Chapter 14

Cellular Encoding

Cellular encoding [16, 17] is a technique for representing an object as a set of directions for constructing it, rather than as a direct specification. Often, this kind of representation is easier to work with in an evolutionary algorithm. We will give several examples of cellular encoding in this chapter. The name "cellular encoding" comes from an analogy between the developmental rules governing construction of the desired objects and the biology governing construction of complex tissues from cells. The analogy is at best weak; don't hope for much inspiration from it. A more helpful way to think of the cellular encoding process is as a form of *developmental biology* for the structure described (as we did with Sunburn in Chapter 4).

Suppose we have a complex object: a molecule, a finite state automaton, a neural net, or a parse tree. A series of rules or productions that transform a starting object into an object ready to have its fitness evaluated can be used as a linear gene.

Instead of having complex crossover operators (which may require repair operators), we can use standard crossover operators for linear genes. The behavior of those crossover operators in the search space is often difficult to understand, but this is also often true of crossover operators used with direct encodings.

The idea of evolving a set of directions for constructing an object is an excellent one with vast scope. We will start by building 2-dimensional shapes using instructions from a linear gene. There are a number of possible fitness functions for such shapes; we will explore two. In the second section of this chapter, we will create a cellular encoding for finite state automata and compare it with the direct encodings used in Chapter 6. In the third section, we will give a cellular encoding method for combinatorial graphs. In the fourth section, we will give a method for using context free grammars to control the production of the parse trees used in genetic programming. This permits the evolution of simple linear genes rather than parse trees and allows the user to include domain specific knowledge in the evolutionary algorithm.

14.1 Shape Evolution

A *polyomino* is a shape that can be made by starting with a square and gluing other squares onto the shape by matching up sides. A production of a 3-square polyomino is shown in Figure 14.1. A polyomino with n squares is called an n-omino. Our first cellular encoding is a scheme for encoding n-ominos.

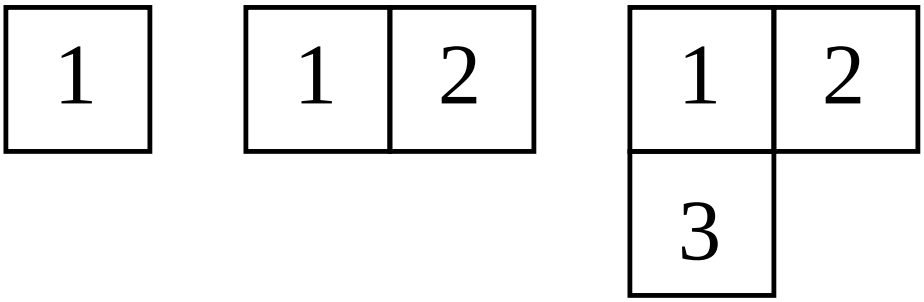


Figure 14.1: Start with initial Square 1; add Square 2; then, add Square 3 to make a 3-square polyomino.

We will use an array of integers as our encoding for n-ominos. The key is interpretation of the integers in the array. Divide each integer by 4. The integer part of the quotient is the number of the square in the n-omino; the remainder encodes a direction: up, down, left, or right.

Algorithm 14.1 Polyomino Development Algorithm
Input: *An array of integers G[] of length k*
Output: *A labeled n-omino and a number F of failures*

Details:
Initialize a $(2k+1) \times (2k+1)$ *array A with zeros;*
Place a 1 in the center of the array;
Initialize a list of squares in the n-omino with the 1;
Initialize C=1, the number of squares so far;
Initialize F=0, the failure counter;
For(i= 0*;* $i < k$*;* $i++$*)*
    Interpret G[k] $(mod\ 4)$ *as a direction X in (U,L,D,R);*
    Find square S of index (G[k]/4) $(mod\ C)$ *in the growing structure;*
    If(square in direction X from S in A is 0)
        C ¡- C+1;
        Put C in square in direction X from S in A;
    Else F ¡- F+1;

End For;
Return A,F;

Let's do a small example. We will use one-byte integers in this example ($0 \leq x \leq 255$), which limits us to at most 65 squares in the n-omino. This should suffice for the examples in this section; shifting to two-byte integers permits the encoding of up to 16,385-ominos, more than we need.

Example 14.1 *Examine the gene* $G = (126, 40, 172, 207, 15, 16, 142)$. *Interpret the gene as follows:*

Locus	*Interpretation*
*126=4*31+2*	*Direction 2(down) from Square 0(31(mod 1)); add Square 1*
*40=4*10+0*	*Direction 0(up)from Square 0(10(mod 2)); add Square 2*
*172=4*43+0*	*Direction 0(up)from Square 1(43(mod 3)); wasted*
*207=4*51+3*	*Direction 3(left)from Square 0(51(mod 3)); add Square 3*
*15=4*3+3*	*Direction 3(left)from Square 3(3(mod 4)); add Square 4*
*16=4*4+0*	*Direction 0(up)from Square 4(4(mod 5)); add Square 5*
*142=4*35+2*	*Direction 2(down)from Square 5(35(mod 6)); wasted*

The result of this interpretation is the 6*-omino:*

5		2
4	3	0
		1

with $F = 2$ *failures (wasted loci). We label the n-omino to track the order in which the squares formed. Notice that not all of the* 15×15 *array A is shown, in order to save space.*

Now that we have an array-based encoding for polyominos, the next step is to write some fitness functions. Our first fitness function is already available: the number of failures. A failure means that a gene specified a growth move in which the polyomino tried to grow where it already had a square. If our goal is to grow large polyominos, then failures are wasted moves.

Experiment 14.1 *Create or obtain software for an evolutionary algorithm that uses the array encoding for polyominos. Treat the array of integers as a string-type gene. Initialize the arrays with numbers selected uniformly at random in the range 0-255. Use arrays of length 12 with two point crossover and single point mutation. The single point mutation should replace one location in the array with a new number in the range 0-255. Use a population size of 400 with a steady state algorithm using single tournament selection of size 7.*

Record the number of tournament selections required to obtain a gene that exhibits zero failures for each of 100 runs of the evolutionary algorithm and save a 0-failure gene from each run. Report the time-to-solution and the shape of the resulting polyominos. Runs that require more than 100,000 tournaments should be cut off, and the number of such tournaments should also be reported.

Experiment 14.1 suffers from a problem common in evolutionary computation; it's hard to tell what the results mean. The only certain thing is that it is possible to evolve length 9 arrays that code for 13-ominos. One interesting question is: are these "typical" 13-ominos? It seems intuitive that some shapes will be better at avoiding failure than others. Let's develop some measures of dispersion for polyominos.

Definition 14.1 *The* **bounding box** *of a polyomino is the smallest rectangle that can contain the polyomino. For the polyomino in Example 14.1, the bounding box is a* 3×3 *rectangle. The* **bounding box size** *of a polyomino is the area of the polyomino's bounding box.*

Definition 14.2 *The* **emptiness** *of a polyomino is the number of squares in its bounding box not occupied by squares of the polyomino. The emptiness of the polyomino given in Example 14.1 is 3.*

Experiment 14.2 *Create or obtain software for a random n-omino creator that works in the following fashion. Start with a central square, as in the initialization of Algorithm 14.1. Repeatedly pick a random square in the array holding the polyomino until you find an empty square adjacent to a square of the polyomino; add that square to the polyomino. Repeat this square-adding procedure until the polyomino has n squares.*

The random polyominos will serve as our reference set of polyominos. Generate 100 random 13-ominos. For these 13-ominos and the ones found in Experiment 14.1, compute the bounding box sizes and emptinesses. If some runs in Experiment 14.1 did not generate 13-ominos, then perform additional runs. Compare histograms of the bounding box sizes and emptinesses for the two groups of shapes. If you know how, perform a test to see if the distributions of the two statistics are different.

The bounding box size is a measure of dispersion, but it can also be used as a fitness function. Remind yourself of the the notion of lexical fitness function from Chapter 5 (page 121).

Experiment 14.3 *Modify the software from Experiment 14.1 to maximize the bounding box size for polyominos. For length 12 genes (size 13 polyominos), the maximum bounding box has size 56.*

Do two collections of 900 runs. In the first, simply use bounding box size as the fitness. In the second set of runs, use a lexical product of bounding box size and number of failures in which bounding box size is dominant and being maximized, and the number of failures are being minimized. In other words, a polyomino with a larger bounding box size is superior, and ties are broken in favor of a polyomino with fewer failures.

Compare the time to find an optimal bounding box for the two fitness functions, and explain the results as well as you can. Save the best genes from each run in this experiment for use in a later experiment.

The shape of polyominos that maximize bounding box size is pretty constrained. They appear somewhere along the spectrum from a cross to a Feynman diagram. Our next fitness function will induce a different shape of polyomino.

Experiment 14.4 *Modify the software from Experiment 14.1 to be a generational algorithm that works with the following fitness function on a population of 60 polyominos with genes of length 19. Fitness evaluation requires an empty* 200×200 *array which wraps in both directions.*

Repeatedly perform the following steps. First, put the polyominos in the population into a random order. Taking each in order, generate a random point in the 200×200 *array. If the upper left corner of the current polyomino's bounding box is placed in that location and all squares of the polyomino can be placed, then place the polyomino there, marking those squares as full and adding the number of squares in the polyomino to its fitness. If the polyomino does not fit in the current location, try other locations by scanning first in the horizontal direction, until either you have tried all locations or a location is found where the polyomino fits.*

Once all the polyominos have had one try, a new random order is generated. Perform fitness evaluation until at least 75% of the 200×200 *array is occupied or until all shapes have had a chance to find a place in the array and failed. Do 100 runs of 1000 generations length and, comparing expressed shapes rather than genes, show and explain the most common shapes in each run. Are some shapes more common than others? Why?*

The fitness function used in Experiment 14.4 lets the shapes compete for space. There are two forces at work here: the need to occupy space and the need to fit into the remaining space. The former pressure should make large shapes, while the latter one will make small shapes. Consider how these pressures balance out when writing up your experiment.

Experiment 14.5 *Modify the fitness function from Experiment 14.4. If a shape does not fit at the randomly chosen location, do not try other locations. Go until the array is 50% full (rather than 75% full). Are the resulting shapes different from those found in Experiment 14.4?*

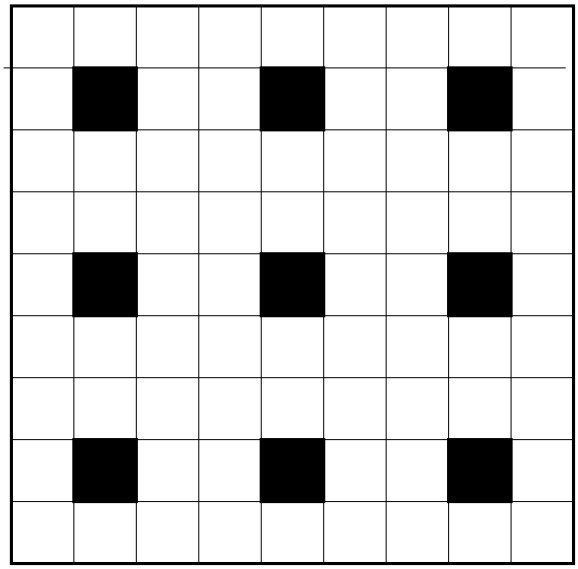


Figure 14.2: A 9×9 grid with all squares that have both coordinates congruent to 1 (*mod* 3) initially filled

The shapes obtained in our versions of Experiments 14.4 and 14.5 were not too different. Let's see if we can cause the experiment to produce a different sort of shape by modifying the fitness function again.

