Chapter 10

Tartarus: Discrete Robotics

In this chapter, we will deal with an environment called *Tartarus* in which a virtual robot (thought of as a bulldozer) shoves boxes around trying to get them up against the walls. The Tartarus environment was proposed by Astro Teller in his paper *The Evolution of Mental Models* [35]. In the first section, we will use a string evolver to test the support routines that maintain the Tartarus environment. In the second section, we will use genetic programming (GP) to run the bulldozer. In the third, we will modify the GP-language to allow the programs a form of memory like the one in the PORS environment. In the fourth section, we will use a different sort of memory, that of Chapter 6. This last modification has a large number of applications that will appear in later chapters.

This chapter absolutely 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 implemetation of the parse tree evaluation routines much easier than in Chapter 9.

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 10.1.

The goal of Tartatus is for the dozer to shove the boxes up against the walls. On each

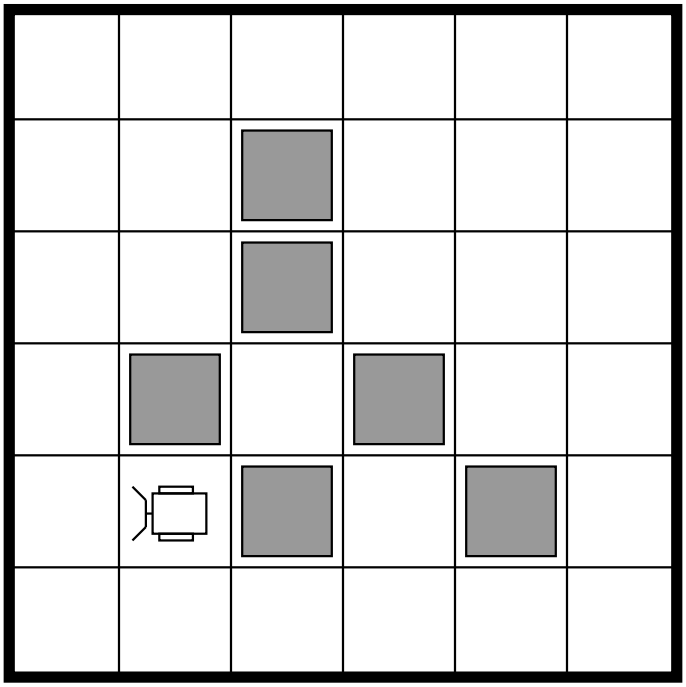


Figure 10.1: A 6×6 Tartarus board with $m = 6$, dozer heading left

move, the dozer may go forward, turn left, or turn right. If a single box with empty space 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, 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 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 10.2 shows 4 boards with scores.

The maximum possible score for any Tarturus 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 Tarturus 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

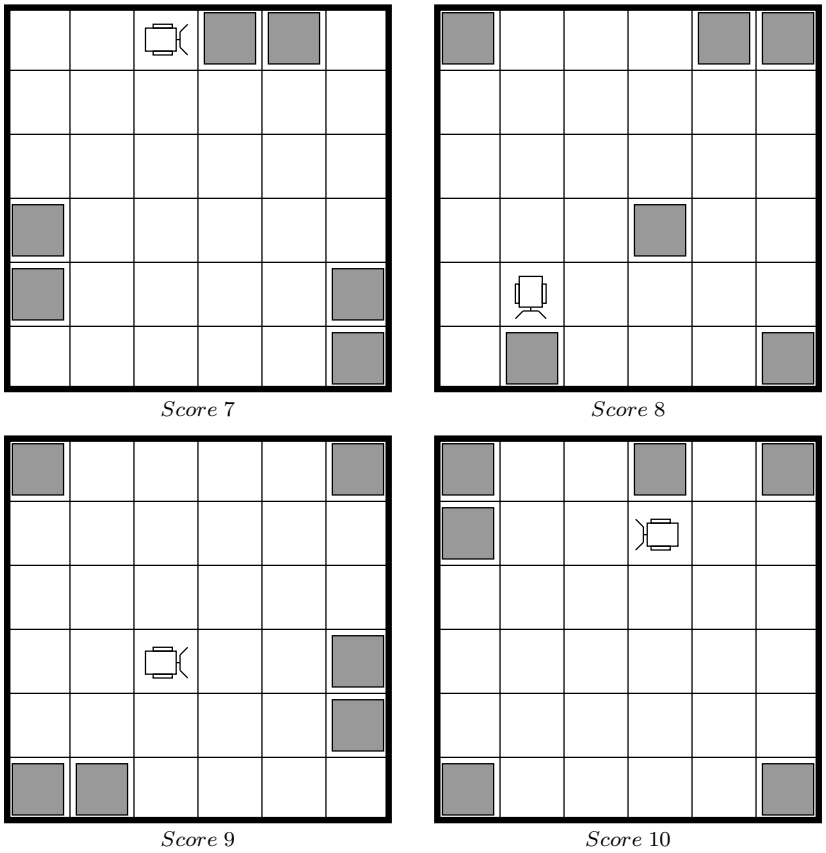


Figure 10.2: Boards after 80 moves with scores

section, we will explore the Tartarus environment *without* genetic programming. These non-GP-experiments 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%

(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 with two point crossover and one of one, two, or three point mutation. Each string specifies 80 moves for the dozer to make. Evaluate the fitness of a string by testing it on 12 Tartarus boards and summing the scores on the boards to get a fitness value. All strings in the population should be evaluated on the same 12 boards in a single generation. A new set of 12 boards should be created for each generation. Run the algorithm for 100 generations.*

Graph the average fitness divided by 12 (the average per-board score) and the fraction of L, R, and F in the population, averaged over 20 runs, for each of the 3 mutation operators. Summarize and explain the results.

At this point, we digress a bit and do two experiments that serve to illustrate an odd feature of evolution and use a very interesting mutation operator first introduced by Kristian Lindgren in a paper on the Iterated Prisoner's Dilemma [27].

