Chapter 12

ISAc List: Alternative Genetic Programming

In this chapter, we will look at another method of producing programs by evolution. This structure is far simpler than the genetic programming techniques we studied in previous chapters.

12.1 ISAc Lists: Basic Definitions

The structure we will now study, the If-Skip-Action list (ISAc list), was invented by Mark Joenks [1] during a fit of boredom in a machine-language programming class. Joenks observed that machine language instructions come in a linear order, followed during execution, except when interrupted by jumps. It struck him that if we wrote a very simple machine language that tested conditions and, based on them, either specified an action or made a jump, then we would have an evolvable programming language. An example of an ISAc list is given in Figure 12.1.

An *ISAc list* is a vector, or array, of ISAc nodes. An *ISAC node* is a quadruple (a, b, act, jmp) where a and b are indices into a data vector, act is an action that the ISAc node may take, and jmp is a specification of where to jump, if the action happens to be a jump action. The *data vector* holds inputs, scratch variables, and constants. In other words, everything we might put into the terminals of a genetic programming setup. An ISAc list comes equipped with a fixed boolean test used by every node. Execution in an ISAc list is controlled by the *instruction pointer*.

The operation of an ISAc list is as follows. We start with the instruction pointer at the beginning of the ISAc list, indexing the zeroth node. Using the entries a, b, act, $jump$ of that node, we look up data item a and data item b in the data vector and apply the boolean

Node	a	b	act	jmp	comment
0	3	2	2	6	if $1-0>0$ set register to zero
1	0	2	1	2	If $x-0>0$ jump to 3
2	3	2	1	5	If $1-0>0$ jump to 6
3	3	2	3	9	If $1-0>0$ increment register
4	3	2	4	4	If $1-0>0$ decrement x
5	3	2	1	0	If $1-0>0$ jump to 1
6	1	2	1	7	If $y-0>0$ jump to 8
7	3	2	1	10	If $1-0>0$ jump to 11
8	3	2	3	7	If $1-0>0$ increment register
9	3	2	5	2	If $1-0>0$ decrement y
10	3	2	1	5	If $1-0>0$ jump to six
11	3	2	0	4	If $1-0>0$ NOP

Figure 12.1: An example of an ISAc list that operates on 2 variables x, y and a scratch register (The data vector v is of the form [x,y,0,1]. The ISAc actions are 0-NOP, 1-jump, 2-zero register, 3-increment register, 4-decrement x, and 5-decrement y. The boolean test for this ISAC list is: if $(v[a]-v[b]>0)$, i.e., is item "a" larger than item "b"?)

test to them. If the test is true, we perform the action in the *act* field of the node, otherwise we do nothing. If that action is "jump," we load the contents of the *jmp* field into the instruction pointer. We then increment the instruction pointer. Pseudo-code for the basic ISAc list execution loop is shown in Figure 12.2.

There are 3 types of actions used in the *act* field of an ISAc node. The first is the *NOP* action, which does nothing. The inclusion of the NOP action is inspired by experience in machine language programming. An ISAc list that has been evolving for a while will have its performance tied to the pattern of jumps it has chosen. If we insert new instructions, the target addresses of many of the jumps change. We could tie our "insert an instruction" mutation operator to a "renumber all the jumps" routine, but this is computationally complex. Instead, we have a "do nothing" instruction that serves as a placeholder. Instructions can be inserted by mutating a NOP instruction, and they can be deleted by mutating into a NOP instruction *without* the annoyance of renumbering everything.

The second type of action used in an ISAc list is the *jump* instruction. For those of you who have been brainwashed by the structured programming police, the jump instructions are *goto* instructions. Any kind of control structure, "for", "repeat-until", "do-while" is really an "if(condition)then goto-label" structure, carefully hidden from the delicate eyes of the software engineer by his compiler or code-generator. In a high level language, this "goto-hiding" does aid in producing sensible, easy-to-read code. ISAc lists are low level programming and are rich in jump instructions. These instructions simply load a new value

```
IP← 0                                          //Set Instruction Pointer to 0.
LoadDataVector(v);                             //Put initial values in data vector.
Repeat                                         //ISAc evaluation loop
   With ISAc[IP] do                            //with the current ISAc node,
   If v[a] − v[b] > 0 then PerformAction(act); //Conditionally perform action
   UpdateDataVector(v);                        //Revise the data vector
   IP ← IP + 1                                 //Increment instruction pointer
Until Done;
```

Figure 12.2: Algorithm for executing an ISAc table

into the instruction pointer when the boolean test in their node is true. Notice that to goto node n we issue a jump to node $n-1$ instruction. This is because even after a jump instruction, the instruction pointer is incremented.

The third type of action is the one of interest to the environment outside the ISAc list. We call these *external* actions. Both NOP and jump instructions are related to the internal bookkeeping of the ISAC list. External actions are reported to the simulator running the ISAc list. In the ISAc list shown in Figure 12.1, the external actions are "zero register", "increment register", "decrement x" and "decrement y". Notice that an ISAc list lives in an environment. It sees the environment through its data vector and may modify the environment through its actions.

Done?