Experiment 14.6 *Modify the software from Experiment 14.5 so that there is a* 201×201 *array used for fitness evaluation in which the squares with both coordinates congruent to 1* (*mod* 3) *start already occupied. Are the resulting shapes different from those found before?*

The outcome of Experiments 14.4 and 14.5 suggest that compact shapes are favored. Let's try initializing Experiment 14.4 with genes that are not at all compact and see if we end up with a different sort of solution.

Experiment 14.7 *Modify the software from Experiment 14.4 to read in randomly selected genes chosen from those created during Experiment 14.3 instead of initializing with random genes. Are the resulting shapes any different from those obtained in Experiment 14.4?*

The shape gene is a simple example of cellular encoding and the experiments in this section are interesting mostly because of their coevolutionary character. The competitive exclusion game the shapes are playing when competing for space is a fairly complex game. You could generalize this system in other directions. Suppose, for example that we scatter shape "seeds" at the beginning of fitness evaluation and then grow shapes by executing one genetic locus per time-step of the development simulation. At that point, the partial shapes would need to guard space to additional development. This puts an entirely new dynamic into the shape's growth.

Problems

Problem 14.1 *Run the Polyomino Development Algorithm on the following length 7 polyomino genes.*

(i) G=(146, 155, 226, 57, 9, 84, 25)

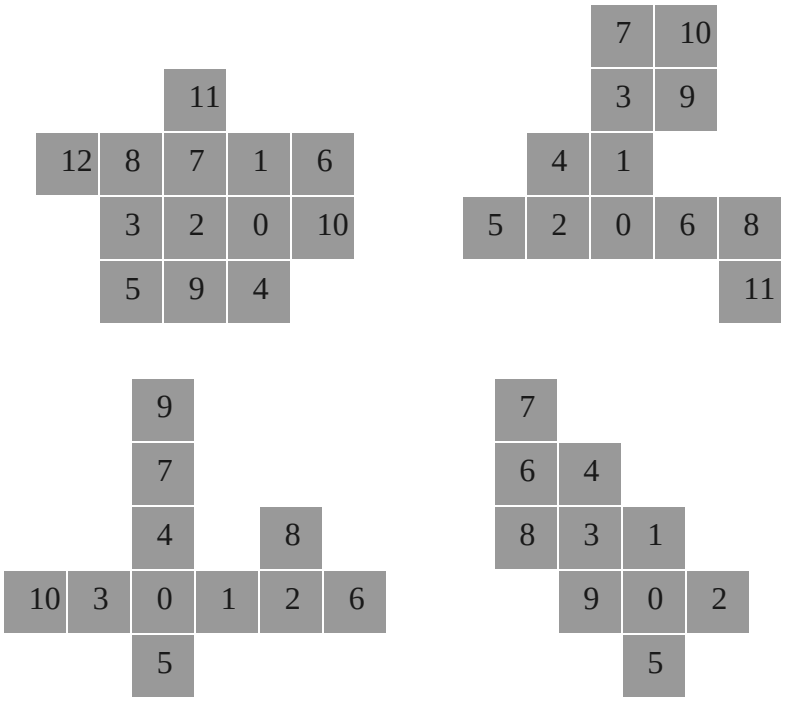
(ii) G=(180, 158, 146, 173, 187, 85, 200)

(iii) G=(83, 251, 97, 241, 48, 92, 217)

(iv) G=(43, 241, 236, 162, 250, 194, 204)

(v) G=(100, 139, 229, 184, 111, 46, 180)

Problem 14.2 *For each of the following polyominos, find a gene of length 12 that will generate that polyomino. The numbers on the squares of the polyomino give the order in which the squares were added to the polyomino during development. Your gene must duplicate the order in which the squares were added.*



Problem 14.3 *The point of cellular encoding is to specify a complex structure as a linear sequence of construction rules. Suppose that we instead stored polyominos in a 2-dimensional array. Create a crossover operator for polyominos stored in this fashion.*

Problem 14.4 *Consider a* 2×2 *square polyomino. Disregarding the gene and only considering the order in which the squares were added, how many different representations are there?*

Problem 14.5 *Enumerate all length 5 polyomino genes that code for a* 2×2 *square polyomino.*

Problem 14.6 *Give an example of a gene of length* k *that creates a polyomino of size 2 (for any* k*).*

Problem 14.7 *Prove that the maximum bounding box size for a polyomino with* n *squares is smaller than the maximum bounding box size for a polyomino with* $n + 1$ *squares.*

Problem 14.8 *For as many* n *as you can, compute the maximum bounding box size for an* n*-omino.*

Problem 14.9 *Reread Experiment 14.4. If a shape fails to find space once, is there any point in checking to see if it fits again? Would a flag array that marks spaces as having failed once speed up fitness evaluation?*

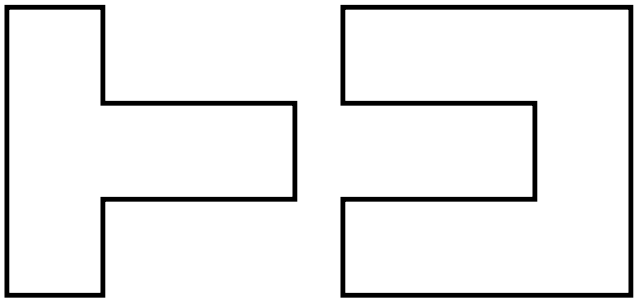
Problem 14.10 *The encoding given for shapes in this section is one possible choice. Try to invent an encoding for shapes (or an alternate algorithm for expressing the shapes) that eliminates wasted moves.*

Problem 14.11 *Does Experiment 14.4 need to be generational? If not, how would you modify it to be steady state?*

Problem 14.12 *In Experiments 14.4-14.6, why leave 25%-50% of the board unfilled?*

Problem 14.13 Essay. *In Experiment 14.4 and 14.5 we are placing shapes by two different methods and then evaluating them based on their success at filling space. Which strategy is better: fitting well with yourself or blocking others?*

Problem 14.14 Essay. *Does Experiments 14.4 or Experiment 14.5 favor compact shapes, like rectangles, more?*



Problem 14.15 Essay. *In Experiment 14.4, shapes are allowed to search for a place they will fit. It's not too hard to come up with complementary shapes that fit together, e.g., the two shown above. Would you expect populations of coexisting shapes that fit together but have quite dissimilar genes to arise often, seldom, or almost never?*

Problem 14.16 Essay. *Since different shapes are evaluated competitively in Experiments 14.4-14.7 the algorithms are clearly coevolutionary rather than optimizing. If most of the genes in a population code for the same shape, does the algorithm behave like a converged optimizer?*

14.2 Cellular Encoding of Finite State Automata

The evolution of finite state automata was studied in Chapter 6. We evolved finite state automata to recognize a periodic string of characters and then used finite state automata as game playing agents. In this section, we will examine what happens when we use a cellular encoding for finite state automata. With polyominos we started with a single square and added additional squares. In order to "grow" a finite state automaton, we will start with a single-state finite state automaton and modify it to make a larger automaton. In order to do this, we will need editing commands. We will work with automata with k possible inputs and outputs, named $0, 1, \ldots, k-1$. When we need a specific input or output alphabet (like $\{C, D\}$ for Prisoner's Dilemma), we will rename these integer inputs and outputs to match the required alphabet.

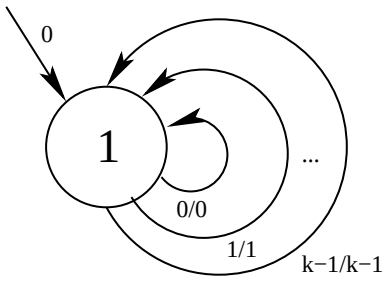


Figure 14.3: The Echo machine (initial action is zero; last input is its current output)

The starting automaton we will use is the *Echo* machine shown in Figure 14.3. While editing a finite state automaton, we will keep track of the *current state* being edited. The current state will be denoted with a double circle in the state diagram. The current state specifies where editing is to happen, the position of a virtual editing head. The cellular representation will consist of a sequence of editing commands which either modify the automaton or move the current state. Most of the editing commands take a member of the input alphabet of the automaton as an argument and are applied to or act along the transition associated with that input. This specifies unambiguous actions, because there are exactly k transitions out of the current state, one for each possible input. (The exception, B, is an editing command that modifies the initial response, no matter which state is the current state.)

The commands we will use to edit finite state automata are given in Table 14.1. They are only one possible set of editing commands for finite state automata. We chose a small set of commands with little redundancy that permit the encoding of a wide variety of finite state automata.

The pin command (P_n) requires some additional explanation. This command chooses one of the transitions out of the state currently being edited and "pins" it to the current state.

Command	Effect
B (Begin)	Increment the initial action.
F_n (Flip)	Increment the response associated with the transition for input n out of the current state.
M_n (Move)	Move the current state to the destination of the transition for input n out of the current state.
D_n(Duplicate)	Create a new state that duplicates the current state as the new destination of the transition for input n out of the current state.
P_n (Pin)	Pin the transition arrow from the current state for input n to the current state. It will move with the current state until another pin command is executed.
R (Release)	Release the pinned transition arrow, if there is one.
I_n	Move the transition for input n out of the current state to point to the state you would reach if you made two transitions associated with n from the current state.

Table 14.1: Commands for editing finite state automata (Incrementing is always modulo the number of possible responses.)

That means that if the current state is moved with an M_n command, then the transition arrow moves with it. This state of affairs continues until either the transition arrow is specifically released with an R command, or until another pin command is executed. (The definition permits only one arrow to be pinned at a time, though it would be possible to pin one arrow of each type unambiguously if the release command also took arguments.) If a transition arrow is still pinned when the editing process ends, then the arrow is left where it is; it is implicitly released when the current state ceases to have meaning, because the editing process ends.

The I_n command is the only command other than the pin command that can be used to move transition arrows. The command I_n moves a transition arrow to the state that the automaton would reach from the current state if two transitions were made in response to inputs of n. (This edit is difficult to perform with the pin command for some configurations of automaton.)

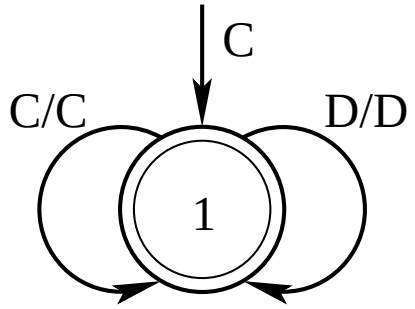
We reserve for the Problems the question of completeness of this set of editing commands. A set of editing commands is *complete*, if any finite state automaton can be made with those commands. Even with a complete set of commands, the "random" finite state automata we can generate are very different from those we obtain by filling in random valid values on blank automata with a fixed number of states. When filling in a table at random, it is quite easy to create states that cannot be reached from the initial state. An automaton created

with editing commands is much less likely to have many isolated states.

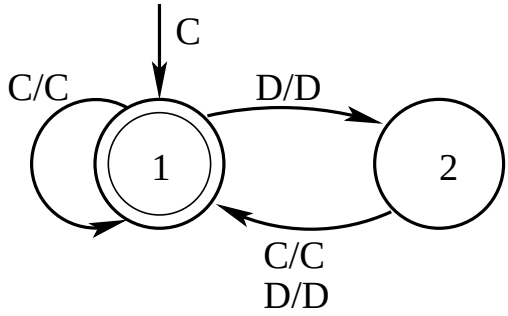
Let's look at an example of several edits applied to the version of the Echo machine that plays Prisoner's Dilemma. Let action 0 be cooperate and action 1 be defect. Then, Echo becomes Tit-for-Tat.