Experiment 10.3 *A string with less than 80 moves can still be used to control a dozer for 80 moves by cycling through the string. Modify the software from Experiment 10.2 to use strings of any length up to 80. Using only one point mutation, rerun Experiment 10.2 for strings of length 5, 10, 20, and 40. Compare the fitness and average final fitness with those for the one-point-mutation 80-character strings in Experiment 10.2. Keeping in mind that every shorter string gives behavior exactly duplicated by a longer string, explain what happened.*

For the next experiment, we need the notion of *neutral mutations*. A neutral mutation is one that does not change a creature's fitness. The null mutation, mentioned in Chapter 3, is an example of a neutral mutation, but it does nothing. In his work on the Prisoner's Dilemma, Lindgren came up with a neutral mutation that changes the creature's genotype substantially without changing its phenotype.

Lindgren stored a Prisoner's Dilemma strategy in a lookup table indexed by the last several plays. He doubled the length of the lookup table by increasing the index set by one play and then making a new lookup table from two copies of the old lookup table, one for each possible value of the new index. The additional index was vacuous in the sense that, no matter what that index said, cooperate or defect, the creature's response was the same.

"Why bother?" I hear you cry. Imagine we add Lindgren's doubling mutation to a population of Prisoner's Dilemma players evolving under an evolutionary algorithm. Occasionally, one of the creatures doubles the size of its gene. Its behavior does not change, but the strategies that can be derived from it under a point mutation change radically.

Look back at the results of Experiment 10.3. Evolution finds it much easier to optimize a short string than a long one. If we optimize a short string, double its length, and then continue the optimization in a completely new space (with *constructively* the same fitness), we may be able to enormously speed the process of evolution.

If we include a gene doubling mutation operator in the mix of genetic operations, then we will need another that cuts genes in half. The reason for this has to do with the behavior of one-sided random walks. Imagine that you flip a coin and stand still for heads, while moving to the right for tails. The net effect will be substantial movement to the right. If we occasionally double a gene and never shorten one, there is danger that all genes will attain some maximum length, *before* the search of the smaller gene lengths is complete.

Definition 10.2 *A* **gene doubling mutation** *for a string type gene is a mutation that doubles the length of the string by concatenating two copies of the string.*

Definition 10.3 *A* **gene halving mutation** *for a string type gene is a mutation that halves the length of the string by splitting the string into its first and second halves and saving one at random.*

Not that we have defined gene doubling and gene halving mutations, we must confront the problem of crossing over pairs of genes of different lengths. There are at least three ways to do this, given in the experiments below.

Experiment 10.4 *Modify the software from Experiment 10.3 in the following fashion. First, modify the string data structure so that it includes the length of the string. Second, modify the crossover operator so that when two strings of different length cross over, the crossover points are chosen inside the smaller string. Perform mutation so that 90% of the time the algorithm does a single point mutation, 5% of the time the algorithm does a gene doubling*

mutation, and 5% of the time the algorithm does a gene halving mutation. Generate the initial population with a 50% chance each of being length 5 or 10. Ignore gene halving mutations on strings of length 5 and gene doubling mutations on strings of length 80.

In addition to average fitness and fraction of types of moves, save the average fitness of each length of string and plot the average of averages within length classes over the course of evolution. Also, give a histogram of the string lengths in the final generation, summed over all runs. In your write up, state if better strings were located at all, if better strings of length 80 were located, and if evolution was faster.

There is an obvious bias in the crossover operator above. It favors, to some degree, genetic material in the latter part of the strings for preservation as an intact chunk. This is offset to a modest degree by the ability of the gene halving mutation to "bring the back half forward," but this may not help. If the moves in the front part of the genes are "more evolved" (better), then exhumed back halves may have trouble competing. A crossover operator that is more difficult to code, but perhaps more fair, is explored in the next experiment.

Experiment 10.5 *Repeat Experiment 10.4 with a new crossover operator as follows. When two strings of differing lengths are crossed over, copy the shorter gene into a scratch variable and double its length, until it is the same length as the longer gene. Cross over the two resulting genes, now the same length, and then truncate one of the resulting strings so that it is the length of the shorter parent gene. This operator provides one child of length matching each parent. Present the same data as in Experiment 10.4 and discuss the impact of changing the crossover operator.*

Definition 10.4 *A crossover operation is said to be* **conservative** *if two identical parents cross over to create identical children. Such crossover operators are also sometimes called* **pure**.

One thing that distinguishes evolutionary algorithms that use string genes from genetic programming is the conservative nature of the crossover operator. Crossing over of strings, even in our last two experiments, lines up equivalent positions in the string. Information doesn't migrate around the gene and identical parents clone themselves when they are reproduced and crossed over. Subtree crossover in genetic programming, on the other hand, does not preserve positions. Identical parents produce radically different children. In the next experiment, we will strike off into the middle ground between conservative string crossover and subtree crossover by using *nonaligned* crossover.

Nonaligned crossover exchanges substrings of the same length that may not be lined up in the two strings. This permits genetic information to move about the collective genome of the population over time.

Experiment 10.6 *Repeat Experiment 10.4 using nonaligned crossover. Present the same data as in Experiment 10.4 and discuss the impact of changing the crossover operator. Compare also with Experiment 10.5, if you performed it.*

Problems

Problem 10.1 Essay. *In Experiments 10.2-10.4 we saved data on the fraction of moves of each type. Explain a way of using that data to modify and improve performance in Experiment 10.1.*

Problem 10.2 *For $k = 6$ and $m = 4, 5,$ and 6, compute the probability that placement of the boxes away from the wall, but otherwise at random, will produce an invalid starting configuration (4 boxes in a 2×2 group).*

Problem 10.3 Essay. *Given the answer to Problem 10.2, is it worth excluding the impossible configurations? Consider both the cost in computer time and the effects on evolution in your answer.*

Problem 10.4 *Give a counting formula for the number of strings of length k over the alphabet $\{L, R, F\}$ in which:*

(i) L and R are never adjacent.

(ii) At least half the characters are F.

(iii) The sequences LLL and RRR never appear.

(iv) (i) and (iii) both hold.

Hint: find a recursion in terms of the string length.

