6
Creating the Animation

Now that the animation can be represented, stored, and played back, all that is left to do is understand how it is created. This is where we will use genetic algorithms, and this chapter will describe how the various stages of G.A. evolution were implemented.

6.1
Genotype

The solution we are trying to optimize is the animation program. In our G.A., then, the chromosome needs to encode data that corresponds to the two animation representations discussed in Chapter 4.

6.1.1
Sequence of Commands

An op-code can be represented by something as simple as a single integer number. Let us imagine a character whose body is composed of 2 springs and 3 claws. As displayed in Figure 6.1, there are twelve valid op-codes for that character. Generalizing for a character with N_S springs and N_C claws, there will be $3N_S + 2N_C$ valid op-codes. (Naturally, this quantity will be different if the character includes other active components, such as angular springs.) Any integer number greater than the number of valid op-codes can be interpreted as a "nothing" op-code — that is, a command that simply maintains the current physical properties of springs, angular springs, motors, etc.

The "DNA" of the animation, then, is a variable-length list of integer numbers.

1. Set spring #1 state: CONTRACTED
2. Set spring #1 state: RELAXED
3. Set spring #1 state: STRETCHED
4. Set spring #2 state: CONTRACTED
5. Set spring #2 state: RELAXED
6. Set spring #2 state: STRETCHED
7. Set claw #1 state: GRIPPING
8. Set claw #1 state: RELEASED
9. Set claw #2 state: GRIPPING
10. Set claw #2 state: RELEASED
11. Set claw #3 state: GRIPPING
12. Set claw #3 state: RELEASED

Figure 6.1: Valid op-codes for a character with two springs and three claws.

6.1.2 Expression Tree

As the name suggests, this representation is best suited to a tree-like data structure. Our implementation was influenced by [Koza 1992] and the guidelines established therein. Specifically:

- Programs are represented by *expression trees*;
- Non-leaf nodes in the tree are unary, binary, or ternary *operators* (see Figure 6.2);
- Leaf nodes in the tree are either *constants* or *variables* (see Figure 6.3);
- A "strict" structure is enforced, that is, the *arity* (number of children) and *type* (Boolean or number) of all nodes is maintained consistent. The tree will never have, for instance, a Boolean constant inserted as a child of an arithmetic operator, or a subtraction with three arguments.

The "DNA" of the animation, then, is a fixed-size list of expression trees — one for each active component (spring, motor, etc.) in the character's body.

Unary operators: These nodes have only one child.

- **Logical "not" (receives Boolean, returns Boolean):** Inverts the value returned by the evaluated subtree.
- **Math function (receives number, returns number):** These nodes are associated with a specific mathematical function (cosine, sine, square root, natural logarithm, or exponential). When evaluated, they take the value returned by the subtree, apply that function to it, and then return the modified value.
- **Negation (receives number, returns number):** Inverts the sign (positive or negative) of the value returned by the evaluated subtree.

Binary operators: These nodes must have two children.

- **Comparator (receives two numbers, returns Boolean):** These nodes are associated with a specific arithmetic comparison operator ("greater", "greater or equal", "less", or "less or equal"). When evaluated, they take the values returned by their two subtrees, compare them with the operator, and then return true or false.
- **Logical operator (receives two Booleans, returns Boolean):** These nodes are associated with a specific Boolean operator ("and", "or", or "exclusive or"). When evaluated, they take the values returned by their two subtrees, combine them with the operator, and then return the resulting value.
- **Math operator (receives two numbers, returns number):** As above, but with mathematical functions (sum, subtraction, multiplication, division, exponentiation).

Ternary operators: These nodes represent "if-then-else" branches, and take three children.

- **Boolean "if" (receives three Booleans):** When evaluated, this node takes the value returned by the first subtree. If it is true, it evaluates the second subtree, otherwise, it evaluates the third subtree. This node, then, always returns a Boolean value.
- **Arithmetic "if" (receives a Boolean and two numbers):** Similar to the above, but always returns a number value.

Figure 6.2: Non-leaf nodes, or operators, used in the expression trees.

Constants: Nodes that always return a fixed value.

- **Boolean constants:** The values true and false.
- **Number constants:** Any real number.

Variables: Nodes whose evaluation depends on the character's current state.

- **Bone-floor contact (Boolean):** These nodes are associated with a specific bone on the character. When evaluated, they return true if the bone is touching the floor.
- **Joint angle (number):** These nodes are associated with a specific joint on the character. When evaluated, they return the angle, in degrees, between the two bones connected to the joint.
- **Bone elevation angle (number):** These nodes are associated with a specific bone on the character. When evaluated, they return the angle, in degrees, between the ground plane and the bone's current orientation.
- **Bone-bone azimuth angle (number):** These nodes are associated with a pair of bones on the character. When evaluated, they return the angle, in degrees, between the two bones' current orientations along the ground plane. (Note that this variable does not exist in the 2D case.)

Figure 6.3: Leaf nodes, or terminals, used in the expression trees.

6.2 Evaluation

The evaluation stage is where a genotype is analyzed and scored based on how good a solution it is. This usually involves decoding the information in the genotype to get a *phenotype*, and then measuring some property of it.

We chose to measure the distance traversed by the character during a ten-second interval while the animation plays. We hoped this would direct the evolution towards "walking" or "running" animations.

6.3 Crossover and Mutation

The crossover operator is responsible for exchanging data between two solutions (parents) to create two new solutions (children). The mutation operator is responsible for slightly modifying the solution in a random manner to avoid stagnation in the evolution process. These operators are heavily

dependant on the genotype representation, so we had to devise two very different strategies for our two representations.

