sensory and state information. The relative importance of these two resources would not have been understood without the string evolver baseline.

This chapter requires familiarity with genetic programming as introduced in Chapter 8 and would benefit from familiarity with Chapter 9. This chapter uses a slightly more complex GP language than PORS, but with simpler data types: the integers or the integers (mod 3). The use of integer operations makes implementation of the parse tree evaluation routines much easier than in Chapter 9. The dependencies of the experiments in this chapter are given in Figure 10.1.

10.1 The Tartarus Environment

A Tartarus board is a $k \times k$ grid, like a checkerboard, with impenetrable walls at the boundary. Each square on a Tartarus board contains nothing, a box, or the robot (henceforth called the *dozer*). A valid starting configuration in Tartarus consists of m boxes together with a placement and heading of the dozer. In the starting configuration, no boxes are in a block of 4 covering a 2×2 area of the board, and no box is adjacent to the wall. The dozer starts away from the wall and can be heading up, down, left, or right. An example of a valid Tartarus starting configuration is given in Figure 10.2.

The goal of Tartarus is for the dozer to shove the boxes up against the walls. On each move, the dozer may go forward, turn left, or turn right. If a single box with space to move into is ahead of it, then, when it moves forward, the box moves as well. If a dozer is facing a wall or a box already against a wall or a box with another box in front of it, a go forward move does nothing.

We will use a fitness function called the *box-wall* function for our work in this chapter. Given a valid initial configuration with $k = m = 6$, the dozer is allowed 80 (or more) moves. Each side of a box against a wall, after all the moves are completed, is worth 1 point to the dozer. This means that a box in the corner is worth 2 points; a box against the wall, but not in the corner, is worth 1. Add the scores from a large number of boards to get the fitness value used for evolution. Figure 10.3 shows 4 boards with scores. The maximum possible score for any Tartarus board with 6 boxes is 10: boxes in all 4 corners and the other 2 boxes against the wall.

Definition 10.1 *A starting configuration of a Tartarus board is said to be* **impossible** *if its score is 4 or less no matter what actions the dozer takes.*

The starting configurations with 2×2 blocks of 4 are impossible, which is why they are excluded.

In the later sections, we will be evolving and testing various sorts of representations of dozer controllers in the Tartarus environment. We will need working routines to support and display the Tartarus environment as well as a baseline for measuring performance. In this section, we will explore the Tartarus environment *without* genetic programming. These non-GP experiments

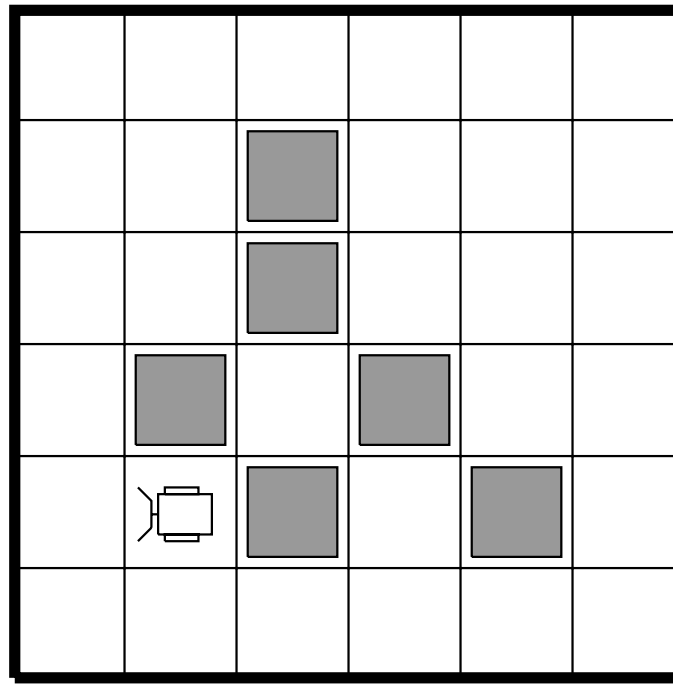

Fig. 10.2. A 6×6 Tartarus board with $m = 6$, dozer heading left.

will serve as a baseline for comparison of the various genetic programming techniques.