Problem 10.5 *Explain why each of the classes of strings given in Problem 10.4 might be a good restricted class from which to draw dozer genes.*

Problem 10.6 *Compute the fitness of the following strings when run on the Tartarus board shown in Figure 10.1. Place the numbers 1-80 on an empty board to show where the dozer moves and give the final configuration of boxes. Warning: this is a very time-consuming piece of busy work; the purpose is to build mental machinery for debugging Tartarus code.*

(i) $FFFLF$,

(ii) $FFLFR$, and

(iii) $FFLFFRFLFR$.

Problem 10.7 Essay. *Refer to the material in Appendix A on Markov chains. Describe a genetic algorithm that evolves the matrix of a Markov chain controlling a dozer in the Tartarus environment. What is a reasonable set of states for such a Markov chain? Can a chain be designed that outperforms the 80-move version of Experiment 10.1?*

Problem 10.8 Essay. *In Chapter 8, we did some work with seeding populations. First, explain why the special strings in Problem 10.4 would be good for seeding a population for Experiments 10.2-10.4. Next, give a description of such an experiment, including pseudo-code for generating strings in classes (i)-(iv).*

Problem 10.9 *Suppose we are looking at string genes for the Tartarus problem studied in this section with $k = m = 6$. Define a set G of genes to have all of the following properties:*

(a) The longest substring of pure Ls or pure Rs is of length 2.

(b) The longest substring of pure Fs is of length 5.

(c) The substrings LR and RL never occur.

Which types of genes from Problem 10.4 appear in G? Prove that if we have a collection of boards B that we are using to test fitness of genes of length n, then, for every gene h of length n outside of G, there is a a gene $g \in G$ of length n, such that g scores as much as h on the collection B.

Problem 10.10 Short Essay. *In what sense is the set G of genes in Problem 10.9 like the set $\mathcal{T}_s^*$ from Definition 8.3?*

Problem 10.11 Programming Problem. *Write a short program that efficiently enumerates the members of the set G of genes from Problem 10.9. Using this program, report the number of such genes of length $n = 1, 2, ..., 16$. For debugging ease, note that there are 23,522 such genes of length 12.*

Problem 10.12 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 10.13 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.*

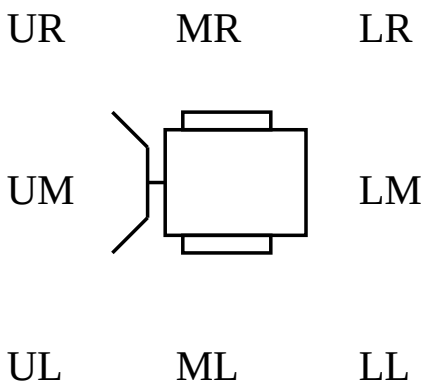


Figure 10.3: Dozer sensor terminal placement

10.2 Tartarus with Genetic Programming

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 10.3 shows on which square, relative to the dozer, each sensor terminal reports. If a square is empty, the terminal returns a 0; a box returns a 1; a wall returns a 2. We also may use a terminal called RND that returns a uniformly distributed random number. The terminals are summarized in Table 10.2.

Name	Type	Description
0, 1, 2	constants	The integers (mod 3)
$UM, UR, MR, LR,$ LM, LL, ML, UL	sensors	Report on squares 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

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	trinary	If first argument is nonzero, returns second argument, otherwise returns third argument

Table 10.2: Dozer GP-language operations

and minimum operation which imposes the usual order on the numbers 0, 1, and 2, and a trinary if-then-else operator which 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 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 10.2 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) If there any qualitative difference 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. Advanced students should test the effects of biasing the random numbers the RND terminal returns.*

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.

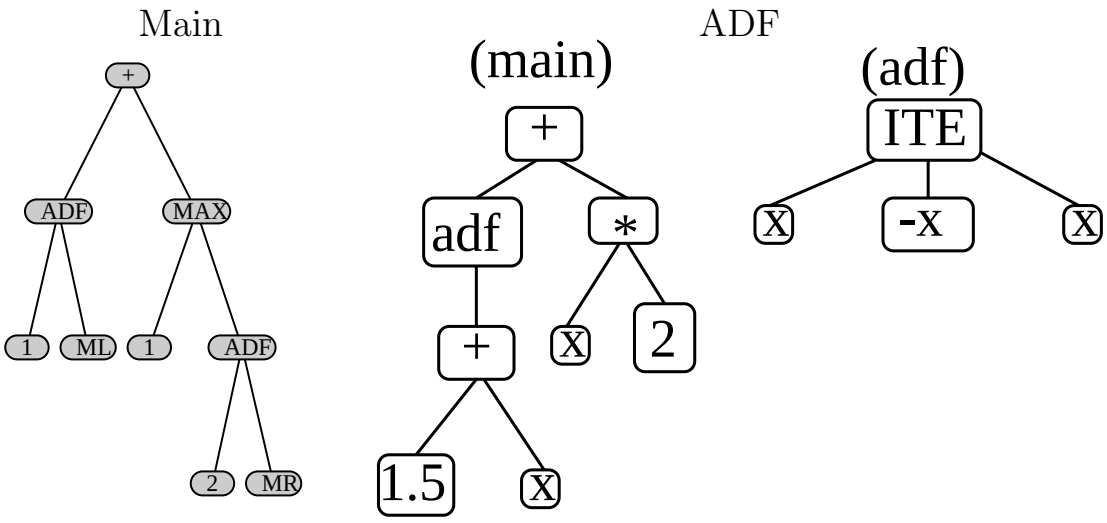


Figure 10.4: Main parse tree and ADF parse tree

Now, we will revisit the automatically defined function. The ADF is the GP-structure that takes the place of subroutines. This is accomplished by adding a second parse tree to the structure of our 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 10.4. In our implementation, it will be a binary operation. In the ADF parse tree, there will be two new terminals called x and y which have the value of the two arguments passed to the ADF. This will require fairly extensive modifications of our GP-software. These include the following.

- 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 two parse trees appropriately. Whenever an ADF operation is encountered in the main parse tree, the evaluator computes the values of its arguments and then evaluates 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 mutate one or the other of two parse trees used to describe a dozer controller.

- Crossover should do one of three things:
  - Cross over the main parse trees of the parents
  - Cross over the ADF parse trees
  - 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 8.3.