Example 14.2 *Let's look at the results of starting with Echo (Tit-for-Tat in the Prisoner's Dilemma) and applying the following sequence of editing commands:* D_1, M_1, P_0, F_1, F_0, *or, if we issue the commands using the inputs and outputs of the Prisoner's Dilemma:* D_D, M_D, P_C, F_D. F_C. *The current state is denoted by a double circle on the state.*

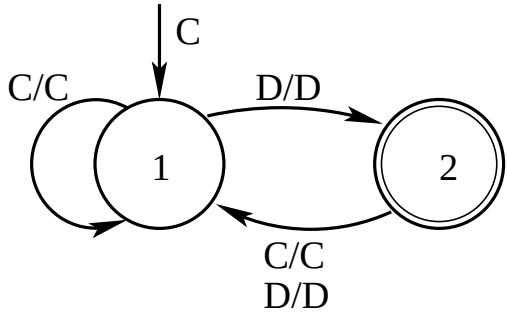
Tit-for-Tat is the starting point.



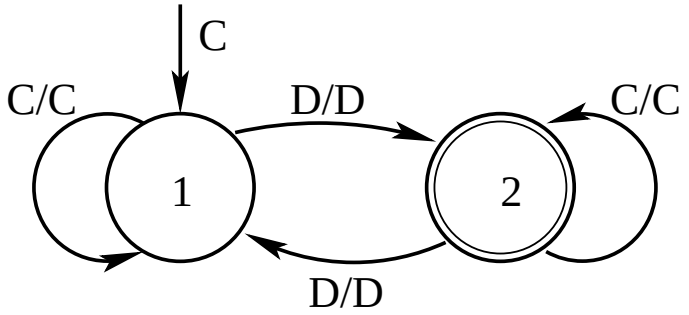
D_D *inserts a copy of 1 as the new destination of 1's D-transition.*



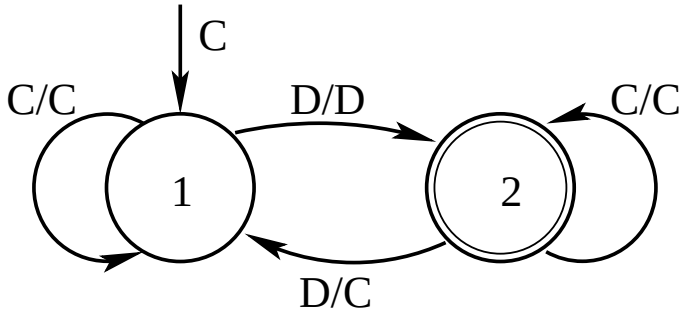
M_D *moves the active state to state 2.*



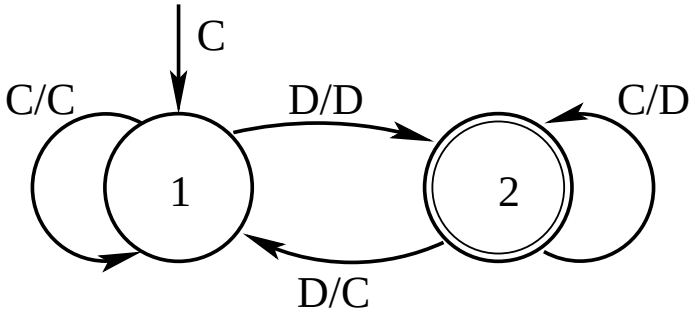
P_C *pins the C-transition from the current state to the current state.*



F_D increments the response on the D-transition from the current state.



F_C increments the response on the C-transition from the current state.



So, this sequence of editing commands transforms our starting automaton, Tit-for-Tat, into a version of Pavlov.

Let's start characterizing the behavior of the system. The following experiment examines the sizes of automata produced by genes of length 20.

Experiment 14.8 *Use an input/output alphabet of size 2. This gives us a total of 12 editing commands. Implement or obtain software for an automaton editor that builds an automaton from a sequence of editing commands. Generate 1,000,000 strings of 20 random editing commands and express them as automata. Compute the number of states in each of these automata and the fraction that have at least one state not connected to the initial state. Make a histogram of the lengths of the self-play strings of the automata. (The self-play string is defined in Chapter 6, page 149.)*

Generate 1,000,000 additional automata and collect the same numbers, but this time make the two commands, D_0 and D_1, three times as likely as the other commands. What effect does this have on the statistics collected?

One of the issues that must be dealt with when using this cellular encoding for finite state automata is that of state number. Review Experiment 6.4. The number of states used was pretty critical to performance in that experiment. In the next experiment, we will perform Experiment 6.4 again, attempting to find the "right" length for the encoding.

Experiment 14.9 *Modify the software from Experiment 6.4 to use the cellular encoding scheme described in this section. Use the string prediction fitness function alone and the lexical product of string prediction with self-driving length, with string prediction dominant. Use strings of 20, 40, 60, and 80 editing commands. Use two point crossover and one point mutation that replaces a randomly selected editing command with a new one selected at random. Attempt to match the reference string 111110 using 12 bits.*

Report mean and standard deviation of the number of generations to solution. In your write up, compare the difference between the plain and lexical fitness functions. What is the effect of changing the length of the strings of editing commands? Is the impact of the lexical fitness partner different for different lengths of strings?

In Experiment 14.8, we tried tinkering with the statistics governing the random generation of editing commands. The relative probability of choosing various commands can be optimized for any experiment. Which probability distributions are"good" depends on the choice of problem.

Experiment 14.10 *Perform Experiment 14.9 again with the D_n commands first twice as likely as the others and then three times as likely. Use whichever fitness function worked best in the previous experiment. Also, choose a set of probabilities of your own for all 12 editing commands, trying to get better performance than any of the 3 sets of probabilities tested before or in this experiment.*

The idea of placing a non-uniform distribution on the set of editing commands used in a cellular representation can be generalized a good deal. See Problem 14.37 for one such generalization. The effect of increasing the probability of the D_n commands is to increase the number of states produced relative to the length of the string of editing commands. There are other ways we could control this.

Experiment 14.11 *Perform Experiment 14.10 with the following technique for generating the initial population of strings of editing commands. Use only 40-character strings. Place exactly 7 D_n commands and 33 other commands in the edit strings in a random order. This will cause all the automata to have 8 states. Compare the impact of this initialization method with the results obtained in Experiment 14.10.*

At this point, we will leave the bit-grinding optimization tasks and return to the world of game theory. The basic Iterated Prisoner's Dilemma experiment was performed as Experiment 6.5. Let's revisit a version of this and compare the standard and cellular encodings.

Experiment 14.12 *Rebuild the software from Experiment 6.5 to optionally use cellular encodings. Also, write a tournament program that permits saved files of Prisoner's Dilemma players to play one another. Run the original software with 8-state automata and also run a cellular encoding with a gene length of 48 (yielding an average of 8 states). Perform 30 evolutionary runs for each encoding.*

Compare the resulting behaviors in the form of fitness tracks. Save the final populations as well. For each pair of populations, one evolved with standard encoding and the other evolved with cellular encoding, play each population against the other for 150 rounds in a between-population round robin tournament. Record which population obtained the highest total score. Did either encoding yield substantially superior competitors?

Prisoner's Dilemma has the property that there are no "best" strategies in round robin tournaments. There are pretty good strategies, however, and a population can stay pretty stable for a long time.

You may already be familiar with another game called Rock Paper Scissors. This game is also a simultaneous two-player game but, unlike Prisoner's Dilemma, there are three possible moves: rock (**R**), paper (**P**), and scissors (**S**). Two players choose moves at the same time. If they choose the same move, then the game is a tie. If the players choose different moves, then the victor is established by the following rules: rock smashes scissors; scissors cut paper; paper covers rock. We will turn these results into numbers by awarding 1 point each for a tie, 0 points for a loss, and 3 points for a victory. Table 14.2 enumerates the possible scoring configurations.

Rock Paper Scissors is a game with 3 possible moves, and, so, we will have 17 editing commands instead of the 12 we had with Prisoner's Dilemma. We've compared the standard and cellular encoding of finite state automata for playing Prisoner's Dilemma already. Let's repeat the experiment for Rock Paper Scissors.

Experiment 14.13 *Rebuild the software from Experiment 14.12 to play Rock Paper Scissors using the scoring system given above. Do agents encoded with the standard or cellular representation compete more effectively or is there little difference? Add the ability to compute the number of states in a finite state automaton that cannot be reached from the starting state and track the mean of this statistic in the population over the course of evolution. Does one representation manage to connect more of its states to the starting state? Is the answer to the preceding question different at the beginning and end of the evolutionary runs?*

Now, let's look at a strategy for Rock Paper Scissors that has a fairly good record for beating human beings.

Move		Score	
Player1	Player2	Player1	Player2
R	R	1	1
R	P	0	3
R	S	3	0
P	R	3	0
P	P	1	1
P	S	0	3
S	R	0	3
S	P	3	0
S	S	1	1

Table 14.2: Scoring for Rock Paper Scissors

Definition 14.3 *The strategy* **LOA** *(law-of-averages) for playing Rock Paper Scissors works as follows. If one move has been made most often by its opponent, then it makes the move that will beat that move. If there is a tie for move used most often, then LOA will make the move, rock, if the tie involves scissors, and the move, paper, otherwise.*

Experiment 14.14 *Rebuild the software from Experiment 14.13 to play Rock Paper Scissors against the player LOA. In other words, we are now optimizing finite state automata to beat LOA rather than co-evolving them to play one another. You must write or obtain from your instructor the code for LOA. Evolve both standard and cellular encodings against LOA playing 120 rounds. Do 30 runs each for 8- and 16-state finite state automata and cellular encodings of length 68 and 136. Which encoding works better? Do more states (or editing commands) help more?*

We've done several comparisons of the standard and cellular encodings of finite state automata. The most recent test the ability of the two encodings to adapt to a strategy that cannot be implemented on a finite state automaton (see Problem 14.31). The ability of a representation to adapt to a strategy written using technology unavailable to it is an interesting one and you can invent other non-finite-state methods of playing games, if you want to try other variations of Experiment 14.14.

One thing we have not done so far is to test two representations directly in a competitive environment. In the next two experiments, we will modify the tournament software used to assess the relative merit of strategies evolved with the standard and cellular encodings into a fitness function. This will permit a form of direct comparison of the two representations.

Experiment 14.15 *Write or obtain software for an evolutionary algorithm that operates on two distinct populations of finite state automata that encode Prisoner's Dilemma strategies. The first should use standard encoding and have 16 states. Use the variation operators from Experiment 6.5. The second population should use cellular encoding with editing strings of length 96, two point crossover, and two point mutation that replaces two editing commands with new ones in a given 96-command editing string.*

Evaluate fitness by having each member of one population play each member of the other population for 150 rounds of Iterated Prisoner's Dilemma. As in Experiment 6.5, pick parents from the top $\frac{2}{3}$ of the population by roulette selection and let them breed to replace the bottom $\frac{1}{3}$ of the population. Perform 100 evolutionary runs.

Record the mean fitness and standard deviation of fitness for both populations in a run separately. Record the number of generations in which the mean fitness of one population is ahead of the other. Report the total generations across all populations in which one population outscored the other.

The character of the game may have an impact on the comparison between representations. We have already demonstrated that Iterated Prisoner's Dilemma and Rock Paper Scissors have very different dynamic characters. Let's see if the last experiment changes much if we change the game.

Experiment 14.16 *Repeat Experiment 14.15 for Rock Paper Scissors. Compare and contrast.*

The material presented in this section opens so many doors that you will probably have thought of dozens of new projects and experiments while working through it. We leave the topic for now.