Experiment 10.1 *Build or obtain routines for maintaining and displaying the Tartarus environment. Develop a data structure for holding Tartarus boards that saves the positions of the boxes and the dozer's position and heading. Include the following routines in your software.* **MakeBoard** *should create a $k \times k$ Tartarus board with m boxes and an initial dozer position and heading in a valid starting configuration.* **CopyBoard** *should copy Tartarus boards.* **Move** *should take a board and a move (turn left, turn right, or go forward) and update the box positions and dozer position and heading.* **Score** *should compute the box-wall fitness function of a board, given a set of moves.* **DisplayBoard** *should print out or display a Tartarus board.*

For $k = m = 6$ (a 6×6 world with 6 boxes), generate 40 valid starting Tartarus configurations and save them so they can be reused. Use each board 20 times. Randomly generate 320 moves (turn right, turn left, and go forward). Record the fitness at $80, 160, 240,$ and 320 moves, and compute the average score. Do this experiment with the following 3 random number generators:

(i) Equal chance of turning left, turning right, and going forward,
(ii) Going forward 60%, turning left 20%, turning right 20%,

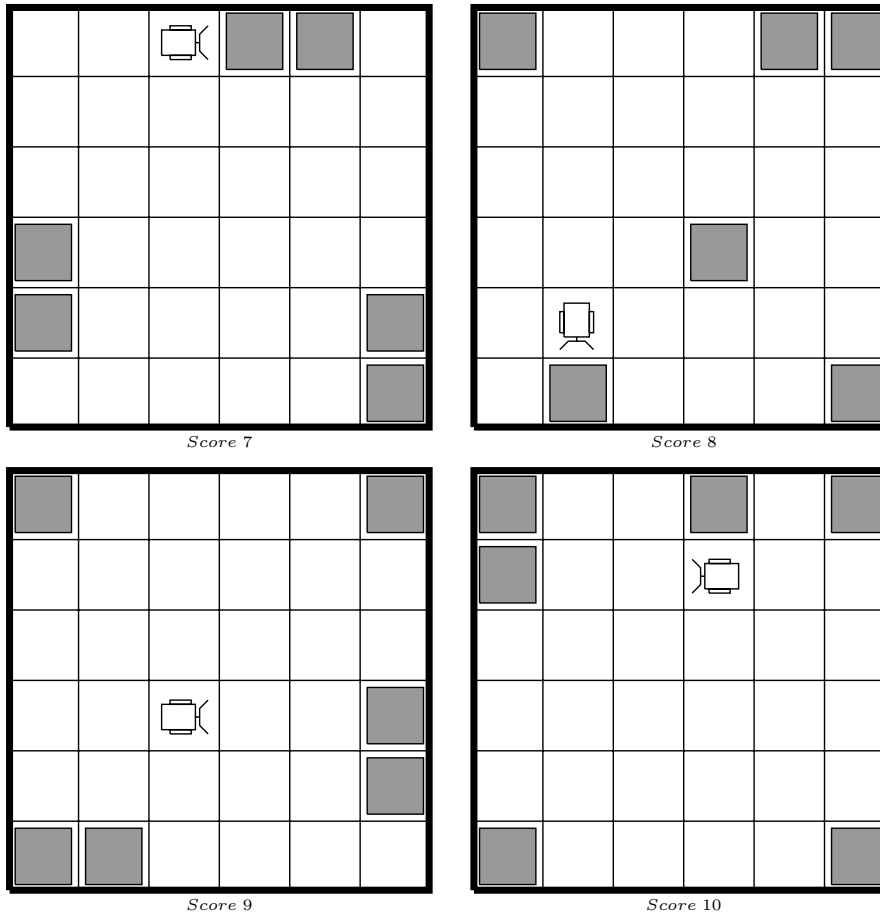

Fig. 10.3. Boards after 80 moves with scores.

(iii) A left turn never follows a right turn, a right turn never follows a left turn, but everything equally likely otherwise.

Report the average scores in a table indexed by the number of moves and types of random number generators. Explain why this forms a baseline experiment for Tartarus; would you expect nonrandom dozers to do better or worse than random moving points?

Experiment 10.1 gives us the basic routines for maintaining Tartarus. We now move on to a minimal Alife technology for Tartarus, the string evolver.

Experiment 10.2 *This experiment fuses string-evolver technology with the Tartarus routines from Experiment 10.1. Write or obtain software for an evolutionary algorithm for evolving strings over the alphabet* $\{L, R, F\}$*. Use a population of* 400 *strings of length* 80 *evolving under tournament selection*

Problem 389. Programming Problem. Write a short program that efficiently enumerates the members of the set G of genes from Problem 387. Using this program, report the number of such genes of length $n = 1, 2, \ldots, 16$. For debugging ease, note that there are 23,522 such genes of length 12.