As usual, we will generate random objects and randomly modify them during the course of evolution. Since the objects we are dealing with in this chapter have jump statements in them, they will often have infinite loops. This is similar to the situation we faced in Chapter 10. We will adopt a similar solution, but one that is more forgiving of long finite loops or non-action-generating code segments. In any environment that uses ISAc structures, we will place a limit on the total number of instructions that can be executed before fitness evaluation is terminated. Typically this limit will be a small integer multiple of the total number of external instructions we expect the ISAc structure to need to execute to do its job.

Even an ISAc structure that does not get into an infinite loop (or a long finite loop) needs a termination condition. For some applications, having the instruction pointer fall off of the end of the list is an adequate termination condition. The example ISAc list in Figure 12.1 terminates in this fashion. We will call this type of ISAc list a *linear* ISAc list. Another option is to make the instruction pointer function modulo the number of instructions in the

ISAc list. In this variation, the first instruction in the list immediately follows the last. We call this type of ISAc list a *circular* ISAc list. With circular ISAc lists, we either have explicit "done" actions, or we stop when the ISAc list has produced as many external actions as the simulator requires.

Generating ISAc Lists, Variation Operators

Generating an ISAc list is easy. You must choose a data structure, either an array of records (a, b, act, jmp) or 4 arrays $a[]$, $b[]$, $act[]$, $jmp[]$ with records formed implicitly by common index. Simply fill the array with appropriately-sized integers chosen uniformly at random. The values of a and b are in the range $0 \ldots n_v - 1$ where n_v is the number of items in the data vector. The act field is typically in the range $0 \ldots n_a + 1$ where n_a is the number of external actions. The two added actions leave space for the jump and NOP actions. Since NOP and jump are always present, it is a good idea to let action 0 be NOP, action 1 be jump, and then for any other action, subtract 2 from the action's number and return it to the simulator. This will make using ISAc structures evolved on one task for another easier, as they will agree on the syntax of purely ISAc list internal instructions. The jmp field is in the range $0 \ldots listsize$ where $listsize$ is the length of the ISAc list.

The variation operators we will use on ISAc structures should seem comfortably familiar. If we treat individual ISAc nodes as atomic objects, then we can use the string-based crossover operators from Chapter 2. One point, two point, multi-point, and uniform crossover take on their familiar meanings with ISAc lists.

Point mutation of an ISAc structure is a little more complex. There are three sorts of fields in an ISAc node, the data pointer fields, a and b, the action field, act, and the jump field, jmp. A *point mutation* of an ISAc structure selects an ISAc node uniformly at random, selects one of its 4 fields uniformly at random, and then replaces it with a new, valid value. For finer resolution, we also define the *pointer mutation*, which selects a node uniformly at random and then replaces its a or b field, an *action mutation* that selects a node uniformly at random and then replaces its act field, and a *jump mutation* that selects a node uniformly at random and replaces its jmp field.

Data Vectors and External Objects

In Chapter 10, we augmented our basic parse trees with operations that could affect external objects, calculator style memories. In Figure 12.1, various actions available to the ISAc list were able to modify an external scratch register and two registers holding variables. Much as we custom designed the parse tree language to the problem in Chapters 8, 9, and 10, we must custom design the environment of an ISAc list.

The primary environmental feature is the data vector, which holds inputs and constants. Figure 12.1 suggests that modifiable registers are another possible feature of the ISAc en-

vironment. To use memory-mapped I/O, we could permit the ISAc list to directly modify elements of its data vector, taking these as the output. We could give the ISAc list instructions that modify which data set or part of a data set are being reported to it in its data vector. We will deal more with the issues in later chapters, but you should keep them in mind.

With the basic definitions of ISAc structures in hand, we are now ready to take on a programming task. The next section starts us off in familiar territory, the Tartarus environment.

Problems

Problem 12.1 *What does the ISAc structure given in Figure 12.1 do?*

Problem 12.2 *Using the notation from Figure 12.1, give a sequence of ISAc nodes that implement the structure:*

```
while(v[1]>v[2])do Action(5);
```

Problem 12.3 *Using the notation from Figure 12.1, give a sequence of ISAc nodes that implement the structure:*

```
while(v[1]==v[2])do Action(5);
```

Problem 12.4 *Using the notation from Figure 12.1, give a sequence of ISAc nodes that implement the structure:*

```
while(v[1]!=v[2])do Action(5);
```

Problem 12.5 *Using the notation from Figure 12.1, give a sequence of ISAc nodes that implement the structure:*

```
while(v[1]>=v[2])do Action(5);
```

Problem 12.6 *Take the commands given for the ISAc list in Figure 12.1 and add commands for incrementing x and y and decrementing the register. Write a linear ISAc list that places $x - y$ into the register.*

Problem 12.7 Essay. *The code fragments from Problems 12.2, 12.3, 12.4, and 12.5 show that any comparison of two data vector items can be simulated (less-than comparisons simply require we reverse a and b). It may, however, be the case that the cost in space and complexity of simulating the test you need from the one test you are allowed will impede discovery of the desired code. Describe how to modify ISAc lists to have multiple different boolean tests available as primitive operations in an ISAc node.*