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 10.14 *Which subsets of the set of 8 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 10.15 *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 11 operations are complete? (See Problem 10.14 for a definition of complete.)*

Problem 10.16 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 10.17 *Give a clear description of (or pseudo-code for) a conservative crossover operation for use on parse trees. Compare its computational cost with the standard crossover operator.*

Problem 10.18 *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 exactly n, give a formula for the maximum number of operations executed in evaluating a parse tree pair. Assume every instruction of the ADF must be re-executed each time it is called by the main parse tree.*

Problem 10.19 *Prove the total number of nodes n in the main parse tree and ADF in Problem 10.18 is even.*

Problem 10.20 *Evaluate the parse tree with ADF shown in Figure 10.4 for:*

(i) $ML = MR = 0$,

(ii) $ML = 1,\ MR = 0$,

(iii) $ML = 0,\ MR = 1$,

(iv) $ML = MR = 1$,

(v) $ML = 2,\ MR = 0$,

(vi) $ML = 0,\ MR = 2,$

(vii) $ML = 1,\ MR = 2,$ *and*

(viii) $ML = 2,\ MR = 1.$

Problem 10.21 *Write by hand a parse tree in the GP-language used in this section that will exhibit the following behaviors in the presence of isolated blocks:*

- *If a block is in the square to the left or right of the dozer, the dozer will turn towards it.*
- *If a block is behind the dozer, the dozer will turn.*
- *If a block is ahead of the dozer, the dozer will go forward.*

You may use an ADF, if you wish.

Problem 10.22 *Take the parse tree you wrote for Problem 10.21 and diagram its behavior for 80 time-steps on the board shown in Figure 10.1.*

Problem 10.23 Essay. *In Problem 10.21, you were asked to create a parse tree that exhibited three behaviors. This is an ambiguous task; each behavior responds to a particular environmental event, but these events are not always distinct. First, explain how many events the three behaviors requested are responding to and detail carefully which combinations of the events can happen at the same time. Second, establish and defend a precedence structure on the behaviors that will yield a good score for Problem 10.22. Advanced students should instead defend their precedence structure against the entire space of possible boards.*

Problem 10.24 *When you did Problem 10.21, you probably noticed that the dozer can push a single box over to the wall and get stuck. Assume that the answer to Problem 10.21 was done as a single parse tree, and that that parse tree is being used as a terminal ADF (an ADF that takes no arguments). Write a main parse tree that returns the output of its ADF with probability p and otherwise does something random (i.e., returns an evaluation of RND) for at least 5 different values of p, using no more than 20 nodes in the main parse tree.*

10.3 Adding Memory to the GP-language

The setup of the Tartarus world makes it impossible for the dozer controller, in many circumstances, to tell the difference between pushing a box forward and pushing it in a futile fashion against a wall. How long should a dozer push a box before giving up? How can a dozer controller know how long it has been pushing a block? There are two broad approaches

to this problem. One is for the dozer controller to learn to count to the width of the world minus one; the other is for the controller to avoid instructing the dozer to push a box more than once without turning. The latter strategy may require more time-steps to accomplish its goal, but is much simpler for evolution to discover. To see this, try to come up with a method of counting in any of the GP-technologies used so far. Tricky, isn't it?

In his paper *the Evolution of Mental Models*[35], Astro Teller proposed to add to his dozer control language a new type of structure called *indexed memories*. He had 20 memories which were accessed by a binary store operation (which took as arguments the number to store and an index into the memories) and a unary recall operation (which took as an argument an index into the memories). We will use a much simpler form of memory, mimicking that used for the PORS problem in Chapter 8. We will add a varying number of new unary operations and terminals for using one or more memory locations with our GP-language.

Experiment 10.12 *Add to the GP-language we have been using so far, a STO and RCL instruction, as in Chapter 8. The store operation is unary. It places a number (mod 3) into an external memory location and also returns the number stored as its value. The RCL operation is a terminal returning the contents of the external memory. Initialize the external memory to zero.*

With this modified GP-language, use the same evolution as in Experiment 10.8 to test the utility of having a memory. We had 4 sorts of GP-languages for Tartarus available before we added the memory: with and without ADF and with and without the RND terminal. Adding in a memory gives a total of 8 possible GP-languages, 4 of which we have already tested. Do evolutionary runs for one of these 4 possibilities. (Later experiments will refer to the memory-with-ADFs version of this experiment.)

- *memory alone,*
- *memory with ADFs,*
- *memory with RND terminals available, or*
- *memory with ADFs and RND terminals available.*

For those runs you perform, compare performance with other available data. If you have runs using and not using the RND operator, discuss the degree to which the RND terminal can interfere with use of the memory. Place your discussion in an evolutionary context: in your opinion, does use of the RND terminal create local optima that arrest the development of memory-using strategies?

Having a single memory gives the dozer controller, in some sense, three internal states: 0, 1, or 2 stored in the memory. Since the version of Tartarus we are using is 6 squares

across, this may not be enough. It may well be worth trying more memories. We will try two distinct techniques: indexing 3 memories and making 3 separate pairs of memory manipulation terminals available.

Experiment 10.13 *Modify the software from Experiment 10.12 to have 3 unary operations STO1, STO2, and STO3 and 3 terminals RCL1, RCL2, and RCL3 that access 3 separate memories. Do the evolutionary runs with ADFs, but no RND terminals and compare the results with the single memory version.*

The next experiment will be more like the indexed memories used by Teller. We will transform the unary store instruction into a binary store instruction and the recall terminal into a unary recall operation.

Experiment 10.14 *Rebuild the GP-language used in Experiment 10.12 to use 3 memory locations as follows. Instead of a unary store operation STO, have a binary store operation so that (STO x y) places the value x into memory location y. Instead of a recall terminal RCL have a unary recall operation so that (RCL x) returns the contents of memory x. Do the same evolutionary runs with ADFs but no RND terminals and compare the results with the single memory version and with Experiment 10.13.*

So far, we have evolved dozer controllers for the Tartarus problem that have access to two types of resources: sensory information and memory. If you look carefully, you can find controllers with only sensory information, with only memory, and with both. To our surprise, memory is more valuable, by itself, than sensory information, at least as they appear in the implementations of this chapter. This in turn raises the questions: "What other types of memory are available?" and "Is there another collection of sensory information we could present to the controllers?" We will address this issue in the Problems and attempt an exploratory answer to the question "How much memory is enough memory?"