Problems

Problem 14.17 *Is there a single string of editing commands that produces a given automaton A?*

Problem 14.18 *Using the set of editing commands given in Table 14.1, find a derivation of the strategy Tit-for-Two-Tats. This strategy is defined in Chapter 6.*

Problem 14.19 *Using the set of editing commands given in Table 14.1, find a derivation of the strategy Ripoff. This strategy is defined in Chapter 6.*

Problem 14.20 *What is the expected number of states in an automaton created by a string of n editing commands, if all the commands are equally likely to be chosen and we are using a k-character input and output alphabet.*

Problem 14.21 *Reread Experiment 14.9. Find a minimum length string of editing commands to create an automaton that would receive maximum fitness in this experiment.*

Problem 14.22 *A* connection topology *for an FSA is a state transition diagram with the response values blank. Show that, assuming any version of a topology can be created with the set of editing commands given in Table 14.1, the responses can be filled in any way you want.*

Problem 14.23 *Is the representation used for polyominos in Section 14.1 complete? Prove your answer is correct. Hint: this isn't a difficult question.*

Problem 14.24 *A polyomino is* simply connected, *if it does not have an empty square surrounded on all sides by full squares. Give an example of a gene for a polyomino that is not simply connected. Then, write out a cellular encoding that can only create simply connected polyominos.*

Problem 14.25 *Is the set of editing commands given in Table 14.1 complete? Either prove it is or find an automata that cannot be made with the commands. You may find it helpful to do Problem 14.22 first.*

Problem 14.26 *The Echo strategy, used as the starting point for editing finite state automata, turns out to be Tit-for-Tat when used in the context of Prisoner's Dilemma. In Iterated Prisoner's Dilemma, Tit-for-Tat is a pretty good strategy. In Rock Paper Scissors, is Echo (effectively, rock first, and then repeat your opponent's last action) an effective strategy?*

Problem 14.27 *Prove that the population average score in a population playing Rock Paper Scissors with the scoring system given in this chapter is in the range,* $1 \leq average \leq 1.5$. *Prove that, if a population consists of a single strategy, then the population gets an average score of exactly 1.*

Problem 14.28 *Give a pair of strategies for Rock Paper Scissors that get an average score of 1.5, if they play one another an even number of times.*

Problem 14.29 *Is the population average score for a population equally divided between two strategies that are correct answers to Problem 14.28 completely predictable? If so, what is it? If not, explain why not.*

Problem 14.30 *Is it possible for a 60-member population playing 120 rounds of Rock Paper Scissors to achieve the upper bound of 1.5 on population average fitness? Explain.*

Problem 14.31 *Prove that the strategy LOA, given in Definition 14.3, cannot be implemented with a finite state automaton.*

Problem 14.32 *Is the Graduate School game (defined in Section 6.3) more like Prisoner's Dilemma or Rock Paper Scissors.*

Problem 14.33 *In the single shot Prisoner's Dilemma, there is a clear best strategy: defect. Does Rock Paper Scissors have this property? Prove your answer.*

Problem 14.34 Essay. *The claim is made on page 381 that the strategy LOA for playing Rock Paper Scissors does well against humans. Verify this fact by playing with a few friends. What sort of strategies does LOA do well against?*

Problem 14.35 Essay. *Either examining the experimental evidence from Experiment 14.14 or working by pure reason, answer the following question. Will the strategy LOA be one that performs well against finite state automata or will it perform poorly?*

Problem 14.36 Essay. *The number of states in a finite state automaton is not explicitly given in cellular encoding. Suppose you want a certain number of states. You could simply go back to the beginning of the string of edit commands and keep editing until you had as many states as desired. Your assignment: figure out what could go wrong. Will this method always generate as many states as you want? Will the type of automata be different than it would be if instead you used a very long string of edit commands and stopped when you had enough states?*

Problem 14.37 Essay. *Discuss the following scheme for improving performance in Experiments 14.9-14.10. Do a number of preliminary runs. Looking at the genes for FSAs that achieve maximal fitness, tabulate the empirical probability of seeing each command after each other command in these genes. Also, compute the probability of seeing each editing command as the first command in these successful genes. Now, generate random initial genes as follows. The first command is chosen according to the distribution of first commands you just computed. Generate the rest of the string by getting the next command from the empirical distribution of next commands you computed for the current command. Do you think this empirical knowledge reuse will help enough to make it worth the trouble? What is the cost? Can this scheme cause worse performance than generating initial populations at random?*

Problem 14.38 Essay. *A* most common strategy *is one that occupies the largest part of the population among those strategies present in a given population. If we look at the average time for one most common strategy to be displaced by another, we have a measure of the volatility of an evolving system. If you surveyed many populations, would you expect to see higher volatility in populations evolving to play Prisoner's Dilemma or Rock Paper Scissors?*

14.3 Cellular Encoding of Graphs

In this section, we venture into the realm of combinatorial graph theory to give a fairly general encoding for 3-connected cubic graphs.

Definition 14.4 *A graph is* **k-connected***, if there is no set of less than k edges that we could delete and thereby disconnect the graph.*

Definition 14.5 *A graph is* **cubic** *if each vertex is of degree 3.*

We will start with a very simple graph and use editing rules to build up more complex graphs. In some ways, the cellular encoding we will use for 3-connected cubic graphs is very similar to the one we used for finite state automata. The transition diagrams of finite state automata are directed graphs with regular out degree (always exactly k output arrows). In other ways, the encoding will be quite different; there will be two editing "agents" or bots at work, rather than a single current state.

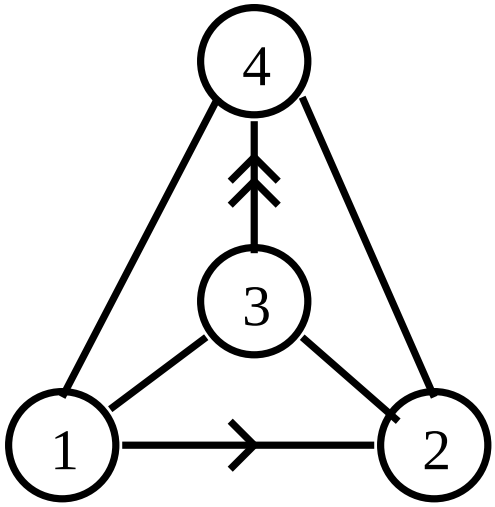


Figure 14.4: The initial configuration for the graph editing bots

The starting configuration for our cellular encoding for graphs is shown in Figure 14.4. The single and double arrows denote the edges that will be the focus of our editing commands. We will refer to these arrows as graph bots with the single arrow denoting the first graph bot and the double arrow denoting the second. During editing, the vertices will be numbered. There are two sorts of editing commands that will be used in the cellular encoding. The first group is used to move the the graph bots; the second will be used to add vertices and edges to the graph.

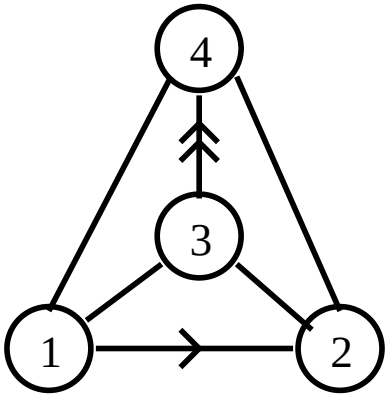
The movement commands will use the fact that the vertices are numbered. The commands, **R1** and **R2**, cause the first and second graph bots, respectively, to reverse their directions. These commands are spoken "reverse one" and "reverse two." The command, **AS1**, causes the first graph bot to advance past the vertex at which it is pointing so that that vertex is now at its tail. There are two ways to do this, since each vertex has degree 3. **AS1** causes the bot to point to the vertex with the smaller number of the two available. The command **AL1** also advances the first graph bot, but moves it toward the larger of the two available vertices. The commands, **AS2** and **AL2**, have the same effect as **AS1** and **AL1** for the second graph bot. These commands are spoken "advance small one," "advance large one," "advance small two," and "advance large two," respectively. The effect of the movement commands on the starting graph are shown in Figure 14.5. One important point: we never permit the graph bots to occupy the same edge. If a command causes the two graph bots to occupy the same edge, then ignore that command.

The commands that modify the graph are **I1** and **I2**, spoken "insertion type one" and "insertion type two." Both of these commands insert two new vertices into the middle of the edges with graph bots and join them with a new edge. The new vertices are given the next two available numbers with the smaller number given to the vertex inserted into the edge containing the first graph bot. The two insertion commands are depicted pictorially in Figure 14.6. **I1** differs from **I2** in the way the graph bots are placed after the insertion. **I1** reverses the direction of the second graph bot; **I2** reverses the direction of the first graph bot. In all cases, the bots are placed so that they are pointing away from the new vertices.

To prove that the 8 commands given are sufficient to make any 3-connected cubic graph requires graph theory beyond the scope of this text. In general, however, any cellular encoding requires the designer to deal with the issue of completeness or at least the issue "can the encoding I've dreamed up find the objects that solve my problem?" We next give a derivation of the cube using the set of editing commands just described.

Example 14.3 *The sequence of commands* **AL2**, **I2**, **AL1**, **AS1**, **I1** *yield the cube. Let's look at the commands one at a time.*

Start:



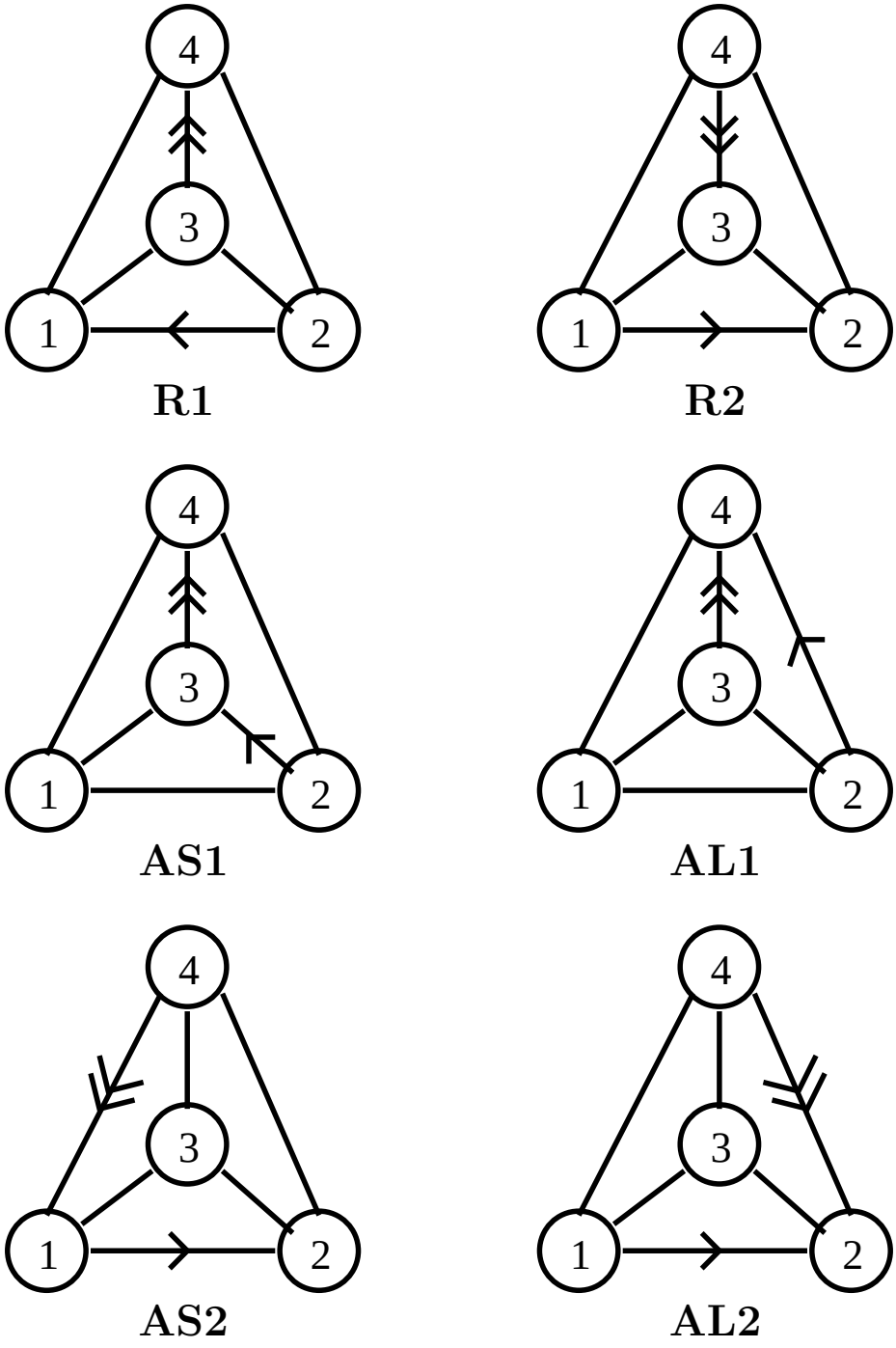


Figure 14.5: The result of applying each of the editing commands to the initial configuration