Problem 12.8 Essay. *Those readers familiar with Beginners All-purpose Symbolic Instruction Code (BASIC) will recognize that BASIC's method of handling subroutines is easily adapted to the ISAc list environment. For our BASIC-noncompliant readers, a BASIC program has a number associated with each line. The command "GOSUB < linenumber >" transfers control to the line number named. When a "RETURN" command is encountered, control is returned to the line after the most recent "GOSUB" command. Several GOSUB commands can be executed followed by several returns, with a stack of return locations needed to decide where to return. Describe a modification of the ISAc list environment to include jump-like instructions similar to the BASIC GOSUB and RETURN commands. Does the current method for disallowing infinite loops suffice or do we need to worry about growth of the return stack? What do we do if the ISAc list terminates with a nonempty return stack? Should this be discouraged and, if so, how?*

Problem 12.9 *Describe the data vector and external commands needed to specialize ISAc structures to work on the Plus-One-Recall-Store efficient node use problem, described in Chapter 8. For efficient node use, we were worried about the total number of nodes in the parse tree. Be sure to state what the equivalent of "size" is for an ISAc structure and carefully restate the efficient node use problem. Give a solution for size 12.*

Problem 12.10 *Reread Problem 12.9. Is it possible to come up with an ISAc structure that solves a whole class of efficient node use problems?*

Problem 12.11 Essay. *In Chapter 9, we used evolutionary algorithms and genetic programming to encode formulas that were being fit to data. Using an analogy between external ISAc actions and keys on a calculator, explain how to use an evolutionary algorithm to fit a formula embedded in an ISAc list to data. What, if any, real constants go in the data vector? Give an ISAc list that can compute the fake bell curve,*

$$f(x) = \frac{1}{1+x^2}.$$

Be sure to explain your external commands, choice of boolean test, and data vector.

Problem 12.12 *Following the setup in Problem 12.11, give an ISAc structure that can compute the function*

$$f(x) = \begin{cases} x^2 & x \geq 0 \\ -x^2 & x < 0. \end{cases}$$

Problem 12.13 *Following the setup in Problem 12.11, give an ISAc structure that can compute the function*

$$f(x,y) = \begin{cases} 1 & x^2+y^2 \leq 1 \\ \frac{1}{x^2+y^2} & x^2+y^2 > 1. \end{cases}$$

Problem 12.14 *Describe the data vector and external commands needed to specialize ISAc structures to play Iterated Prisoner's Dilemma (see Section 6.2 for details). Having done so, give ISAc lists that play each of the following strategies. Use the commented style of Figure 12.1.*

(i) Always cooperate,

(ii) Always defect,

(iii) Tit-for-tat,

(iv) Tit-for-two-tats,

(v) Pavlov, and

(vi) Ripoff.

Problem 12.15 *Are ISAc lists able to simulate finite state automata in general? Either give an example of a finite state automaton that cannot, for some reason, be simulated by an ISAc list, or give the general procedure for performing such a simulation, i.e., coding a finite state automaton as an ISAc list.*

12.2 Tartarus Revisited

The first thing we will do with ISAc lists is revisit the Tartarus task from Chapter 10. You should reread the description of the Tartarus problem on page 257. We will specialize ISAc lists for the Tartarus problem as follows. We will use a data vector that holds the 8 sensors (see Figure 10.3) and 3 constants, $v = [UM, UR, MR, LR, LM, LL, ML, UL, 0, 1, 2]$. The ISAc actions will be: 0 - NOP, 1 - Jump, 2 - Turn Left, 3 - Turn Right, 4 - Go Forward. This specification suffices for our first experiment.

Experiment 12.1 *Implement or obtain software to create and run circular ISAc lists, as well as the variation operators described in Section 12.1. Be sure to include routines for saving ISAc lists to a file and reading them from a file. With these routines in hand, as well as the Tartarus board routines from Chapter 10, build an evolutionary algorithm that tests ISAc lists specialized for Tartarus on 40 boards. Use 80 moves (external actions) on each board with a limit of 500 ISAc nodes evaluated per board. Use a population of 60 ISAc lists of length 60. Use point mutation and two point crossover for your variation operators and single tournament selection with tournament size 4 for your model of evolution. Perform 20 simulations for 200 generations each and compare your results with Experiment 10.15.*

One thing you may notice, comparing this experiment with Experiment 10.15, is that ISAc lists run much faster than GP-automata. Let's see if we can squeeze any advantage out of this.

Experiment 12.2 *Redo Experiment 12.1, testing ISAc structures on 100 Tartarus boards with population size 400. Run 100 simulations and compare the results with Experiment 12.1. Compare both the first 20 simulations and the full set of 100. Also save the best ISAc structure from each simulation in a file. We will use this file later as a "gene pool."*

In animal breeding, the owner of a high-quality animal can make serious money selling the animal's offspring or stud services. In our attempts to use evolution to locate good structures, we have, for the most part, started over every time with random structures. In the next couple of experiments, we will see if using superior stock as a starting point can give us some benefit in finding good ISAc list controllers for Tartarus dozers. There is an enormous literature on animal breeding. Reading this literature might inspire you with a project idea.

Experiment 12.3 *Modify the code from Experiment 12.2 so that, instead of generating a random initial population, it reads in the 100 best-of-run genes from Experiment 12.2, making 4 copies of each gene as its initial population. Run 25 simulations and see if any of them produce Tartarus controllers superior to the best in the gene pool.*

In Experiment 12.3, we started with only superior genes. There is a danger in this; the very best gene may quickly take over causing us to simply search for variations of that gene. This is especially likely, since each member of a population of superior genes has pretty complex structure that does not admit much disruption; crossover of two different superior genes will often result in an inferior structure. To try to work around this potential limitation in the use of superior stock, we will seed a few superior genes into a population of random genes and compare the result with that of using only superior genes. In future chapters we will develop other techniques for limiting the spread of good genes.