Problems

Problem 10.25 *Using the version of the GP-language from Experiment 10.14, write by hand a dozer controller to attack the board shown in Figure 10.1. We recommend that this problem be worked in small groups. Cruel instructors may wish to assign grades competitively.*

Problem 10.26 *Identify and explain your identification of dozer controllers from the chapter so far that:*

(i) have memory only,

(ii) have sensory information only, and

(iii) have both.

Problem 10.27 *How many internal states (internal means not depending on the values returned by sensor terminals) can a dozer controller in Experiment 10.13 or 10.14 have?*

Problem 10.28 *Define a* state *of the Tartarus board to be a set of possible positions for the boxes together with a position and heading of the dozer. Compute the number of possible states of Tartarus for $k = m = 6$ (6×6 board with 6 boxes). How many memories, storing a value (mod 3) would a GP-language require to give the dozer controller that many internal states?*

Problem 10.29 Essay. *Reread Problem 10.28. Discuss the value of having at least one internal state per external state. There are three things you may wish to consider: the problem of recognizing a state of the world, the problem of managing the information involved, and the effect symmetries of the board may have. Based on your discussion, try to estimate a useful number of states.*

Problem 10.30 *In his experiments, Teller had 20 indexed memories which could store a value in the range $0 \leq n \leq 19$. Based on your answer to Problem 10.28, how many internal states do Teller's dozer controllers have per possible external state of the world?*

Problem 10.31 *Refer to Problem 10.7, if you did it. A Markov chain controller (Markov chains are discussed in Appendix A) for a dozer is a 3 state Markov chain with the states corresponding to the 3 actions the dozer uses: turn right, turn left, and go forward. Suppose we were using a language with a single memory and the RND terminal, as in Experiment 10.12, then which of the following Markov chain dozer controllers could we simulate in that language? The Markov chains are represented by their transition matrices indexed in the order L(eft), (R)ight, and (F)orward. Give a constructive proof of your answer when possible.*

$$\textit{(i)}\begin{bmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 0 & 0 \end{bmatrix} \quad \textit{(ii)}\begin{bmatrix} 0 & 0 & 1 \\ 0 & 0 & 1 \\ 1/2 & 1/2 & 0 \end{bmatrix} \quad \textit{(iii)}\begin{bmatrix} 1/3 & 0 & 2/3 \\ 0 & 1/3 & 2/3 \\ 1/3 & 1/3 & 1/3 \end{bmatrix}.$$

Problem 10.32 *For each of the Markov chains given in Problem 10.31, compute which strings of moves they can produce. You answer should be a specification of a subset of the set of all strings over the alphabet $\{L, R, F\}$.*

Problem 10.33 *Give the transition matrix of a Markov chain controller that can produce sequences of moves where the following rules hold:*

- *no more than 2 left turns in a row,*
- *no more than 1 right turn in a row,*

- *no more than 5 moves ahead in a row,*
- *no right turns follow left turns,*
- *no left turns follow right turns.*

Hint: how many states do you need, minimum?

Problem 10.34 Programming Problem. *Write a program that takes as input a parse tree and produces a C, C++, Pascal, or Lisp (your choice of one) routine that takes a sensor state of Tartarus as input and returns a dozer move as output, in a fashion exactly that of the automaton used as input. This is a parse tree compiler.*

10.4 Tartarus with GP-Automata

In this section, we will fuse the finite state automata we learned to evolve in Chapter 6 with parse trees to produce an artificial life technology called a GP-automaton. Given the obvious desirability of being able to count to small numbers, a very easy and natural function for a finite state automaton, it would be nice if we could adapt finite state automata to Tartarus.

In Chapter 6, we used finite state automata with a small input and output alphabet. Consider a finite state automaton used to control a dozer. The output alphabet would be simply $\{L, R, F\}$. What about the input alphabet? Since there are 8 terminals that sense adjacent squares, each of which can take on 3 values (0-empty, 1-box, 2-wall), there are naively $3^8 = 6561$ possible inputs. The true number is smaller, as you are asked to compute in Problem 10.35, but there are still many hundreds of possible inputs to the dozer in the Tartarus environment, a dauntingly large "input alphabet" for our putative finite state automaton.

If we were to implement a finite state automaton in the tabular fashion of Chapter 6 then we would require several hundred columns in the transition table, each specifying the next state and action for a possible state of the sensor terminals. This means each data structure would have a number of genetic loci in the thousands. This structure is too large to use with an evolutionary algorithm which finishes in a reasonable amount of computer time.

The natural solution to this dilemma is to filter the input data down to a manageable number of objects, *before* presenting it to the finite state controller. In effect, we want to compress the input bandwidth of Tartarus down to something manageable. If we were working with real-valued input data, for example, we could divide the data into a few ranges and use those as the input alphabet for our finite state automaton. With Tartarus, it is not obvious what sort of data compression would be natural, and so we will leave the matter to evolution.

In sections 10.2 and 10.3, we have already created a system for building and testing potential bandwidth compression devices: parse trees! Our parse trees took a set of sensor terminal values and distilled from them a move: turn left, turn right, or go forward. Those parse trees with a fitness of 1 or more were able to make at least one fairly sensible decision for at least one configuration. If we fuse those decision makers with a finite state automaton that can do exactly the sort of counting that is natural for the Tartarus environment, we may reap great performance improvements.