Problem 390. Essay. The advantage of using aligned crossover operators is that the population can tacitly agree on the good values for various locations and even agree on the "meaning" of each location. Nonaligned crossover (see Definition 7.21) disrupts the position-specificity of the locations in the string. Consider the following three problems: the string evolver from Chapter 2 on the reference string

011010010011011001011101,

real function optimization on a unimodal 8-variable function with a mode at $(1, 2, 3, 4, 5, 6, 7, 8)$, and the string evolver in Experiment 10.6. Discuss the pros and cons of using nonaligned crossover to solve these problems.

Problem 391. Essay. Consider the three techniques discussed in this section: gene doubling, nonaligned crossover, and imposing restrictions (e.g., "no LR or RL") on the strings used in the initial population. Discuss how these techniques might help or interfere with one another.

10.2 Tartarus with Genetic Programming

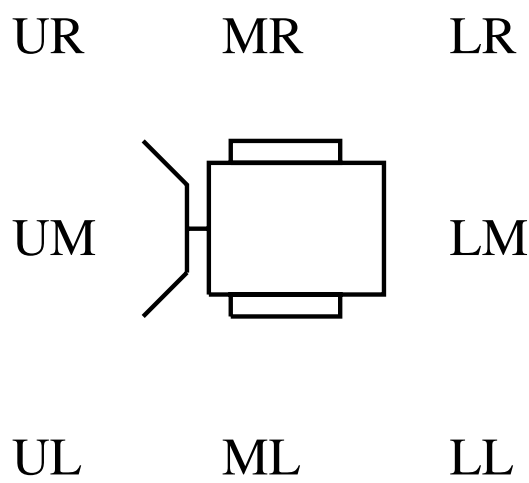

Fig. 10.4. Dozer sensor terminal placement.

In this section, we will develop a system for evolving parse trees to control the dozer. The data type for our parse trees will be the integers (mod 3) with

the translation to dozer actions $0 = L$, $1 = R$, $2 = F$. The terminals include the constants 0, 1, and 2. In most of the experiments, we will use 8 "sensor" terminals that will report what is in the squares adjacent to the dozer. These terminals are UM (upper middle), UR (upper right), MR (middle right), LR (lower right), LM (lower middle), LL (lower left), ML (middle left), and UL (upper left). The positions are relative to the dozer, not absolute. Figure 10.4 shows on which square, relative to the dozer, each sensor terminal reports. A sensor terminal returns 0 for an empty square, 1 for a box, and 2 for a wall. We also may use a terminal called RND that returns a uniformly distributed random number. The terminals are summarized in Table 10.1.

Name	Type	Description
0, 1, 2	constants	The integers (mod 3)
$UM, UR, MR, LR,$ LM, LL, ML, UL	sensors	Reports on a square adjacent to the dozer
RND	special	Returns a uniformly distributed constant (mod 3)

Table 10.1. Dozer GP language terminals.

We will be using a number of operations in the dozer GP language, changing which are available in different experiments. All the experiments in this section will use unary increment and decrement (mod 3), addition and subtraction (mod 3), a binary maximum and minimum operation that imposes the usual order on the numbers 0, 1, and 2, and a ternary if-then-else operator that takes zero as false and nonzero as true. These operations are summarized in Table 10.2. This list will be substantially extended in the next section, so be sure to make your code able to handle the addition of new operations and terminals.

If you wrote and documented the parse tree manipulation code used in Chapter 8, you will be able to modify it for use in this chapter. We will need the same parse tree routines as those used in Experiment 8.1, but adapted to

Name	Type	Description
INC	unary	Adds one (mod 3) to its argument
DEC	unary	Subtracts one (mod 3) from its argument
ADD	binary	Adds its arguments (mod 3)
SUB	binary	Subtracts its arguments (mod 3)
MAX	binary	Returns the largest of its two arguments, $0 < 1 < 2$
MIN	binary	Returns the smallest of its two arguments, $0 < 1 < 2$
ITE	ternary	If first argument is nonzero, returns second argument; otherwise returns third argument

Table 10.2. Dozer GP language operations.

the terminals and operations given above. The first experiment will test the technique of genetic programming in the Tartarus environment using the 8 environmental sensors. The next experiment will test the effects of adding the RND random number terminal. The parse tree manipulation routines should be able to allow and disallow the use of the RND terminal.