Experiment 12.4 *Modify the code from Experiment 12.3 so that, instead of generating a random initial population, it reads in the 100 best-of-run genes from Experiment 12.2. The software should then select 10 of these superior genes at random and combine them with 390 random ISAc structures to form an initial population. Run 25 simulations, each with a different random selection of the initial superior and random genes, and see if any of them produce Tartarus controllers superior to the best in the gene pool. Also, compare results with those obtained in Experiment 12.3.*

We have, in the past, checked to see if the use of random numbers helped a Tartarus controller (see Experiment 10.8). In that experiment, access to random numbers created a local optimum with relatively low fitness. Using gene pools gives us another way to check if randomness can help with the Tartarus problem.

Experiment 12.5 *Modify your ISAc list software from Experiment 12.4 to have a fourth external action, one that generates a random action. Choose that random action so that it is Turn Left 20% of the time, Turn Right 20% of the time, and Go Forward 60% of the time. Redo Experiment 12.4 permitting this random action in the randomly-generated parts of the initial population, but still reading in random-number-free superior genes. Run 100 simulations and compare the scores of the final best-of-run creatures with those obtained in past experiments. Do the superior creatures use the random action? Did the maximum fitness increase, decline, or remain about the same?*

The choice of length 60 ISAc lists in Experiment 12.1 was pretty arbitrary. In our experience with string baselines for Tartarus in Chapter 10 (Experiment 10.1), string length was a fairly critical parameter. Let us see if very short ISAc structures can still obtain decent fitness scores on the Tartarus problem.

Experiment 12.6 *Modify your ISAc list software from Experiment 12.2 to operate on length 10 and length 20 ISAc lists. Do 100 simulations for each length and compare the results, both with one another and with those obtained in Experiment 12.2. Do these results meet with your expectations?*

We conclude this section with another Tartarus generalization. In the past, we played Tartarus on a 6×6 board with 6 boxes. We now try it on a larger board.

Experiment 12.7 *Modify your ISAc list software from Experiment 12.2, and the Tartarus board routines, to work on an 8×8 board with 10 boxes. Run 100 simulations, saving the best genes in a new gene pool. Verify that fitness, on average, increases over time and give a histogram of your best-of-run creatures.*

The brevity of this section, composed mostly of experiments, is the result of having already investigated the original Tartarus problem in some detail. The Tartarus task is just one of a large number of tasks we could study even in the limited environment of a virtual agent that can turn or go forward on a grid with some boxes. In the next section, we will take a look a several other tasks of this sort that will require only modest variations in software. We leave for later chapters the much harder problem of getting multiple virtual agents to work together.

Problems

Problem 12.16 *In Chapter 10, we baselined the Tartarus problem with fixed sets of moves, and used a simple string evolver (Experiment 10.2) to locate good fixed sets. Give a method for changing such a string of fixed moves into an ISAc structure that (i) exhibits exactly the same behavior, but (ii) is easy to revise with mutation.*

Problem 12.17 *In Experiment 12.1, we permitted up to 500 ISAc list nodes to be evaluated in the process of generating 80 moves on the Tartarus board. This may be an overgenerous allotment. Design a software tool that plots the fitness of an ISAc list for different limits on ISAc nodes. It should perform its tests on a large number of boards. Is there a way to avoid evaluating the ISAc list on one board several times? This is a tool for post-evolution analysis of a fixed ISAc structure.*

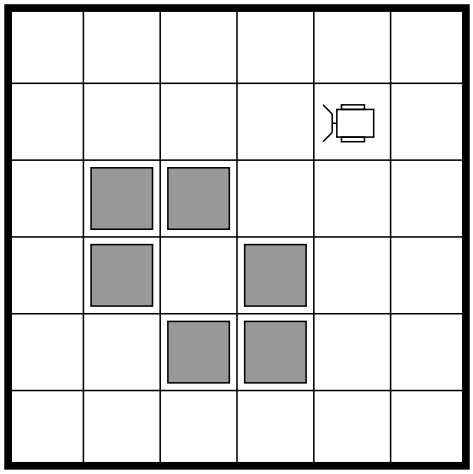


Figure 12.3: An example of an impossible Tartarus starting configuration of the type discovered by Steve Willson

Problem 12.18 *When studying the Tartarus problem, Steve Willson noticed that the close grouping of 4 blocks wasn't the only impossible Tartarus configuration (see Definition 10.1). Shown in Figure 12.3 is another such impossible configuration. Explain why the Willson configuration is impossible. Is it impossible for all initial positions and headings of the dozer?*

Problem 12.19 *Compute the number of starting* 6×6 *Tartarus boards for which there is a close grouping of 4 boxes and the number that are Willson configurations. Which sort of impossible board is more common? Be sure to include dozer starting positions in your count.*

Problem 12.20 *Reread Problem 12.18. Describe an evolutionary algorithm that locates impossible boards. You will probably need to co-evolve boards and dozer controllers. Be careful to choose a dozer controller that evolves cheaply and easily. Be sure, following the example from Problem 12.18, that dozer position is part of your board specification.*

Problem 12.21 Essay. *In the experiment in which we inject a small number of superior genes into a large population of random genes, there is a danger we will only get variations of the best gene in the original population. Discuss a way to use superior genes that*

decreases the probability of getting only variations of those superior genes. Do not neglect non-elite selection, use of partial genes, and insertion of superior genes other than in the initial population. Try also to move beyond these suggestions.