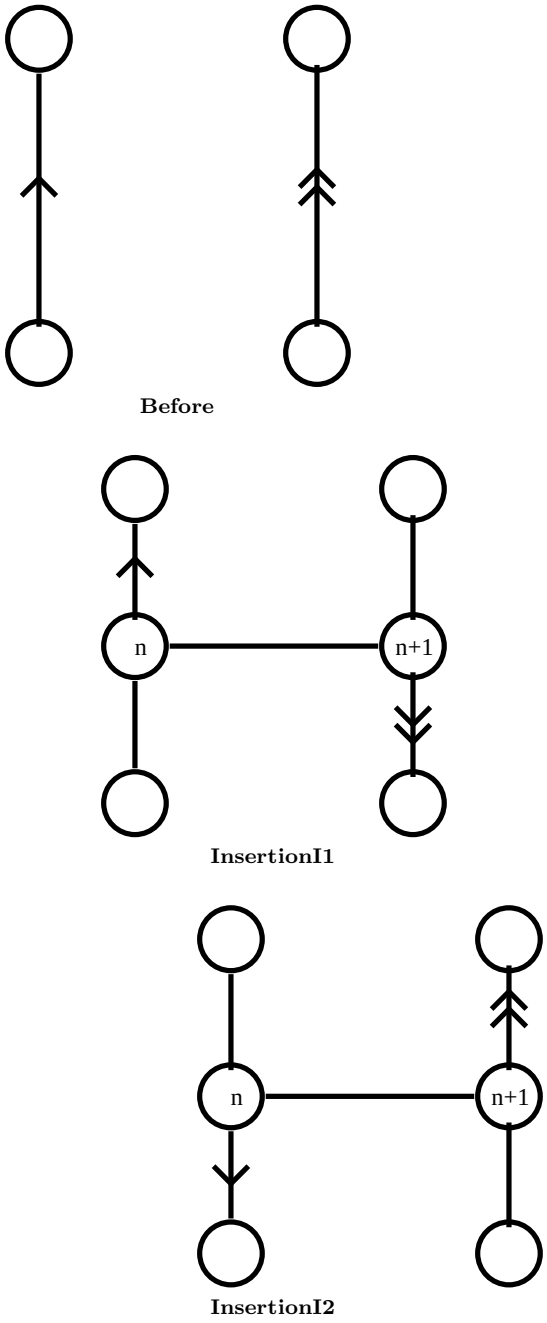
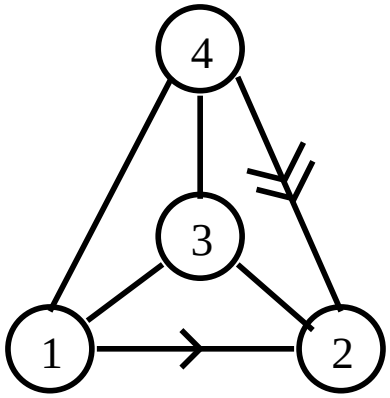
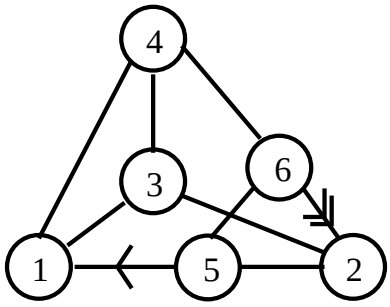


Figure 14.6: A generic positioning of the graph bots and the results of executing the two insertion commands (The commands differ only in their placement of the graph bots after the insertion.)

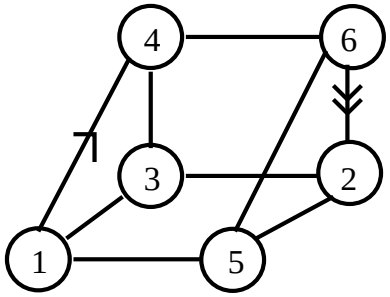
Apply **AL2:**



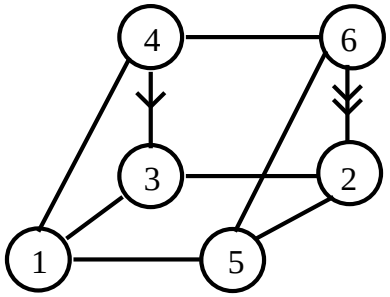
Apply **I2:**



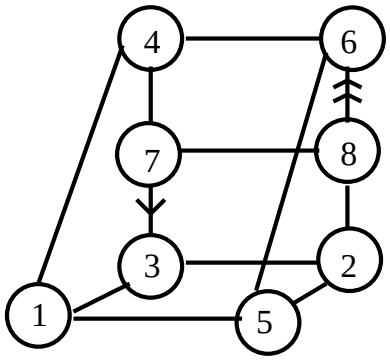
Apply **AL1:**



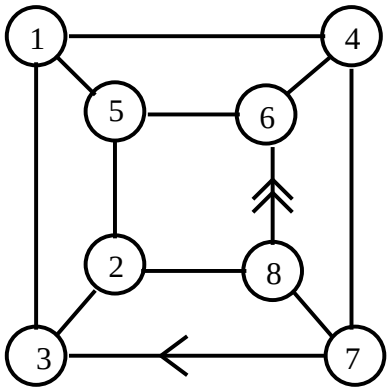
Apply **AS1:**



Apply **I1:**



The resulting graph is a somewhat bent version of the cube. Redrawn as a standard cube, we get:



This derivation leaves the graph bots on the graph at the end. When the graph is passed on to another routine, e.g., a fitness evaluation, the graph bots are discarded.

We've now got an encoding for graphs as a string of editing commands. We can use an evolutionary algorithm to evolve graphs by just dusting off a string evolver over an 8-character alphabet. There is still a very important piece missing, however: the fitness function.

What do people want in a graph, anyhow? Cast your mind back to the graphs used in Chapter 13. We used various cubic Petersen graphs which were quite symmetric and had fairly high diameter, and we used random cubic graphs, obtained with edge moves, that were not at all symmetric and had pretty low diameter, given their size. What about graphs with intermediate diameters? Our first task will be to search for these.

Instead of just using the diameter as a fitness function, we're going to break up the notion of diameter into smaller pieces with a few additional definitions. Read Section C.3 in Appendix C on distances in graphs.

Definition 14.6 *The* **eccentricity** *of a vertex in a connected graph is the largest distance between it and any other vertex in the graph. For a vertex v, this quantity is denoted $Ecc(v)$.*

The diameter of a graph is simply the maximum eccentricity of any of its vertices. To get a graph with an intermediate diameter, we will minimize the sum of the squared deviations of the eccentricities of all of the graph's vertices from the desired diameter. This will push the graph towards the desired diameter.

Definition 14.7 *The* **eccentricity deviation fitness function** *for eccentricity E for a graph G with vertex set $V(G)$ is defined to be*

$$ED_E(G) = \sum_{v \in V(G)} (E - Ecc(v))^2.$$

Notice that this fitness function is to be minimized.

When we were working with cellular encoding of finite state automata, we found that controlling the number of states in the automaton required a bit of care. Unlike the standard encoding, the number of states was not directly specified as a parameter of the experiment. It was, however, one more than the number of D_n commands. Since the insertion commands for graph editing insert two vertices, the number of vertices in a graph is four plus twice the number of I commands. If we generate genes at random, we will not have good control over the graph size.

In order to get graphs of a specific size, we will execute edit commands until the graph is the desired size. This means the genes need to be long enough to have enough insertion commands. On average, one command in four is an edit. We will test two methods of getting graphs that are big enough.

Experiment 14.17 *Implement or obtain software for a string evolver over the alphabet of graph editing commands defined in this section. Use genes of length 130 with two point crossover and three point mutation. When creating a graph from a string of editing commands, continue editing until either there are 256 vertices in the graph or you reach the end of the edit string. Use lexical fitness in which fitness is the number of vertices in the graph with ties broken by the function*

$$ED_{12}(G),$$

given in Definition 14.7. Evolve populations of 200 graphs using a steady state algorithm with size 7 single tournament selection.

Report the mean and deviation of both fitness functions and the best value of

$$ED_{12}(G)$$

from each run. Permit evolution to continue for 500 generations. Also, save the diameter of the most fit graph in each run. Report a histogram of the diameters of the most fit graphs in each run. How fast do the genes converge to size 256 graphs? Was the process efficient at minimizing $E_{12}(G)$?

This is a new way of using a lexical fitness function. Instead of putting the fitness of most interest as the dominant partner, Experiment 14.17 puts a detail that has to be gotten right as the dominant partner. This forces a sufficient number of insertion commands into the genes. Once this has happened, we can get down to the business of trying to match a mean eccentricity of 12. Now, let's try another approach to the size control problem.

Experiment 14.18 *Repeat Experiment 14.17 using a different method for managing the size of the graphs. Instead of lexical fitness with size of the graph dominant, use only the fitness function $ED_{12}(G)$. Use genes of length 60 cycling through until the graph is of sufficient size. Explicitly check each gene to make sure it has at least one insertion command and award it a fitness of zero if it does not. (This is unlikely to happen in the initial population but may arise under evolution.)*

Sampling from a region of eccentricity space that is difficult to reach with the explicit constructions and random algorithms given in Appendix C, elicits graphs that might be interesting to use in the kind of experiments given in Chapter 13. The reason for thinking such graphs might behave differently is that they have one parameter, average eccentricity, that is different.

Looking at the diameters of the various cubic graphs we used in Chapter 13, we also see that the large diameter cubic graphs were all generalized Petersen graphs and, hence, highly symmetric. The random graphs are all at the very low end of the diameter distribution. An evolved population of graphs created with our editing commands is unlikely to contain a highly symmetric graph. Let's see how it can do at sampling the extremes of the diameter distribution.

Experiment 14.19 *Modify the software from Experiment 14.18 to maximize the diameter of graphs. Use the diameter as the fitness function. Use genes of length 60. Since vertices are needed to build diameter, no lexical products will be needed to encourage the production of diameter. Run 10 populations for 5000 generations and save a best gene from generation 50, 100, 500, and 5000 in each run.*

Examine the genes and report the fraction of insertion commands in the best genes from each epoch. Also, save and graph the mean and variance of population fitness, the best fitness, the mean and variance of the vertex set sizes for the graphs, and the fraction of insertion commands in each generation.

It may be that the only imperative of evolution in the preceding experiment is to have all insertion commands. Let's perform a second experiment that speaks to this issue.