Name	Type	Description
$0, 1, 2$	terminal	Possible input values
$UM, UR, MR, LR,$ LM, LL, ML, UL	terminal	Report on squares adjacent to the dozer
ODD	unary	Return 1 if the argument is odd, 0 otherwise
COM	unary	Compute one minus the argument (complement)
$\sim$	unary	Unary minus
$+, -$	binary	The usual addition and subtraction
$=, <>, >=, <=, >, <$	binary	Comparisons that return 0 for false and 1 for true
MAX, MIN	binary	maximum and minimum
ITE	trinary	if-then-else, 0=false and 1,2=true

Table 10.3: Decider language, terminals and operations

Definition 10.5 *A* **GP-automaton** *is a finite state automaton that has a parse tree associated with each state.*

You should review Section 6.1. The GP-automata we use for Tartarus are defined as follows. Each GP-automata has n states, one of which is distinguished as the initial state. As with standard finite state automata, we will have a transition function and a response function. The response function will produce moves for the dozer from the alphabet {**L**, **R**, **F**}. The transition function is based on the parity of integers (odd, even), with the input values being produced by parse trees called *deciders*. Each state has a decider (parse tree) associated with it. These deciders will be integer-valued parse trees using operations and terminals as given in Table 10.4. Their job is to "look at" the Tartarus board and send back a small amount of information to the finite state automaton. In use, we will evaluate a decider on the current Tartarus board and use its output to drive the finite state automaton's transition and response functions.

The data structure used for the GP-automaton is an integer specifying the initial state, together with an array of states. Each state is a record containing: a decider, an array of the responses that state makes if the decider returns an odd or even number, and the next

Start: 3→9			
	If Even	If Odd	Deciders
0	2→8	1→2	$(\sim (\sim (<= 1 (\sim 1))))$
1	3→7	2→8	(Odd UL)
2	2→11	0→11	LR
3	2→9	2→10	$(<= UL$ (Odd (Odd ML)))
4	2→9	1→5	(Odd $(\sim$ (min LR $(\sim ML)))))$
5	2→8	3→3	$(<= (= UL\ 0)$ (Odd -2))
6	0→3	3→9	(Odd (+ (max LM -1) 1))
7	3→1	3→2	(ITE (~ 1) MR (Odd UM))
8	2→4	0→5	(Com $(\sim (<=$ (Odd UL) $LL)))$
9	0→0	1→3	(Com (Odd MR))
10	2→4	2→8	(Com $(\sim (+ LL (\sim (\sim MR)))))$
11	0→5	0→3	(min LM (max (Odd 2) MR))

Figure 10.5: A 12 state GP-automaton for controlling a dozer in Tartarus (Responses and transitions are given in the form $response \rightarrow transition$.)

state to go to if the decider returns an odd or even number. Notice that the GP-automata for Tartarus do not require an initial action. As we shall see, this data structure is selected with an eye toward the genetic operations we will be using. An example of a GP-automaton is given in Figure 10.5.

One interesting property of GP-automata, in contrast to parse trees, is the divorce of input and output type. This divorce is inherited from finite state automata and represents a potentially valuable difference from pure parse tree systems. While it is always possible to interpret the output of a parse tree to change its type, i.e., 0=L, 1=R, 2=F, the process can rest on quite arbitrary decisions.

Genetic Operations on GP-automata

One of the bugaboos of genetic programming is the crossover operator. With the parse trees appearing in several separate parts of the GP-automaton, we need no longer mix and match our entire data structure when doing crossover. The crossover operator we will use for GP-automata is performed by treating the array of states as a string and simply letting the initial state of each parent be copied to the corresponding child. We thus may use any of the string crossover operators from Chapter 2 on GP-automata: one point, two point, as well as the more exotic operators.

As crossover becomes more straightforward, it is appropriate that mutation become vexed. There are a large number of fairly natural mutation operators that make sense for

GP-automata. There are two classes of mutations: mutation of the nondecider parts of the finite state controller and mutation of the deciders.

Definition 10.6 *To do a* **finite state point mutation** *of a GP-automaton, choose uniformly at random one of: the initial state, a single transition, or a single response. Replace it with a valid random value.*

Definition 10.7 *To do an* **exchange mutation**, *exchange two uniformly chosen deciders.*

Definition 10.8 *To do a* **replacement mutation**, *copy one uniformly selected decider over another.*

Definition 10.9 *To do a* **cross mutation**, *perform a subtree crossover operation on a pair of uniformly selected deciders.*

Definition 10.10 *To do a* **decider point mutation**, *perform a normal parse tree point mutation (subtree replacement) on a decider.*

In our first experiment in this section, we will simply get some GP-automata evolving. In the second experiment, we will explore the utility of the various mutation operators. In the third experiment, we attempt to characterize the behavior of the various controllers evolved. In the fourth experiment, we will make a substantial modification to the GP-automata paradigm to allow the possibility of extended computation. In the fifth, we take another look at population seeding, but with a new twist.

Experiment 10.15 *Build or obtain routines for handling GP-automata. This should include all 5 of the mutation operators given above, as well as two point crossover on the array of states. The deciders should be parse trees using the operations and terminals given in Table 10.4. Build an evolutionary algorithm that operates on a population of 60 GP-automata controlled dozers with 8 states. Use tournament selection with tournament size 4 as your model of evolution. Do one mutation on each new dozer controller, split evenly among the 5 mutation operators given and null mutation (doing nothing). Chop any deciders that grow to have more than 20 nodes. For fitness, evaluate each controller on 40 Tartarus boards in the usual fashion.*

Run the algorithm for 200 generations saving the average and best fitness of each population in each generation and the fraction of actions of each type. Do 20 runs and save the best controller in the final generation. Compare the per-board fitness with other experiments you have done. For comparison, write a program that tests each of the best-of-run controllers on 5000 Tartarus boards. The average over 5000 boards can be used to rate controllers in a stable fashion. In your write up, graph the average, over populations and runs, of the average and average best fitness in each generation.

With GP-automata software in hand, we are ready to try exploring the various mutation operators.

Experiment 10.16 *Take the software from Experiment 10.15 and rebuild it to allow you so set the probability of using each mutation operator. Redo the experimental runs but with the following mixes of mutation operators (use all the mutation operators in a run equally often).*

(i) Finite state point mutation and decider point mutation.

(ii) Finite state point mutation and cross mutation.

(iii) Finite state point mutation, cross mutation, and exchange mutation.

Report the same data as in Experiment 10.15 and compare the performances of the two experiments.

The next experiment is intended to help you understand the results of the preceeding two experiments. If things are going as they did for us, then using GP-automata caused a substantial increase in performance. It would be nice to try to achieve some degree of qualitative understanding of this increase.