Problem 12.22 *Given below is a length 16 circular ISAc list. It uses the data vector described in this section, and the NOP, jump, and external actions are given explicitly, i.e., ACT 0 is turn left, ACT 1 is turn right, and ACT 2 is go forward. For the Tartarus initial configuration shown in Figure 10.1, trace the action of a dozer controlled by this ISAc list and give the score after 80 moves. Put the numbers 1-80 on a blank Tartarus board, together with heading arrows, to show how the dozer moves.*

```
 0: If v[9]>v[8]   then ACT 2
 1: If v[7]>v[9]   then ACT 1
 2: If v[4]>v[1]   then ACT 2
 3: If v[10]>v[10] then NOP
 4: If v[4]>v[9]   then ACT 2
 5: If v[10]>v[9]  then ACT 1
 6: If v[4]>v[0]   then ACT 2
 7: If v[9]>v[2]   then ACT 2
 8: If v[2]>v[8]   then JMP 3
 9: If v[3]>v[6]   then ACT 0
10: If v[10]>v[7]  then ACT 2
11: If v[7]>v[10]  then NOP
12: If v[6]>v[7]   then ACT 0
13: If v[0]>v[8]   then ACT 2
14: If v[10]>v[7]  then ACT 0
15: If v[3]>v[6]   then ACT 1
```

Problem 12.23 *If we increase board size, are there new variations of the Willson configuration given in Figure 12.3? Please supply either pictures (if your answer is yes) or a mathematical proof (if your answer is no). In the latter case, give a definition of "variations on a Willson configuration."*

Problem 12.24 *Recompute the answer to Problem 12.19 for 6 boxes on an $n \times n$ board.*

Problem 12.25 *Give a neutral mutation operator for ISAc structures. It must modify the structure without changing its behavior. Ideally it should create variation in the children the ISAc list can have.*

Problem 12.26 Essay. *Suppose we use the following operator as a variation operator that modifies a single ISAc list. Generate a random ISAc list, perform two-point crossover between*

the random ISAc list and the one being modified, and pick one of the two children at random as the new version of the structure. Is this properly a mutation operator? Is it an operator that might help evolution? Give a quick sketch of an experiment designed to support your answer to the latter question.

Problem 12.27 *In Experiment 12.5 we incorporate a left-right symmetric random action into the mix. Would an asymmetric random action have been better? Why or why not?*

Problem 12.28 *In Section 10.1 (string baseline), we used gene-doubling and gene-halving mutations (see Definitions 10.2 and 10.3). Give definitions of gene-doubling and gene-halving mutations for ISAc lists. Your mutations should not cause jumps to favor one part of the structure.*

12.3 More Virtual Robotics

In this section, we will study several virtual robotics problems that can be derived easily from Tartarus. They will incorporate modest modifications of the rules for handling boxes and substantial modifications of the fitness function. We will make additional studies using initial populations that have already undergone some evolution. As a starting point, they will, we hope, perform better than random structures. In addition to rebreeding ISAc lists for the same task, we will study crossing task boundaries.

The first task we will study is the *Vacuum Cleaner* task. The Vacuum Cleaner task does not use boxes at all. We call the agent the *vacuum* rather than the dozer. The vacuum moves on an $n \times n$ board and is permitted $n^2 + 2n$ moves: turn left, turn right, go forward, or stand still. When the *vacuum* enters a square, that square is marked. At the end of a trial, the fitness of the vacuum is +1 for the first mark in each square, -1 for each mark after the first in each square. We call this the *efficient cleaning* fitness function. The object is to encourage the vacuum to visit each square on the board once. We will need to add the fourth action, *stand still*, to the external actions of the ISAc list software.

The only variation between boards is the starting position and heading of the vacuum. In addition, the heading is irrelevant in the sense that the average fitness over the set of all initial placements and headings and the average fitness over all initial placements with a single heading are the same. Because of this, we will always start the vacuum with the same heading and either exhaustively test or sample the possible placements. We now perform a string baseline experiment for the Vacuum Cleaner task.

Experiment 12.8 *Modify the Tartarus software to use a world with walls but no boxes and to compute the efficient cleaning fitness function. Use a* 9×9 *board and test fitness on a single board starting the vacuum facing north against the center of the south wall. Use a string evolver to evolve a string of 99 moves over the alphabet* {*left, right, forward, stand still*

} *in three ways. Evolve 11-character strings, 33-character strings, and 99-character strings, using the string cyclically to generate 99 moves in any case.*

Have your string evolver use two point crossover and 0-n point mutations (where n is the length of the string divided by 11). Take the model of evolution to be single tournament selection with tournament size 4. Run 100 simulations for up to 400 generations on a population of 100 strings, reporting time-to-solution and average and maximum fitness for each simulation. Compare different string lengths and, if any are available, trace one maximum fitness string.

With a baseline in hand, we now will evolve ISAc structures for the Vacuum Cleaner task. Notice we are sampling from 16 of the 81 possible initial placements, rather than computing total fitness.