6.3.1 Sequence of Commands

The crossover operator of choice is the *one-point crossover* (see Figure 6.4). This method is useful because it recombines the two animations without completely destroying the parents' sequencing of op-codes.

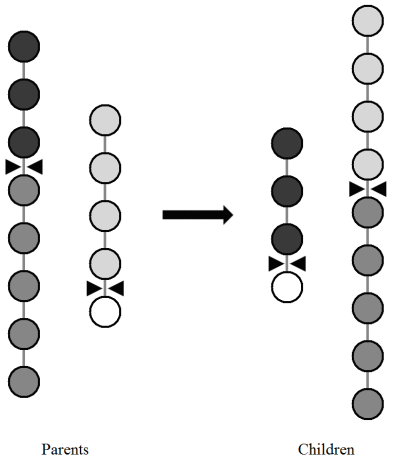

Figure 6.4: One-point crossover. Each of the parent chromosomes is "cut" at a random point to create two sub-sequences; by swapping the sub-sequences, two new chromosomes are created.

When applying the mutation operator, one of four methods is chosen at random:

- **Swap:** Two op-codes in the sequence are selected and their positions are swapped.
- **Destroy:** An op-code in the sequence is removed.
- **Create:** A position in the sequence is chosen to insert a new random op-code, which may be valid or invalid ("nothing" op-code).
- **Flip:** An op-code in the sequence is selected and replaced with a new random op-code. As above, the new op-code may or may not be valid.

6.3.2 Expression Trees

In this case, one chromosome actually consists of several trees (one tree for each active body component). We reckoned that each tree should follow its own evolutionary process, and it did not make sense to cross trees belonging to different components. Thus, when breeding two chromosomes, we apply the crossover operator to pairs of trees belonging to the same component (see Algorithm 4).

Algorithm 4 The function **CrossControllers** breeds and returns two child animations, nicknamed *bro* and *sis*. **CrossTrees**, whose implementation is not shown here, is a function that receives two trees (parents) and returns two new trees (children).

```
function CrossControllers(mom, dad)
  bro ← empty tree list
  sis ← empty tree list
  for each component i in the character do
    bro[i], sis[i] ← CrossTrees(mom[i], dad[i])
  end for
  return bro, sis
end function
```

When performing the crossover of expression trees, we chose the *one-point crossover* operator again (see Figure 6.5). However, care needs to be taken when selecting the crossover point: Because we enforce a type-safe structure in the tree, the crossover must avoid grafting a branch in an incompatible point (e.g., exchanging a Boolean-type node with a number-type node). To fulfill this requirement, we first select a random edge from the *mom* tree, then perform a search in the *dad* tree and collect all edges that are compatible with *mom*'s selected edge. From that subset, we select a random edge that will be the crossover point in *dad*.

Mutation is is performed on a node-by-node basis, that is, if the algorithm is run with a mutation probability of p_{mut} then every node in the tree has p_{mut}

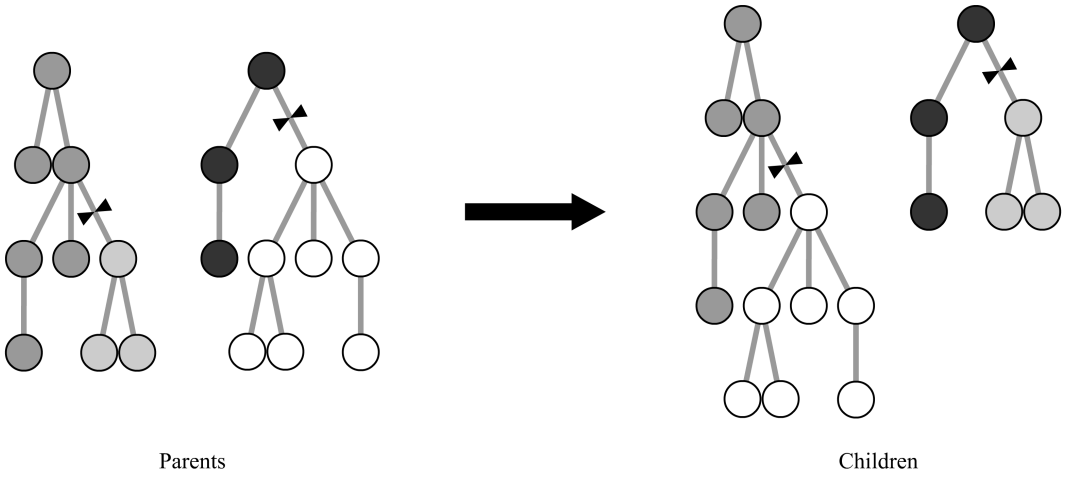

Figure 6.5: One-point crossover for tree structures. Each of the parent trees is "cut" at a random point, and their branches are swapped to create two new chromosomes.

chance of mutation according to the following behaviors:

- **Variable terminals** mutate by pointing to a different component (e.g., a *bone-floor contact* node associated with the contact status of bone #2 will change to be associated with the contact status of bone #4).
- **Constant terminals** mutate by changing their value to some other random value.
- **If-then-else nodes** mutate by swapping the edges *then* and *else* (and, by extension, their subtrees). Since these two edges are always of the same type, this operation is type-safe.
- **Unary operators** mutate by changing their function (e.g., a *sine* node may change to perform a *negation* instead).
- **Binary operators** mutate by changing their function (e.g., an *and* node may change to perform an *exclusive or* instead) and/or swapping their two edges (e.g., $a - b$ may become $b - a$).