Experiment 10.17 *To help understand and analyze your results, write or obtain a lab program that can read in the best-of-run GP-automata from Experiments 10.15 and 10.16. This program should allow you to test the GP-automata on a number of boards and also should allow you to watch the dozer move, time-step by time-step, on boards that you specify. The details are left to you, and the project should be graded in part on the design of the lab. Use the lab to develop a taxonomy of dozer behaviors. Write up an explanation of how (and how well) each type of behavior you found works.*

In many ways a GP-automaton is more like a normal computer program than a standard GP-parse tree. Instead of a single statement (parse tree) living in a trivial or iterative control structure, the program represented by a GP-automaton has many statements (deciders and actions) that live inside a nontrivial control structure (the finite state automaton). One important difference between a standard computer program and a GP-automaton is that a computer program may execute many statements before it produces output, while a GP-automaton produces one piece of output per statement (decider) executed.

It is not too hard to extend the GP-automata paradigm to include the execution of multiple deciders before producting output. In standard finite state automata, there are sometimes transitions that produce no response. These are called λ-*transitions* after λ, the empty string. We will add a new action, the *null action* to the responses that the GP-automata can make. When a GP-automaton produces a null action, it immediately makes

the transition to the next specified state and executes it normally, without waiting for the next time-step, a form of λ-transition. This means that null actions can allow the execution of multiple deciders per time-step in the GP-automata. There is one potential pitfall: infinite loops. A GP-automaton that uses null actions has the potential to get stuck in a loop in which one null action leads to another forever. To finesse this potential problem, we will simply cut off a GP-automaton, if it evaluates more than same fixed number of deciders in a time-step.

Experiment 10.18 *Modify the software from Experiment 10.15 to allow null actions and extended computation in the fashion described above. Cut off the GP-automaton, if it uses 8 null actions in a row and have the dozer sit in place for a time-step in this event. Do that same evolution runs as in Experiment 10.15 and compare performance.*

There is an interesting point implied by our attempt to improve performance by using null actions. In the null action of GP-automata, there are 3 outputs possible from each decider (the GP-language), but 4 possible actions. In a GP-automaton, it is possible to divorce actions from computations from inputs. This means that, without type checking or the attendant computational complexity, it is possible to use 3 distinct data types. When dealing with discrete events in a real-parameterized situation that requires discrete responses, the advantage of this separation of the 3 classes of data (input, computational, and output) will become more pronounced.

At the beginning of this section, while setting the stage for GP-automata, we noted that using finite state automata in the Tartarus environment is quite tricky due to the large size of the input bandwidth, and then cheerfully noted that the parse tree technology from the preceding sections gave us a very natural bandwidth compression technology. We then used random parse trees in our GP-automata, even though we have software that could give us evolved GP-automata. In the next experiment, we will take the natural step of creating our initial populations of GP-automata from pre-evolved parse trees. In order to do this next experiment, you will need the software from Experiment 10.7.

Experiment 10.19 *First change the software from Experiment 10.7 to have a chop limit of 20 nodes and to use the language used by the deciders in this section. Interpret the integer output (mod 3) to produce actions. Perform 10 runs, saving the entire final population of all the runs into a single file. Now, modify the software from Experiment 10.15 to use parse trees chosen randomly from the file, instead of randomly generated parse trees. Perform the same evolutionary runs as in Experiment 10.15 and compare the performance. Be sure to account for, or at least mention, the additional computation done before the GP-automata in this experiment started evolving.*

There are, as usual, a large number of other experiments one could perform with the technologies in this chapter. Students looking for projects might try exploring the mutation

operators more completely, changing the board size, changing the fitness function, or changing the task being attempted by the dozers. Variation of task will be done in a large way in Chapter 12. It also might be of interest to see how performance varies as the number of states in an automaton are varied. Finally, we have made no effort to test GP-automata with access to randomness or memories in their deciders. We will come back to the idea of GP-automata in different environments in later chapters.

Problems

Problem 10.35 *Using any of the GP-languages from Section 10.2 or 10.3, the dozer controller may "see" a large different number of configurations of adjacent squares. Compute how many such configurations there are.*

Problem 10.36 *Is the crossover operator given in this section for GP-automata conservative? See Definition 10.4.*

Problem 10.37 *Give an example of a GP-automata application in which it would be valuable to exploit the divorcing of input, computation, and output data type, so as to use 3 separate data types.*

Problem 10.38 Essay. *Consider a parse tree dozer controller and a GP-automaton dozer controller. Assuming both have the same number of nodes of parse trees somewhere inside, which executes more code per time-step on average? Treat both models of GP-automaton computation given in this section (with and without null actions).*

Problem 10.39 Programming Problem. *Write a program that takes as input a GP-automaton and produces a C, C++, Pascal, or Lisp (your choice of one) routine that takes a sensor state of Tartarus as input and returns a dozer move as output, in a fashion exactly that of the GP-automaton used as input. This is a GP-automaton compiler.*

Problem 10.40 Essay. *Explain the possible evolutionary utility of the exchange and replacement mutation operators. If available, support your ideas with data from the experiments.*

Problem 10.41 *Redo Problem 10.25 using 4-state GP-automata similar to those used in Experiment 10.18.*

Problem 10.42 Essay. *In this chapter, we enhanced standard genetic programs with random number terminals, ADFs (subroutines), memory, indexed memory, and finite state automata. Describe techniques for and discuss the possible advantages and disadvantages of adding stack manipulation operations and terminals to the GP-languages used in the Tartarus environment. Assume an operation PUSH that puts x on the top of the stack and returns the value of x, and terminals POP (pops the stack), TOP (reports the value on the top of the stack), and EMP (returns 0, if the stack is not empty, 1, if it is).*

Problem 10.43 *Explain the sense in which the string genes are finite state automata. Where is the state information stored? How many transitions are there out of each state when a string gene is viewed as a finite state automaton?*

Problem 10.44 *Give a procedure for constructing a GP-automata with n states that has the property that its transition and response functions duplicate the behavior of a string of length n.*