Experiment 12.9 *Modify the evolutionary algorithm from Experiment 12.8 to operate on ISAc structures of length 60. Evolve a population of 400 ISAc lists, testing fitness on 16 initial placements in each generation. Report fitness as the average score per board. Use two point crossover and from 0-3 point mutations, with the number of point mutations selected uniformly at random. Perform 100 simulations lasting at most 400 generations, saving the best ISAc list in each simulation for later use as a gene pool. Plot the fitness as a function of the number of generations and report the maximum fitnesses obtained. How did the ISAc structures compare with the string baseline?*

We will now jump to testing on total fitness (all 81 possible placements), using the gene pool from the last experiment. The hope is that evolving for a while on a sampled fitness function and then evolving on the total fitness function will save time.

Experiment 12.10 *Modify the evolutionary algorithm from Experiment 12.9 to read in the gene pool generated in that experiment. For initial populations, choose 10 genes at random from the gene pool and 390 random structures. Evolve these populations, testing fitness on all 81 possible initial placements in each generation. Report fitness as the average score per board. Perform 100 simulations lasting at most 400 generations. Plot the fitness as a function of the number of generations and report the maximum fitnesses obtained.*

The Vacuum Cleaner task is not, in itself, difficult. Unlike Tartarus, where a perfect solution is difficult to specify, it is possible to simply write down a perfect solution to the Vacuum Cleaner task. You are asked to do this in Problem 12.35. The Vacuum Cleaner task does, however, require that the ISAc structure build some sort of model of its environment and learn to search the space efficiently. This efficient space searching is a useful skill and makes the Vacuum Cleaner gene pool from Experiment 12.9 a potentially valuable commodity. In the next task, we will use this efficient space sweeping as a starting point for learning a new skill, eating.

htb

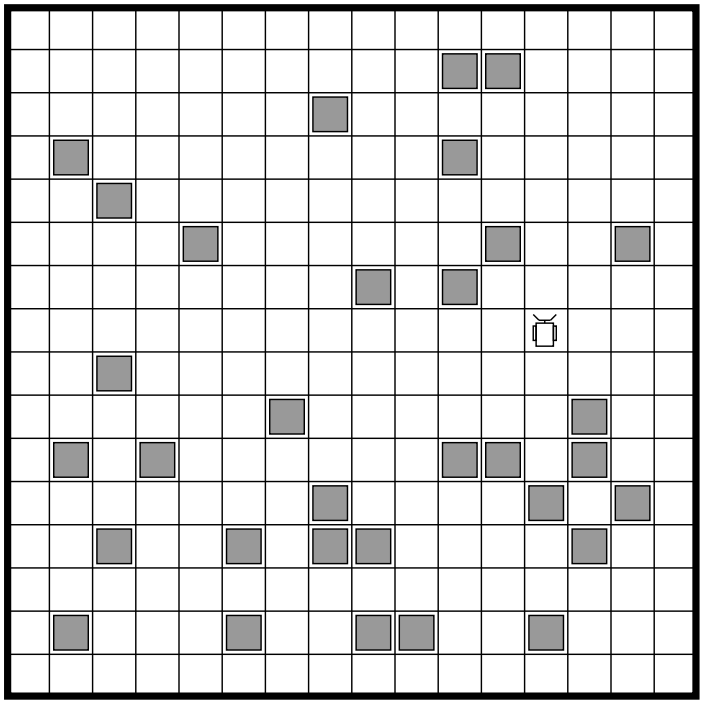


Figure 12.4: A valid starting configuration for the Herbivore task

The *Herbivore* task will add a new agent name to our roster. Tartarus has the dozer; the Vacuum Cleaner task has the vacuum. The agent used in the Herbivore task is called the *cowdozer*. The Herbivore task takes place in an $n \times n$ world and uses boxes. The rules for boxes, pushing boxes, and walls are the same as Tartarus, save that an additional action is added: the *eat* action. If the cowdozer is sitting with a box directly ahead of it and it executes an *eat* action, then the box vanishes. Our long term goal is to create agents that can later be used in an ecological simulation. For now, we merely wish to get them to eat efficiently.

A single Herbivore board is prepared by scattering k boxes at random on the board. These boxes may be anywhere, as the eat action causes a complete lack of "impossible" boards. A valid starting configuration is shown in Figure 12.4. Be sure to keep the numbering of ISAc actions (NOP, jump, turn left, turn right, and go forward) consistent in the Vacuum Cleaner and Herbivore tasks. Give *stand still* and *eat* the same action index. This facilitates the use of a Vacuum Cleaner gene pool as a starting point for Herbivore. For the Herbivore task, our fitness function will be the number of boxes eaten.

With the Vacuum Cleaner task, it wasn't too hard to select an appropriate number of moves. The task of justifying that choice is left to you (Problem 12.30). For the Herbivore task, this is a harder problem. The cowdozer must search the board and, so, would seem

to require at least as many moves as the vacuum. Notice, however, that the cowdozer does not need to go to every square - rather, it must go beside every square. This is all that is required to find all the boxes on the board. On the other hand, the dozer needs to turn toward and eat the boxes. This means, for an $n \times n$ board with k boxes, we need some fraction of $n^2 + 2n$ moves plus about $2k$ moves. We will err on the side of generosity and compute moves according to Equation 12.1.

$$moves(n, k) = \frac{2}{3}n^2 + 2(n + k),\ n \times n \text{ board}, k \text{ boxes}. \tag{12.1}$$

We will now do a series of 5 experiments that will give us an initial understanding of the Herbivore task and its relation to the Vacuum Cleaner task.