There is some entirely new code needed: given a board including dozer position and heading, compute the values of the 8 terminals used to sense the dozer's environment.

Experiment 10.7 *Look at the list of tree manipulation routines given in Experiment 8.1. Create and debug or obtain software for the same set of routines for the terminals and operations given in Tables 10.1 and 10.2, and also terminal and operation mutations (see Definitions 9.3 and 9.4). In this experiment,* do not *enable or use the RND terminal.*

Using these parse tree routines, set up a GP system that allows you to evolve parse trees for controlling dozers. Recall that the three possible outputs are interpreted as 0*, turn left;* 1*, turn right; and* 2*, go forward. You should use a population of* 120 *trees under tournament selection with a probability of* 0.4 *of mutation. Your initial population should be made of parse trees with* 20 *program nodes, and you should chop any parse tree with more than* 60 *nodes.*

Sum the box-wall fitness function over 40 *boards to get fitness values, remembering to test each dozer controller in a generation on the same boards, but generate new boards for each new generation. Evolve your populations for* 100 *generations, saving the maximum and average per-board fitness values in each generation as well as the best parse tree in the final generation of each run. Do* 30 *runs and plot the average of averages and average of best fitness.*

Answer the following questions:

(i) How do the parse trees compare with the strings from Section 10.1?
(ii) Do the "best of run" parse trees differ from one another in their techniques?
(iii) Are there any qualitative differences between the strings and the parse trees?

Now we again ask the question, can evolution make use of random numbers? In this case, we do so by activating the RND terminal in our GP software.

Experiment 10.8 *Modify your parse tree routines from Experiment 10.7 to include the RND terminal. Do the same evolution runs, answer question (ii) from Experiment 10.7, and compare the results of this experiment with those obtained without the RND terminal.*

It is of interest to know to what degree the parse trees in the "best of run" file from Experiment 10.8 are simply generating biased random numbers as opposed to reacting to their environment.

Experiment 10.9 *Build or obtain a piece of software that can read in and evaluate the parse trees saved from Experiment 10.8 and determine the fraction of moves of each type (turn left, turn right, or go forward) and detect the use of RND and of the sensor terminals. Do* 100 *evaluations of each parse tree confronted with no adjacent boxes, one adjacent box in each of the* 8 *possible locations, a wall in each of the* 4 *possible positions, and each of the* 20 *possible combinations of a box and a wall. Answer the following questions:*

(i) Do the parse trees act differently in the presence of boxes and walls?
(ii) What is the fraction of the parse trees that use the RND terminal?
(iii) What is the fraction of the parse trees that use (some of) the sensor terminals?
(iv) Do any parse trees use only sensors or only the RND terminal?

In a paragraph or two, write a logically supported conjecture about the degree to which the parse trees under consideration are only biasing mechanisms for the RND terminal.

Now, we will revisit the automatically defined function. The ADF is the GP structure acts like a subroutine. This is accomplished by adding a second parse tree to the structure of the dozer controller. As in Section 9.4, the second parse tree contains the "code" for the ADF. In the "main program" parse tree we add a new operation called ADF. An example of a parse tree and its accompanying ADF are shown in Figure 10.5. In our implementation, ADF will be a binary operation: there will be two new terminals called x and y in the ADF parse tree that have the values of the two arguments passed to the ADF. This will require fairly extensive modifications of our GP software. These include the following:

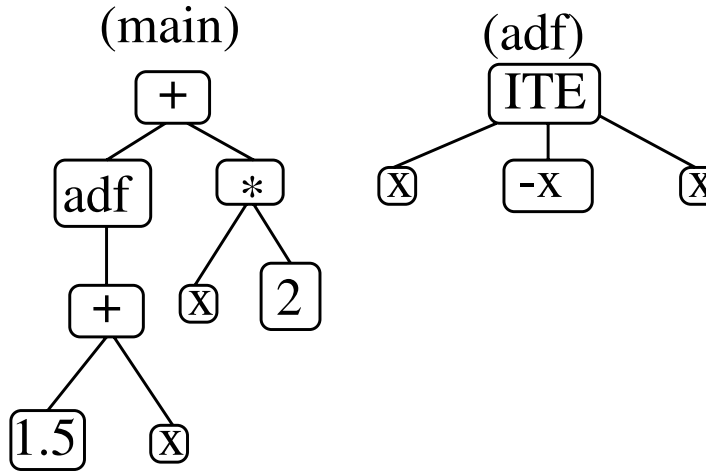

Fig. 10.5. Main parse tree and ADF parse tree.