Problem 10.45 Essay. *Suppose we have a source of excellent string genes of length 8. Would you expect a population generated by the technique suggested in Problem 10.44 to do better compared to the random populations use in Experiment 10.15: (i) at first, and (ii) by the end of the experiment?*

Problem 10.46 Essay. *Review Problem 10.30. In* The Evolution of Mental Models*, Teller reports fitnesses in the ballpark of 4.5, averaged over a large number of boards. If all went as it should, the best GP-automata you evolved in this section got fitnesses up in the area of* $5 - 6.5$*. In light of the various string baseline experiments in Section 10.1, what is a good number of states to use?*

10.5 Allocation of Fitness Trials

How many Tartarus boards should we use to evaluate the fitness of a dozer controller? We dealt with a similar problem in Chapter 5 when evaluating the fitness of symbots (see Problem 5.10). While we want to have a controller that does well on all 300,000+ possible Tartarus boards, we cannot afford to test it on them all. If we do not use a large number of boards, there is a danger that an inferior controller will outscore a superior controller on the Tartarus boards we happen to pick. The conflict then is between computer time needed to test controllers and quality of fitness evaluation needed to save the superior controllers. Let us verify that this problem exists.

Experiment 10.20 *Take the best-of-run GP-automata that you saved in Experiment 10.15. Rank order them by the average per-board fitness they achieved in the evolutionary algorithm that produced them. After that, fix a set of 5000 Tartarus boards selected uniformly at random, and evaluate the GP-automata's per-board average fitness on those boards. Rank order them according to the new fitness numbers. Did the order change?*

If your experiment went as ours did, a 40-board test does not produce the same rank ordering as a 5000-board test does. It is not unreasonable to imagine that this problem is more acute for better controllers. With a population of random controllers that have not undergone evaluation and selection, it is likely that most of them will be quite unfit. Testing

on only a few boards is enough to sort out the slightly-more-fit minority. At the other end of evolution, we have many controllers that are quite good, and we are likely to be making fine distinctions among them. Since the best controller in the population may have to survive for dozens of fitness evaluations before a better controller arrives, it has dozens of chances to die from an unlucky selection of test boards. With this thought in mind, we propose our first experiment in wisely allocating fitness trials.

Experiment 10.21 *Ask your instructor which evolutionary algorithms from this chapter are to be modified to perform this experiment. Good ones would be Experiments 10.3, 10.7, 10.13, or 10.15. Replace the fitness evaluation in the evolutionary algorithms you are going to modify with a fitness evaluation that uses 10 boards for the first generation and a number of boards that is the larger of 10 or 10 times the maximum average per-board fitness in the preceding generation, rounded to the nearest integer, thereafter. Report the effect, if any, on the average and standard deviation (over runs performed for a given setup) of the maximum fitness in the final generation.*

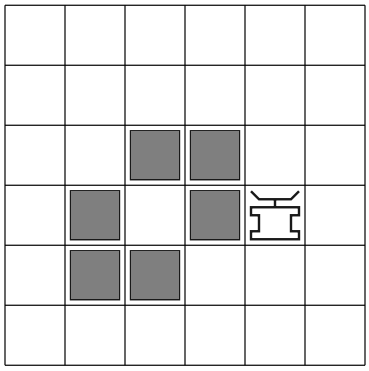
Let us turn our attention now from the issue of how many boards to the issue of *which* boards. The ideal situation would be to locate a small collection of boards with the property that if a dozer controller does well on those boards, then it will do well on the full set of boards. The problem is that there is no *a priori* way to know which those boards are. We can turn to nature for inspiration in this regard, by transforming our noisy optimization problem of finding a good dozer controller into a co-evolution of boards and controllers.

The basic idea is this. Any 6×6 Tartarus board has 10 points of fitness "at risk" (the maximum possible score for a dozer on that board). Whenever a dozer meets a board, those 10 points are divided between the dozer and the board: the dozer gets points as usual, and the board gets those points not earned by the dozer. This gives us a fitness function for boards; boards with high fitness are those on which it is difficult for the dozers to score points. An *impossible board* is one on which it is impossible for *any* dozer to score 10. Now, we can evolve the boards. Recall that a "board" is a placement of 6 boxes with no close group of 4 boxes, together with a placement and heading of the dozer.

Experiment 10.22 *Rebuild the software from Experiment 10.7 to use an evolving population of boards in the following manner. Have 40 slots for boards and fill them with 40 distinct boards (allow no repetition). Make sure that no board with a close group of 4 is included in the group. For fitness evaluation, test all dozer controllers on all 40 of the boards, awarding fitness to the boards in the manner described in the text. After the fitness evaluation, sort the boards by fitness and delete the k worst boards, replacing them with new ones. Save the best-of-run dozer controllers for each run and compare them with the results of Experiment 10.7 using the software developed in Experiment 10.20. Perform this experiment for* $k = 2, 10,$ *and* 20*. In addition to saving the best dozer controller in each run, save the highest scoring board in the final generation.*

Problems

Problem 10.47 *The evolutionary algorithm used to find difficult boards in Experiment 10.22 works by replacing the worst members of the population of boards with new random boards. Give a design for an evolutionary algorithm that would function more efficiently, in your opinion, at locating difficult boards.*



Problem 10.48 *Examine the board above. Prove that the maximum score possible on the above board is 4. This board was discovered by Stephen Willson and is an example of an impossible board.*

Problem 10.49 Essay. *When we excluded boards with close groups of 4 boxes, it was because such configurations are impossible. If we are evolving difficult boards as in Experiment 10.22, then the system may locate impossible ones. Comment on whether an evolutionary algorithm like the one you designed in Problem 10.47 or the minimal system used in Experiment 10.22 is more efficient at finding impossible boards.*

Problem 10.50 *In this chapter, we have used random number generators, strings, parse trees, parse trees with memories, and GP-automata as dozer controllers. If you were to build an evolutionary algorithm whose express purpose was to locate impossible boards, which of these would you pick as the controller? Would it be better to evolve the controller or to pick a set of really good controllers you had already evolved?*

Problem 10.51 *Construct another impossible configuration besides the one given in Problem 10.48 and anything involving a close-packed group of 4. You may make the board larger and use more boxes.*