Experiment 14.20 *Repeat Experiment 14.19 with genes of length 30 and cycle through them twice. Compare with the results of Experiment 14.19.*

The last two experiments attempted to make high diameter graphs. Such graphs are "long" and may resemble sausages when drawn. We will now try to do the opposite. Since having few vertices always yields very low diameter, we will write a more complex fitness function that encourages many vertices and low diameter (compactness).

Definition 14.8 *For a graph G with vertex set $V(G)$, let*

$$CP(G) = \frac{|V(G)|}{\sum_{v \in V(G)} Ecc(v)}.$$

This function is called the **large compact graph** *function. It divides the number of vertices by the sum of their eccentricities. This function is to be maximized.*

Experiment 14.21 *Repeat Experiment 14.19 with the large compact graph function as the fitness function. Compare the resulting statistics and explain. Did the fitness function in fact encourage large compact graphs?*

So far, we have used the graph editing representation to sample the space of cubic graphs for rare diameters and eccentricities. The resulting graphs are amorphous and probably not of any great interest to graph theorists. They may have application to the kind of work done in Chapter 13. These problems were mostly intended to help us to understand and work with the graph editing system. At this point, we will go on to a much more difficult mathematical problem.

Definition 14.9 *The* **girth** *of a graph is the length of the shortest closed cycle in the graph. The* **girth at** v*, for a vertex* v *of a graph, is the length of shortest closed cycle of which that vertex is a member.*

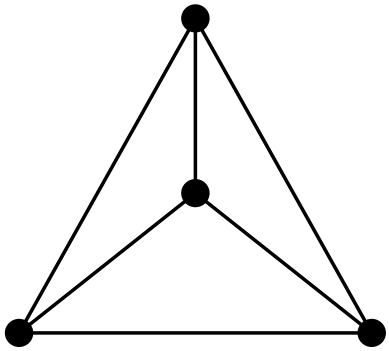
Look at the graphs used in Problem 14.42. These graphs have girth 4 and the girth at every vertex is 4. The Petersen graph $P_{5,2}$ has girth 5.

Definition 14.10 *A* **(3,n)-cage** *is a cubic graph with girth* n *and the smallest possible number of vertices. Examples of some of the known cages are given in Figure 14.7.*

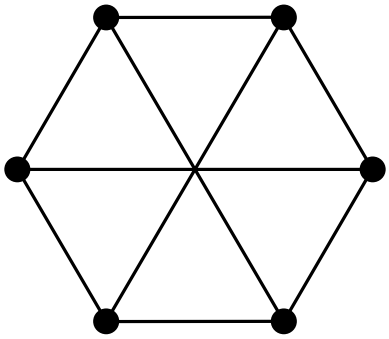
The $(3, n)$-cages are also called the cubic cages or even just the cages, because the notion was first defined for cubic graphs. The cubic cages form an interesting example of a phenomenon that is widespread in mathematics: small examples are not representative. The cages shown in Figure 14.7 are unique and symmetric. Unique means they are the only cubic graphs with their girth and size of vertex set. In this case, symmetric means that there is a way to permute the vertices that takes edges to edges such that any vertex can be taken to any other. The $(3, 7)$-cage is unique, but not symmetric. No other cage is symmetric in this sense. There are 18 different $(3, 9)$-cages, 3 different $(3, 10)$-cages, one known $(3, 11)$-cage, and a unique $(3, 12)$-cage. The $(3, 13)$-cage(s) is(are) not known.

Starting with beautiful, unique, symmetric graphs the family of cages rapidly degenerate into fairly ugly graphs that are not unique. The ugliness means that cages will, in the future, probably mostly be worked on by stochastic search algorithms (though success here is not guaranteed at all). The current lower bound on the size of a $(3, 13)$-cage is 202 vertices, a number that Brendan McKay and Wendy Myrvold computed by a cleverly written exhaustion of all possibilities. The current best know girth 13 cubic graph has 272 vertices and is given by an algebraic construction found by Norman Biggs.

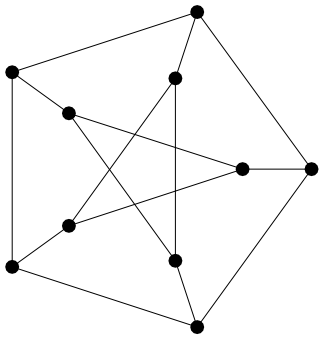
As before, the sticky wicket is writing a fitness function. The girth of a graph is the minimum of the girths at each of its vertices. Girth, however, would make a shockingly inefficient fitness function. At a given number of vertices, graphs of a smaller girth are far more common than those of a higher girth. At 30 vertices, it is possible to get girth 8, for



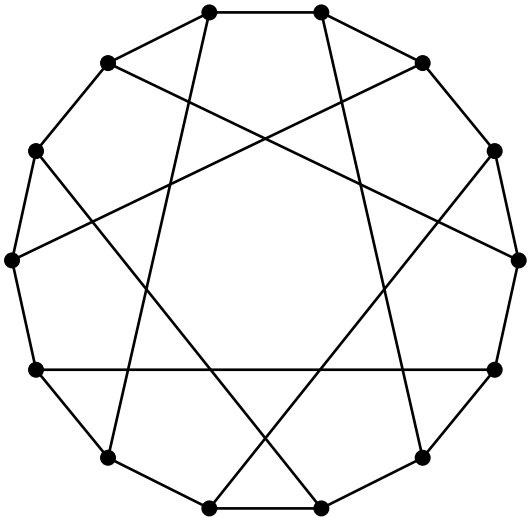
K_4



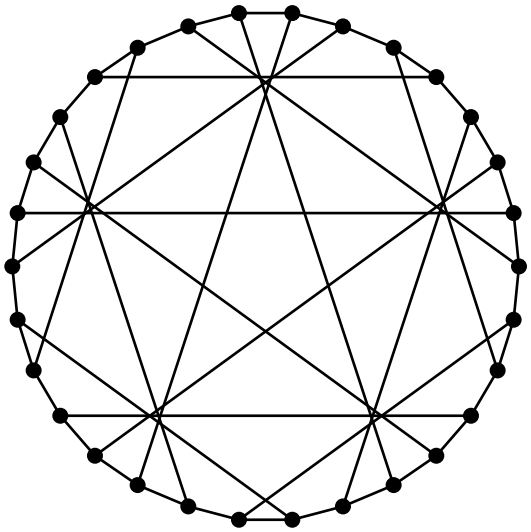
$K_{3,3}$



Petersen Graph



Heawood Graph



Tutte-Coxeter Graph

Figure 14.7: The $(3, n)$-cages for $n = 3, 4, 5, 6$, and 8

example, only by the discovery of a unique and highly symmetric graph. Before we decide on a fitness function, let's perform a sampling experiment to see how much trouble we are in.

Experiment 14.22 *Write or obtain software for an algorithm that starts with the initial graph configuration from Figure 14.4 and executes editing commands, sampled uniformly at random, until the graph has 30 vertices. Generate 100,000 graphs in this fashion and make a histogram of their mean girth at each vertex and their girth. Report, also, the ratio of each girth to the most common girth. Were any graphs of girth 8 found?*

Experiment 14.22 should have verified the assertion about rarity of high girth graphs and the problem with using girth directly as a fitness function. The mean girth at vertices is a much smoother statistic and will form the basis for herding graphs toward higher girth in the course of evolution.

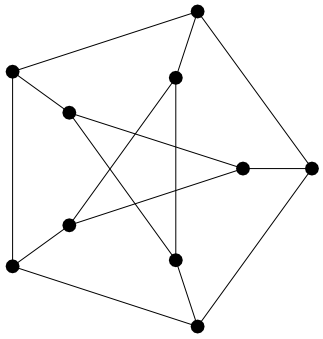
Experiment 14.23 *Write or obtain software for a steady-state evolutionary algorithm that operates on a population of k graph edit strings of length n generated uniformly at random. Use size 7 tournament selection, two point crossover, and three point mutation. Use the lexical product of girth and mean girth at each vertex, with girth being the dominant fitness function. Save and graph the mean and variance of both fitness functions and the maximum girth in each generation. Try all possible pairs of n and k, for $k = 100, 500, 1000$ and $n = 30, 60, 90$. For the length 30 strings, run through the strings once, twice, or three times. What girths do you obtain?*

In the past, we have tried various schemes for initializing populations to give evolution a boost. Combining Experiments 14.22 and 14.23 give us a means of doing this.

Experiment 14.24 *Repeat Experiment 14.23 initializing each run with the following procedure. Generate 100,000 genes. Test the fitness of each graph. As each graph is tested, save its gene only if it is in the top k of the graphs tested so far. Compare the results with those of Experiment 14.23.*

This section gives only a small taste of what could be done with cellular encodings of graphs. It treats one possible encoding for an interesting but limited class of graphs. There are many other problems possible. It would be possible, for example, to have a "current vertex" editor like the ones used for finite state automata in Section 14.2. The editing commands might involve insertion of whole new subgraphs in place of the current vertex. They could also include commands to swap edges as in Chapter 13 (page 353).

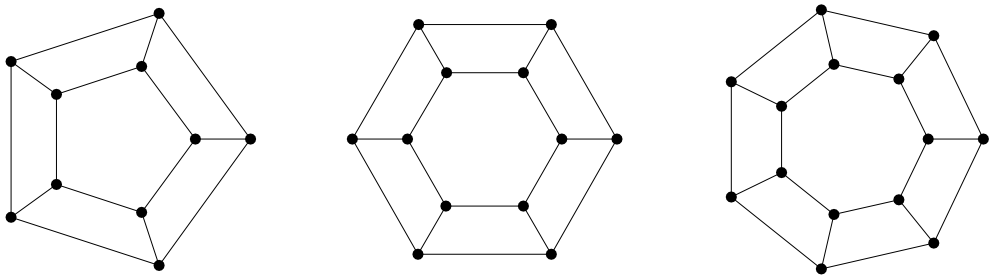
Problems



Problem 14.39 *Find a derivation, using the editing commands given in this section, for the Petersen graph. Use the standard starting point, as in Example 14.3.*

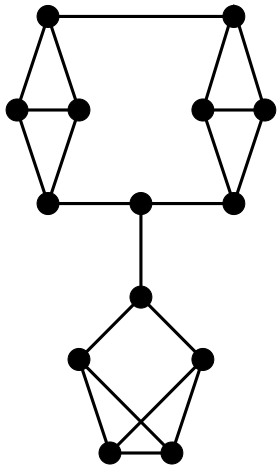
Problem 14.40 *Find a sequence of editing commands which transforms K_4 into $K_{3,3}$ into the Petersen graph into the Heawood graph. (Extra Credit: find a sequence of commands which transforms the Petersen graph into the Tutte-Coxeter graph.)*

Problem 14.41 *Give a minimal derivation for a graph with girth 4 using the starting graph and editing commands given in this section.*



Problem 14.42 *The cube, derived in Example 14.3, is also called the 4-prism. Above are the 5-prism, the 6-prism, and the 7-prism. Find a sequence of editing commands, including a segment repeated some number of times, that can create the n prism for any n. Say how many times the repeated fragment must be repeated to get the n prism.*