Experiment 12.11 *Modify your board maintenance software to permit the eat action and the generation of random Herbivore boards. Be sure to be able to return the number of boxes eaten; this is the fitness function. With the new board routines debugged, use the simulation parameters from Experiment 12.8, except as stated below, to perform a string baseline for the Herbivore task. Use board size $n = 9$ with $k = 27$ boxes, and do 126 moves per board. Test fitness on 20 random boards. Use strings of length 14, 42, and 126. Do 0-q point mutations, where q is the string length divided by 14. Run 100 simulations and save the best strings from each of the 42-character simulations in a gene pool file. Plot average and maximum fitness as a function of time.*

Now we have a string baseline, we can do the first experiment with adaptive structures.

Experiment 12.12 *Modify the evolutionary algorithm from Experiment 12.11 to operate on ISAc structures of length 60. Evolve a population of 400 ISAc lists, testing fitness on 100 Herbivore boards. Report fitness as the average score per board. Use two point crossover and from 0-3 point mutations, with the number of point mutations selected uniformly at random. Perform 100 simulations lasting at most 400 generations, saving the best ISAc list in each simulation for later use as a gene pool. Plot the fitness as a function of the number of generations and report the maximum fitnesses obtained. How did the ISAc structures compare with the string baseline?*

And now let's check to see how the breeding-with-superior-genes experiment works with Herbivore genes.

Experiment 12.13 *Modify the evolutionary algorithm from Experiment 12.12 to read in the gene pool generated in that experiment. For initial populations, choose 10 genes at random from the gene pool and 390 random structures. Evolve these populations, using the same fitness evaluation as in Experiment 12.12. Report fitness as the average score per board. Perform 100 simulations, lasting at most 400 generations. Plot the fitness as a function of the number of generations and report the maximum fitnesses obtained.*

At this point, we will try something completely different: starting a set of Herbivore simulations with Vacuum Cleaner genes. There are three general outcomes for such an experiment: vacuum genes are incompetent for the Herbivore task and will never achieve even the fitness seen in Experiment 12.12; vacuum genes are no worse than random, and fitness increase will follow the same sort of average track it did in Experiment 12.12; Vacuum Cleaner competence is useful in performing the Herbivore task, so some part of the fitness track will be ahead of Experiment 12.12 and compare well with that from Experiment 12.13.

Experiment 12.14 *Redo Experiment 12.13 generating your initial population as follows. Read in the gene pool generated in Experiment 12.9 and use 4 copies of each gene. Do not use any random genes. For each of these genes, scan the gene for NOPs and, with probability 0.25 for each NOP, replace them with eat actions. Stand still actions should already use the same code as eat actions, and eat replaces stand still as the fourth action, beyond the basic three used in Tartarus. Perform 100 simulations, plotting average fitness and maximum fitness over time. Compare with the fitness tracks from Experiments 12.12 and 12.13.*

Now, we look at another possible source of superior genes for the Herbivore task, our string baseline of that task. In Problem 12.16, we asked you to outline a way of turning strings into ISAc structures. We will now test at least one version of that notion.

Experiment 12.15 *Redo Experiment 12.13 with an initial population of 84-node ISAc lists generated as follows. Read in the gene pool of length 42 strings generated in Experiment 12.11, transforming them into ISAc structures by taking each entry of the string and making it into an ISAc node of the form:* If$(1 > 0)$Act*(string-entry), i.e., an always-true test followed by an action corresponding to the string's action. After each of these string-character actions, put a random test together with a NOP action. All JMP fields should be random. Transform each string 4 times with different random tests together with the NOPs. Do not use any random genes. Perform 100 simulations, plotting average fitness and maximum fitness over time. Compare with the fitness tracks from Experiments 12.12, 12.13, and 12.14.*

We have tried a large number of different techniques to build good cowdozer controllers. In a later chapter, these cowdozers will become a resource as the starting point for ecological simulations. We now move on in our exploration of ISAc list robotics and work on a task that is different from Tartarus, Vacuum Cleaner, and Herbivore in several ways. The *North Wall Builder* task tries to get the agent, called the *constructor*, to build a wall across the north end of the trial grid.

The differences from the grid-robotic tasks we have studied so far are as follows. First, there will be a single fitness case and, so, no need to decide how to sample the fitness cases. Second, while there are blocks, the blocks are delivered to the board in response to the actions of the constructor. Third, we remove the walls at the edges of the world. The constructor will have the same 8 sensors that the other grid-robots had and will still detect a