- We must be able to maintain two separate (but closely related) GP languages for the main and ADF parse trees. In the main parse tree, there is a new binary operation called ADF. In the parse tree describing the ADF, there are two new terminals x and y.
- The parse tree evaluator must be able to evaluate the two parse trees appropriately. Whenever an ADF operation is encountered in the main parse tree, the evaluator must compute the values of its arguments and then evaluate the ADF parse tree with those values used for x and y.
- Randomly generated dozer controllers now contain two parse trees, and all the utility routines must work with both trees.
- Our mutation operator must now choose whether to mutate the main or the ADF parse tree.
- Crossover should do one of three things:
  - Cross over the main parse trees of the parents.
  - Cross over the ADF parse trees of the parents.
  - Take the main and ADF trees from distinct parents in both possible ways.

The use of ADFs in genetic programming gives many of the benefits of subroutines in standard programming. They are small pieces of code that can be called from several locations in the main parse tree. A good ADF can evolve and spread to multiple different members of the population, roughly a form of code reuse. A program with an ADF is more nearly modular and may therefore be easier to mine for good ideas than a single large parse tree. Let us see to what degree using ADFs helps our dozer controllers.

Experiment 10.10 *Make the modifications needed to allow the software from Experiment 10.8 to use ADFs. Use a starting size of* 20 *nodes for the main parse trees and* 10 *nodes for the ADF trees. Chop the main parse trees when their size exceeds* 40 *nodes, and the ADF trees when their size exceeds* 20.

Redo the runs for Experiment 10.8 using ADFs both with and without the RND terminal. Set the mutation rate to 0.4. *Half the time mutate the ADF; half the time mutate the main parse tree. For crossover, cross over the ADF with probability* 0.4, *the main tree with probability* 0.4, *and simply trade the main tree and ADF tree with probability* 0.2.

In your write-up, compare the performance of the 4 *possible types of evolved controllers, with and without RND and with and without ADF.*

There is another issue to treat before we conclude this section on plain genetic programming for Tartarus. Crossover is a very disruptive operation in genetic programming. In a normal evolutionary algorithm, it is usually the case that if we cross over two identical creatures, then the resulting children will be identical to the parents. This is manifestly not so for genetic programming, as we saw in Problem 291.

The standard crossover operation in genetic programming, subtree crossover, is an example of a *nonconservative* crossover operator.

In the theory of evolutionary computation as presented in the earlier chapters, the role of crossover was to mix and match structures already in the population of creatures rather than to generate new ones. The GP operator is far more capable of creating new structures than a standard (conservative) crossover operator. It is thus probably a good idea to use null crossover, the do-nothing crossover defined in Chapter 2, a good deal of the time. When applying null crossover, the children are simply verbatim copies of the parents.

Experiment 10.11 *Rebuild the software from Experiment 10.8 to use standard crossover with probability p and null crossover with probability $(1-p)$. Redo the data acquisition runs for $p = 0.1, 0.5$ and also use the runs done in Experiment 10.8, which can be viewed as $p = 1.0$ runs. Compare performance and redo the sort of analysis done in Experiment 10.9 on the best-of-run files from all 3 sets of runs.*

Problems

Problem 392. Which subsets of the set of the 7 operations given in Table 10.2 are complete sets of operations for $\mathbb{Z}_3$? A set S of operations is said to be *complete* if any operation with any number of arguments can be built up from the members of S.

Problem 393. We can view constants as being operations that take zero arguments. Taking this view, there are 3 operations named 0, 1, and 2 in our GP language. If we add these 3 operations to those in Table 10.2, then which subsets of the resulting set of 10 operations are complete? (See Problem 392 for a definition of complete.)

Problem 394. Essay. Why would it be nice to have a complete set of operations in a GP language? Is it necessary to have such a complete set? Is there any reason to think having more than a minimal complete set of operations is nice?

Problem 395. Give a clear description of (or pseudocode for) a conservative crossover operation for use on parse trees. Compare its computational cost with the standard crossover operator.

Problem 396. Suppose we have a GP language with only binary operations and terminals, including a binary ADF. If the ADF and the main parse tree between them possess a total of exactly n nodes, give a formula for the maximum number of operations executed in evaluating a parse tree pair. Assume that every instruction of the ADF must be reexecuted each time it is called by the main parse tree.

Problem 397. Prove that the total number of nodes n in the main parse tree and ADF in Problem 396 is even.