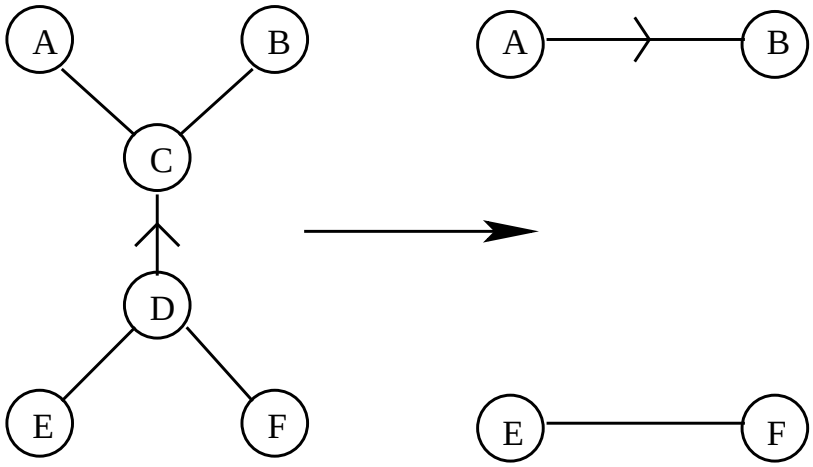
Problem 14.43 *Define the graph $IG(n)$ to be the result of applying the edit command* **I1** *to the initial configuration n times. Draw $IG(0)$, $IG(1)$, $IG(2)$ and $IG(3)$.*



Problem 14.44 *Make a copy of the graph above and label each vertex with its eccentricity.*

Problem 14.45 *Make a copy of the graph above and label each vertex with the girth at that vertex.*

Problem 14.46 *Prove that the set of editing commands for cubic graphs given in this section always produces a 3-connected graph. Do this by showing the commands cannot produce a graph that can be disconnected by deleting one edge or by deleting any two edges.*



Problem 14.47 *Suppose that we add the above deletion command, D_1, to our editing commands. Assume that $A < B$ (so as to make the direction of the arrow unambiguous). Also assume, if you want, that there is a corresponding command, D_2, involving the second graph bot. Prove that the language can now create disconnected graphs and graphs that are 1-connected or 2-connected but not 3-connected.*

Problem 14.48 *What restrictions do we need to place on the use of the deletion command(s) defined in Problem 14.47, if we want to avoid graphs with edges from a vertex to itself or multiple edges?*

Problem 14.49 *Reread Experiment 14.18. Compute the probability of any genes appearing in the initial population that have no insertion commands.*

Problem 14.50 *Design a short sequence of editing commands that, if repeated, will create a large diameter graph. Estimate or (better yet) compute exactly the ratio of the diameter of the graph to the number of repetitions of your sequence of editing rules.*

Problem 14.51 *Prove that, if we delete one of the insertion commands from the language, we can still make all of the same graphs.*

Problem 14.52 *Write or obtain software for expressing graph edit genes. Take 100,000 samples obtained by running through random 120-character genes until a graph with 100 vertices is constructed. Make a histogram of the diameter and mean eccentricity of the graphs.*

Problem 14.53 *The edge swap operation used to generate random regular graphs in Chapter 13 is described in Appendix C (on page 491in the definition of random regular graph). What restrictions would have to be placed on the positioning of the graph bots to permit adding a command that performed such an edge swap on the edges where the bots are? Assume that the new edges connect the former heads of the graph bots and the former tails of the graph bots, leaving the heads of the graph bots pointing toward the same vertex.*

Problem 14.54 *Give and defend a different representation for evolving graphs. It may be cellular or direct. If possible, make it more general than the representation given in this section.*

Problem 14.55 Essay. *In Chapter 10, we developed a GP-automata representation for discrete robots performing the Tartarus task. Suppose that we were to put GP-automata in charge of our graph bots. List and defend a set of input terminals for the deciders that would be good for the task used in Experiment 14.17.*

Problem 14.56 Essay. *In Chapter 10, we developed a GP-automata representation for discrete robots performing the Tartarus task. Suppose that we were to put GP-automata in charge of our graph bots. List and defend a set of input terminals for the deciders that would be good for the task used in Experiment 14.23.*

Problem 14.57 Essay. *In Chapter 12, we developed ISAc lists for discrete robots performing a variety of discrete robotics tasks. Suppose that we were to put ISAc lists in charge of our graph bots. List and defend a set of data vector entries that would be good for the task used in Experiment 14.17.*

Problem 14.58 Essay. *In Chapter 12, we developed ISAc lists for discrete robots performing a variety of discrete robotics tasks. Suppose that we were to put ISAc lists in charge of our graph bots. List and defend a set of data vector entries that would be good for the task used in Experiment 14.23.*

Problem 14.59 Essay. *Is the diameter or the average eccentricity a better measure of how dispersed or spread out a graph is, assuming we are using the graph to control mating as in Chapter 13.*

Problem 14.60 Essay. *Generalize the editing method given in this section for cubic graphs to regular graphs of degree 4. Give details.*

14.4 Context Free Grammar Genetic Programming

One of the problems with genetic programming is the disruptiveness of subtree crossover. Another problem is controlling the size of the parse trees created. In this section, we will use a cellular representation to take a shot at both problems by changing our representation for parse trees. We will do this by creating *context free grammars* that control the growth of parse trees. The grammar will form a cellular encoding for the parse trees. Grammar rules will be used the way editing commands were used for finite state automata and graphs in earlier sections of this chapter.

In addition to neatening crossover and controlling size, using grammatical representations for specifying parse trees solves the data typing problem. Using the old method, we could only use multiple types of data in a parse tree by encoding them in the data type of the tree. In Chapter 9, for example, the ITE operation took 3 real number arguments, but the first was used as if it were a Boolean argument by equating negative with false. Since grammars can restrict what arguments are passed to operations, they can do automatic data type checking. There will be no need to write complex verification or repair operators that verify subtree crossover obeys data typing rules.

The use of grammars will also permit us to restrict the class of parse trees examined by embedding expert knowledge into the grammar. For example, the grammar can exclude redundant combinations of operators, like store a number in a memory and then store the exact same number in the same memory again. Both data typing and expert knowledge are embedded in the algorithm at *design* time rather than run time. So, beyond the overhead

for managing the the context free grammar system, there is close to zero runtime cost for them. Cool, huh?

A *context free grammar* contains: a collection of *non-terminal symbols*, a collection of *terminal symbols*, and a set of *production rules*. (This is a different use of the word "terminal" than we used in previous chapters on genetic programming. To avoid confusion, in this chapter, we will refer to the terminals of parse trees as "leaves.") In a context free grammar, a non-terminal symbol is one that can still be modified; a terminal symbol is one that the grammar cannot modify again. A sequence of production rules is called a *production*. A production starts with a distinguished non-terminal symbol, the *starting non-terminal*. It then applies a series of production rules. A production rule replaces a single non-terminal symbol with a finite string of terminal and non-terminal symbols. By the end of a production, all non-terminal symbols are resolved. Let's do an example.

Example 14.4 *Recall the PORS language from Chapter 8. Here is a context free grammar for the PORS language.*

Non-terminals: **S**
Starting non-terminal: **S**
Terminals: **+, 1, Rcl, Sto**

Production Rules:
Rule 1: **S → (+ S S)**
Rule 2: **S → (Sto S)**
Rule 3: **S → Rcl**
Rule 4: **S → 1**

Starting with a single S let's see what parse tree we get if we apply the sequence of rules 121443.

Start: **S**
Apply 1: **(+ S S)**
Apply 2: **(+ (Sto S) S)**
Apply 1: **(+ (Sto (+ S S)) S)**
Apply 4: **(+ (Sto (+ 1 S)) S)**
Apply 4: **(+ (Sto (+ 1 1)) S)**
Apply 3: **(+ (Sto (+ 1 1)) Rcl)**

The result is a correct solution to the Efficient Node Use Problem for 6 nodes.

There are two things that you will have noticed in Example 14.4. First, there are sometimes multiple non-terminals to which a given production rule could have been applied. When applying productions in a cellular encoding, we must give a rule for which non-terminal to use when multiple non-terminals are available. In the example, we used the leftmost available non-terminal. Second, there is the problem of unresolved non-terminals. If we are generating random non-terminals, it is not hard to imagine getting to the end of a list of production rules before resolving them all. In order to use a string of context free grammar products as a cellular encoding, we must deal with this issue.

Definition 14.11 *The* **cauterization rules** *are a set of rules, one for each non-terminal in a context free grammar, that replace a non-terminal with a string of terminal symbols. These rules are used to finish a production when an evolved list of production rules are used and leave non-terminals behind.*

A third issue that may be less obvious is that of rules that cannot be applied. If there is a production rule that cannot be applied, e.g., for want of an appropriate non-terminal symbol, then simply skip the rule.

We are now ready to formally state how to use context free grammars as a cellular representation for parse trees.

Definition 14.12 *A* **cellular representation for parse trees** *is a context free grammar together with a set of cauterization rules and rules for choosing which non-terminal will be chosen when multiple non-terminals are available. The representation is a string of production rules which the evolutionary algorithm operates on as a standard string gene.*

Let's perform some experiments.

Experiment 14.25 *Build or obtain software for a steady state evolutionary algorithm for the PORS n-node Efficient Node Use Problem using a context free grammar cellular encoding for the parse trees. Use strings of* $2n$ *production rules from the 4 rule grammar given in Example 14.4. The creatures being evolved are thus strings over the alphabet* $\{1, 2, 3, 4\}$.

Let the algorithm be steady state and use tournament selection of size 11 on a population of 400 productions. Use two point crossover and single point mutation. When executing a production, do not execute production rules 1 and 2 if they drive the total number of nodes above n (count all symbols, **+, 1, Rcl, Sto,** *and* **S**, *as nodes) If the tree still has non-terminal symbols after the entire string of production rules has been traversed, use rule 4 as the cauterization rule. When multiple non-terminals are available, use the leftmost one.*

Perform 100 runs recording mean and standard deviation of fitness as well as time-to-solution for 30 runs for $n = 12, 13,$ *and* 14. *Cut off runs that take more than 1,000,000 mating events to finish. Also, save the final populations of productions in each run for later use.*

Experiment 14.25 is our first try at context free grammar genetic programming. We used the simplest non-trivial genetic programming problem we have studied. Let's check the effect of tinkering with a few of the parameters of the system. Review the definition of nonaligned crossover, Definition 7.21.

Experiment 14.26 *Repeat Experiment 14.25 with 3 variations. In the first, expand the rightmost non-terminal, rather than the leftmost. In the second, replace one point mutation with two point mutation. In the third, make* $\frac{1}{4}$ *of all crossover nonaligned. Document the impact of each of these variations.*

In the introduction to this section, we claimed that we can use the grammar to cheaply embed expert knowledge into our system. Let's give an example of this process. The tree, **(Sto (Sto T))** where **T** is a tree, wastes a node. Let's build a grammar that prevents this waste.

Definition 14.13 No Sto-Sto grammar

Non-terminals: **S,T**
Starting Non-terminal: **S**
Terminals: **+, 1, Rcl, Sto**

Production Rules:
Rule 1: **S** → **(+ S S)**
Rule 2: **S** → **(Sto T)**
Rule 3: **S** → **Rcl**
Rule 4: **S** → **1**
Rule 5: **T** → **(+ S S)**

Using the No Sto-Sto grammar will be a little trickier, because the use of a **T** forces 3 nodes.

Experiment 14.27 *For the No Sto-Sto grammar, perform the variation of Experiment 14.25 that worked best. Do not let the number of symbols other than* **T** *plus 3 times the number of* **T** *symbols exceed the number of nodes permitted. For* **T***, use the cauterization rule* **T**→**(+ 1 1)**. *Compare your results with the other PORS Efficient Node Use experiments you have performed.*

Let's see how well population seeding works to generalize the result. It is time to use the productions we saved in Experiment 14.25.

