we obtain in generation $(i + 1)$ from crossover or mutation is a syntactically and semantically correct program (i.e. it can be executed without crashing), otherwise we could have problems when evaluating its fitness. Given that GP is a process that uses random change, this can be hard to arrange.

Two approaches are common. Programs can have bit-string representations or they can have tree representations. Bit-string representations are very common for GAs in general; tree representations are probably more common for GP. We'll look at both representations and use them for production systems.

3.1 A bit-string representation for production systems

Suppose we have m propositions, $p_0, \dots, p_m$ and l actions $a_0, \dots, a_l$. Each rule will be represented by a bit-string comprising m bits to represent the rule's condition and $\lceil \log_2 l \rceil$ bits to represent the rule's action. For example, if we have 8 propositions ($m = 8$), we need 8 bits for the rule's condition. And if we have 5 actions then we need 3 bits for the rule's action ($l = 5$ so we need $\lceil \log_2 5 \rceil = 3$ bits). (2 bits could only encode 4 different actions; 3 bits can encode 8 different actions.) Each of the agent's possible actions can then be assigned a unique binary identifier, e.g.:

TURN_LEFT	000
TURN_RIGHT	001
MOVE	010
NULL_ACTION	011
⋮	⋮

Now consider the following rule:

if p_0 **then** TURN_LEFT

The condition of this rule can be represented as the following bit-string 1#######000. In these bit-strings, we use 0 (**false**) and 1 (**true**), as you'd expect, but we also allow # to represent Don't Care (so they're not really bit-strings). So the string 1####### says that p_0 must be true and we don't care about the other propositions. Given that TURN_LEFT is encoded by 000 (above), the rule as a whole is represented as

$$1\#\#\#\#\#\#\#000$$

The rule

if p_1 **then** TURN_RIGHT

is similarly represented by

$$\#1\#\#\#\#\#\#001$$

The rule

$$\textbf{if} \quad \neg p_0 \wedge \neg p_1 \quad \textbf{then} \quad \text{MOVE}$$

is represented by

$$00\#\#\#\#\#\#010$$

The production system as a whole would be represented by the concatenation of these strings, and probably some extra bits to 'pad' the string out so that it meets some length requirement, e.g.:

1######000#1######00100######010################################

Question *What would be the representation for the following rule?*

$$\textbf{if} \quad p_3 \wedge p_4 \wedge \neg p_0 \wedge p_7 \quad \textbf{then} \quad \text{NULL\_ACTION}$$

(A mathematical observation is that this representation allows only rule conditions that are in *conjunctive normal form*, i.e. they comprise propositions or the negation of propositions conjoined (ANDed) together. It does not allow disjunction in rule conditions. No expressiveness is lost, however, because the rules themselves are implicitly disjoined and so the effect of a disjunction can be had through multiple rules. For example, the rule:

$$\textbf{if} \quad p_0 \vee p_1 \quad \textbf{then} \quad \text{MOVE}$$

can be equivalently represented by two rules:

$$\textbf{if} \quad p_0 \quad \textbf{then} \quad \text{MOVE}$$

$$\textbf{if} \quad p_1 \quad \textbf{then} \quad \text{MOVE}$$

The price is a less compact representation.)

The great advantage of bit-string representations is they give easy implementation of genetic operations. In crossover, you take two bit strings; you choose a random position in them; and you swap their tails from this point. For example, suppose we choose position 8, then applying crossover to these two bit-strings:

$$101\#\#10111\#0$$

$$001\#\#11001\#\#$$

produces

$$101\#\#10101\#\#$$

$$001\#\#11011\#0$$

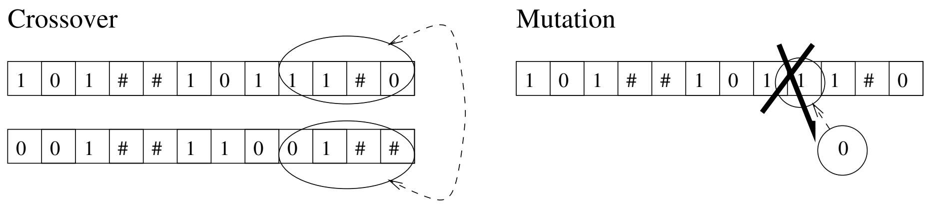
In mutation, you simply choose a random position and flip the bit to another value. For example, if we choose position 8:

$$101\#\#10111\#0$$

could become

$$101\#\#10101\#0$$

These are illustrated graphically below:



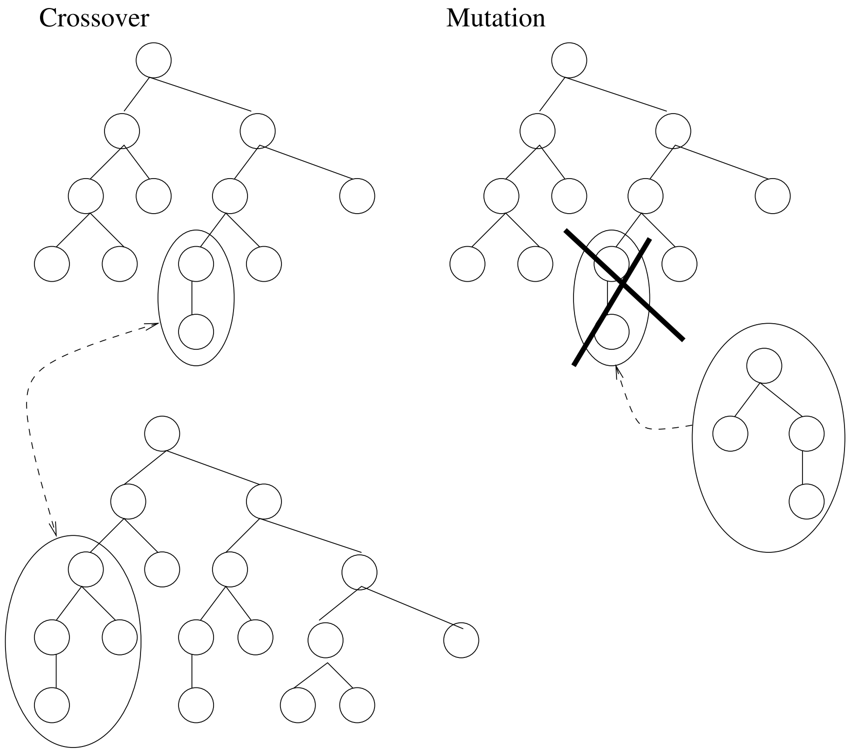
(In fact, what we are depicting here is *single-point crossover*. There are alternatives such as *two-point crossover* and *uniform crossover*. But we will not cover these here.)

Of course, when we're evaluating the fitness of new production systems, we must decide what to do when the parts of the bit-string that are supposed to encode actions do not refer to one of the l actions. (You could use a default or randomly-chosen action, you could reject this individual immediately on syntactic grounds, or, when evaluating the individual, you can assign a zero fitness.)

3.2 A tree representation for production systems

Programs and production systems can be represented as trees. In general, the internal nodes would be functions, operators, predicates, connectives and commands. The leaf nodes would be constants or variables.

We saw, in the previous lecture, that our production systems are already stored as trees. In crossover, a randomly-chosen subtree of the mother and a randomly subtree of the father are switched. In mutation, a randomly-chosen subtree of the parent is replaced by a randomly-generated new subtree:



Again, you must decide how to avoid the evolution of syntactically or semantically incorrect programs; or, if you do allow the evolution of such programs, you must decide how you will evaluate them.

4 Closing Remarks

We've been looking at the evolution of production systems. But GAs in general, and GP in particular, are used throughout AI to evolve all sorts of individuals: programs in various languages, digital circuits, similarity measures, etc.