Experiment 14.28 *Repeat Experiment 14.25 using the saved populations from Experiment 14.25 as the starting populations. Instead of* $n = 12, 13,$ *and* 14*, do runs for* $n = 15, 16,$ *and* 17*. Run 9 experiments, using each of the 3 types of populations saved (for* $n = 12, 13,$ *and* 14*) to initialize the runs.*

Before you perform the runs, predict which initialization will help the most and least with each kind of run ($n = 15, 16,$ *and* 17*). Predict whether random initialization would be superior for each of the 9 sets of runs. Compare your predictions with your experimental results and explain the reasoning that led you to make those predictions.*

This embedding of expert knowledge and evolved knowledge can be carried farther but additional development of killer PORS grammars and initializing populations are left for the Problems.

Let's take a look at the effect of a kind of rule analogous to biological *introns*. An intron is a sequence of genetic code that does not produce protein. In spite of "doing nothing," introns can effect (and in fact enable) crossover.

Experiment 14.29 *Repeat Experiment 14.26 using the best variation, but with a fifth production rule that does nothing. Perform 2 sets of runs which include the fifth rule. Make the strings of production rules 25% longer.*

We will now shift to a new problem: a maximum problem with two operations and a single numerical constant.

Definition 14.14 *A* **maximum problem** *is one in which the computer is asked to produce the largest possible result with a fixed set of operations and constants subject to some resource limitation.*

The PORS Efficient Node Use Problem is a type of maximum problem in which the resource limitation was total nodes. Limiting parse trees by their total nodes is not traditional in genetic programming as it was originally defined by John Koza and John Rice. Instead, parse trees are typically depth restricted. The depth of the tree from the root node is limited. The following maximum problem is a standard one for depth-limited genetic programming. We will start by building a standard depth-limited genetic programming system.

Experiment 14.30 *The* PTH maximum problem *uses the operations,* $+$ *and* $\times$*, and the numerical constant, one-half (***0.5***). As with the PORS Efficient Node Use Problem, the fitness of a parse tree is the result of evaluating it, with the value to be maximized. Create or obtain software for a steady state evolutionary algorithm that operates on PTH parse trees of maximum depth* k*, with the root node considered to be depth zero. Use size 7 single tournament selection. During reproduction, use subtree crossover 50% of the time. When subtree crossover creates a tree that exceeds the depth limit, prune it by deleting nodes that*

are too deep and transforming those nodes at depth k to leaves. For each tree, use a mutation operator that selects an internal node of the tree uniformly at random (if it has one) and changes its operation type. Run 50 populations until they achieve the maximum possible value for $k = 4$ (16) and for $k = 5$ (256). Cut a given run off if it has not achieved the maximum possible value in 1,000,000 mating events.

With a standard baseline experiment in place, we can now try some experiments for the PTH problem with context free grammars. The following experiments demonstrate the way knowledge can be embedded in grammars.

Definition 14.15 Basic PTH Grammar

Non-terminals: **S**
Starting Non-terminal: **S**
Terminals: **+, *, 0.5**

Production Rules:
Rule 1: **S** → **(* S S)**
Rule 2: **S** → **(+ S S)**
Rule 3: **S** → **0.5**

Experiment 14.31 *Repeat Experiment 14.30 with a context free grammar encoding using the basic PTH grammar. Expand the leftmost non-terminal first and use the obvious cauterization rule: rule 3 from the grammar. Do not execute any production that will make the tree violate its depth limit. Use two point crossover and two point mutation on your strings of production rules. Use strings of length 40 for $k = 4$ and strings of length 80 for $k = 5$. In addition to reporting the same statistics as those from Experiment 14.30 and comparing to those results, examine the genes in the final population. Explain why the system that exhibited superior performance did so.*

Let's see if we can get better results using a more effective grammar.

Definition 14.16 Second PTH Grammar

Non-terminals: **S,T**
Starting Non-terminal: **S**
Terminals: **+, *, 0.5**

Production Rules:
Rule 1: **S** → **(* S S)**

Rule 2: **S** → **(+ T T)**
Rule 3: **T** → **(+ T T)**
Rule 4: **T** → **0.5**

Experiment 14.32 *Repeat Experiment 14.31 with the second PTH grammar. Report the same statistics and compare. What was the impact of the new grammar? Cauterize all non-terminals remaining at the end of a production to* **0.5**.

It is possible to build even more special knowledge into the grammar.

Definition 14.17 Third PTH Grammar

Non-terminals: **S, T, U**
Starting Non-terminal: **S**
Terminals: **+, *, 0.5**

Production Rules:
Rule 1: **S** → **(* S S)**
Rule 2: **S** → **(+ T T)**
Rule 2: **T** → **(+ U U)**
Rule 3: **U** → **0.5**

Experiment 14.33 *Repeat Experiment 14.32 with the third PTH grammar. Report the same statistics and compare. What was the impact of the new grammar? Cauterize all non-terminals remaining at the end of a production to* **0.5**.

The three grammars for the PTH problem contain knowledge about the solutions to those problems. This is a less-than-subtle demonstration of how to cook a problem to come out the way you want. We now turn to a grammar for Boolean parse trees and look to see if we can extract generalizable knowledge from the system.

Definition 14.18 Boolean Parse Tree Grammar

Non-terminals: **S**
Starting Non-terminal: **S**
Terminals: **AND, OR, NAND, NOR, NOT, T, F,** $X_i(i = 1 \ldots n)$

Production Rules:
Rule 1: **S** → **(AND S S)**

Rule 2: **S** → **(OR S S)**
Rule 3: **S** → **(NAND S S)**
Rule 4: **S** → **(NOR S S)**
Rule 5: **S** → **(NOT S S)**
Rule 6: **S** → **T**
Rule 7: **S** → **F**
Rule 8: **S** → X_1
...
*Rule 7+n:***S** → X_n

The Boolean parse tree grammar works on n-input variables and so has a number of rules that vary with n. First, let's repeat a familiar experiment and see how well the system performs.

Experiment 14.34 *Create or obtain software for a steady state evolutionary algorithm that uses a context free grammar genetic programming representation for Boolean parse trees based on the Boolean parse tree grammar given above. Use a population of 400 productions of length 15 for the 2-parity problem, with the parity being the number of true inputs* (*mod* 2)*. Notice that your productions will be over a 9-letter alphabet. Cauterize by changing the first non-terminal to* X_1*, the second to* X_2*, and so on, cyclically. Fitness is the number of correct predictions of the parity of two binary variables for all possible combinations of Boolean values those variables can have. Use size 7 single tournament selection. Record time-to-solution for 100 runs and save the final population of productions from each run.*

The next step is to see if productions that solve the 2-parity problem can help solve higher order parity problems.

Experiment 14.35 *Repeat Experiment 14.35 for the 3-parity problem with length 30 productions over the Boolean parse tree grammar. Perform one set of runs with random initialization and a second set in which you replace a random substring of length 15 in each initial string with one of the saved strings from Experiment 14.34. Load all the strings from all the final populations and select at random among the entire set of strings while initializing. Report times-to-solution and document the impact of the non-standard initialization method.*

We now want to move on to demonstrate explicit data typing with context free grammar genetic programming. In the grammars used so far, there is something like data typing, e.g., look at the roles of **U** and **T** in the third PTH grammar. When we used GP-automata for the Tartarus problem, the deciders were created with genetic programming. The decider's job was to reduce a large set of possible patterns to a single bit (the parity of an integer) that could drive transitions of the finite state portion of the GP-automata. What we will do next is create a grammar for the deciders with the types **B** (Boolean) and **E** (sensor expression).

Definition 14.19 Tartarus Decider Grammar

Non-terminals: **S, B, E**
Starting Non-terminal: **S**
Terminals: **UM,UR,MR,LR,LM,LL,LM,UL,==,!=,0,1,2**

Production Rules:
Rule 1: **S** → **(AND B B)**
Rule 2: **S** → **(OR B B)**
Rule 3: **S** → **(NAND B B)**
Rule 4: **S** → **(NOR B B)**
Rule 5: **S** → **(NOT B B)**
Rule 6: **B** → **T**
Rule 7: **B** → **F**
Rule 8: **B** → **(== E E)**
Rule 9: **B** → **(!= E E)**
Rule 10: **E** → **UM**
Rule 11: **E** → **UR**
...
Rule 17: **E** → **UL**
Rule 18: **E** → **0**
Rule 19: **E** → **1**
Rule 20: **E** → **2**

Experiment 14.36 *Rebuild Experiment 10.18 to use a cellular parse tree encoding with the above grammar for its deciders. Change "if even" and "if odd" to "if false" and "if true." For cauterization rules, use* **S→(== UM 1)**, **B→ T**, *and* **E→***UM. Use a string of 20 production rules for each decider. Compare the results with those obtained in Experiment 10.18.*

Problems

Problem 14.61 *Give a string of context free grammar productions to yield an optimal tree for the Efficient Node Use Problem for all n given in Figure 8.5.*

Problem 14.62 *The No Sto-Sto grammar encodes a fact we know about the PORS Efficient Node Use Problem: a store following a store is a bad idea. Write a context free grammar that encodes trees for which all leaves of the parse tree executed before the first store are 1s and all leaves after the first store executed are recalls.*

Problem 14.63 *Using the four-rule grammar for PORS given in Example 14.4, do the following.*

(i) Express the production **S=1212121443333**.

(ii) Express the production **T=1212144133133**.

(iii) Perform two point crossover on the productions after the third and before the ninth character, coloring the production rules to track their origin in **S** and **T**. Express the resulting strings, coloring the nodes generated by productions from **S** and **T**.

Problem 14.64 *Give the exact value of the correct solution to the PTH problem with depth k trees. Prove your answer is correct.*

Problem 14.65 *Suppose that, instead of limiting by depth, we limited PTH by total nodes. First, show that the number of nodes in the tree is odd. Next, show that, if $f(n)$ is the maximum value obtainable with n nodes, then*

$$f(n+2) = Max(f(n) + 0.5, f(k) * f(m)),$$

where m and k are odd, and $n + 1 = m + k$. Using this fact, compute the value of $f(n)$ for all odd $1 \leq n \leq 25$.

Problem 14.66 *Explicitly state in English the knowledge about solutions to the PTH problem embedded in the second and the third PTH grammars given in this section.*

Problem 14.67 *Is it possible to write a grammar such that, if all the non-terminals are resolved, it must give an optimal solution to the PTH problem?*

Problem 14.68 Essay. *One problem we have in using genetic programming with real-valued functions is that of incorporating real constants. In the standard (parse tree) representation, we use ephemeral constants: constants generated on the spot. Describe a scheme for incorporating real constants in one or more production rules to use context free grammars with real-valued genetic programming.*

Problem 14.69 *Suppose we are using context free grammar genetic programming and that we have a mutation that sorts the production rules in the gene into increasing order. If this mutation were mixed in with the standard one at some relatively low rate, would it help?*

Problem 14.70 *Would you expect nonaligned crossover (see Definition 7.21) to help or hinder, if used at a moderate to low rate in the PTH problem?*

Problem 14.71 *Could one profitably store the leaves in parse trees for the PTH problem as nil pointers?*

Problem 14.72 *Suppose, in Experiment 14.25, we permit only 1s and 2s in the production rules and simply allow the cauterization rule to fill in the leaves. What change would this make in the experiment?*

Problem 14.73 *Suppose we added a new terminal,* **0.4***, and a 4th rule,* **S**→*0.4, to the basic grammar for the PTH problem. What would the maximum value for depth 4 and 5 now be? Would the problem become harder or easier to solve?*

Problem 14.74 *Give a cellular representation for parse trees with terminals,* **+, *, x, /,** *e, where e is a randomly generated ephemeral constant that yields rational functions. (Recall that a rational function is a ratio of two polynomials.)*

Problem 14.75 Essay. *Looking at the various grammars used for the PTH problem, address the following question. Do the more later grammars make the search space larger or smaller? Defend your view carefully.*

Problem 14.76 Essay. *Design and defend a better grammar for deciders for Tartarus than the one given in Definition 14.19.*