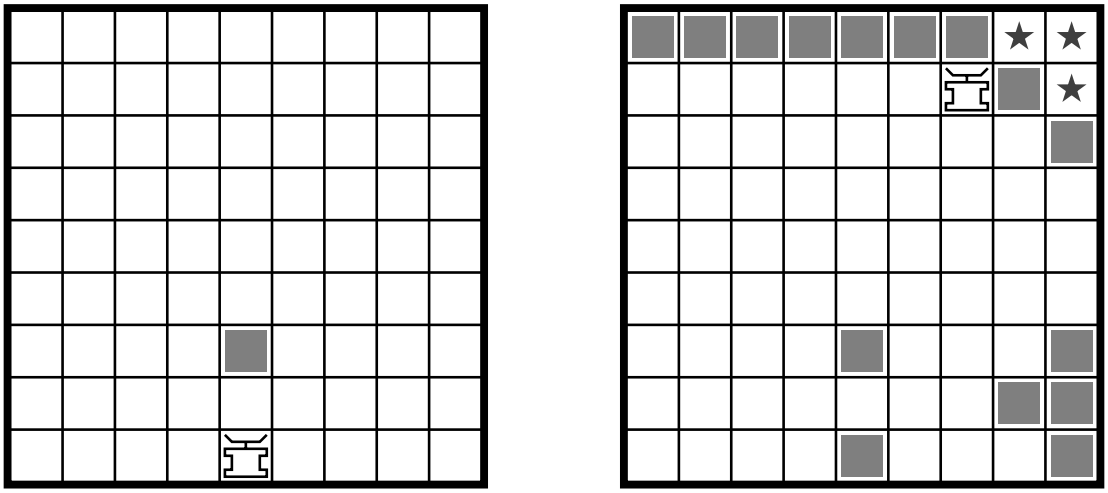


Figure 12.5: The starting configuration and a final configuration for the North Wall Builder task (The stars denote uncovered squares, on which the constructor lost fitness.)

"2" at the edge of the world. What changes is the results of pushing a box against the edge of the board and of a go forward action when facing the edge of the board. A box pushed against the edge of the board vanishes. If the constructor attempts to go forward over the edge of the board, it vanishes, and its fitness evaluation ends early. For a given constructor, the *survival time* of the constructor is the number of moves it makes without falling off of the board.

The starting configuration and a final configuration are shown in Figure 12.5. North Wall Builder uses a square board with odd side lengths. In order to evolve constructors that build a north wall, we compute fitness as follows. Starting at the north edge of the board in each column, we count the number of squares before the first box. These are called *uncovered squares*. The *North Wall Builder fitness function*(NWB fitness function) is: the number of squares on the board minus the number of uncovered squares. The final configuration show in Figure 12.5 gets a fitness of $9 \times 9 - 3 = 78$.

The box delivery system is placed as shown in Figure 12.5, two squares from the south wall in the center of the board along the east-west axis. It starts with a box in place and the constructor pushes boxes as in Tartarus, save that boxes and the constructor may fall off the edge of the board. If the position of the box delivery system is empty (no box or constructor on it), then a new box appears. The constructor always starts centered against the south wall facing north toward the box delivery system. We permit the constructor $6n^2$ moves on an $n \times n$ board. Having defined the North Wall Builder task, we can begin experiments with

a string baseline.

Experiment 12.16 *Modify your board software to implement the rules for the North Wall Builder task, including pushing boxes off the edge of the world, returning a value that means the constructor has fallen off the edge of the world, and computing the NWB fitness function. In addition to fitness, be sure to write the routines so that survival time is computed. Having done this, run a string baseline experiment like Experiments 12.8 and 12.11. Use a population of 100 strings of length 100, cycling through the strings to generate the required number of moves. Work on a* 9×9 *board, allowing each constructor 486 steps. Report maximum and average fitness and survival time as a function of generations of evolution, performing 30 simulations. Add survival time as a tiebreaker in a lexical fitness function and perform 30 additional simulations. Does it help?*

If your string baseline for the North Wall Builder task is working the way ours did then you will have noticed that long survival times are somewhat rare. In North Wall Builder, reactivity (the ability to *see* the edge of the world) turns out to be quite valuable. We now can proceed to an ISAc experiment on North Wall Builder.

Experiment 12.17 *Modify the evolutionary algorithm from Experiment 12.16 to operate on ISAc structures of length 100. Evolve a population of 400 ISAc lists, using the NWB fitness function by itself and then the lexical product of NWB with survival time, in different sets of simulations. Report mean and maximum fitness and survival time. Use two point crossover and from 0-3 point mutations, with the number of point mutations selected uniformly at random. Perform 100 simulations for each fitness function, lasting at most 400 generations. Save the best ISAc list in each simulation for later use as a gene pool. How did the ISAc structures compare with the string baseline for North Wall Builder? Did the lexical fitness help? Did it help as much as it did in the string baseline? Less? More?*

What is the maximum speed at which the constructor can complete the NWB task? In the next experiment, we will see if we can improve the efficiency of the constructors we are evolving by modifying the fitness function. Suppose that we have two constructors that get the same fitness, but one puts its blocks into their final configuration several moves sooner than the other. The faster constructor will have more "post fitness" moves for evolution to modify to shove more boxes into a higher fitness configuration. This suggests we should place some emphasis on brevity of performance.

One way to approach this is to create a *time-averaged* fitness function. At several points during the constructor's allotment of time, we stop and compute the fitness. The fitness used for selection is the average of these values. The result is that fitness gained as the result of block configurations that occur early in time contributes more than such fitness gained later. Notice that we presume that there are reasonable construction paths for the constructor for which a low intermediate fitness is not required.

Experiment 12.18 *Modify the software from Experiment 12.17 to use an alternate fitness function. This function is the average of the old NWB fitness function sampled at time-steps 81, 162, 243, 324, 405, and 486 (n^2, $2n^2$, ..., $6n^2$). Retain the use of survival time in a lexical fitness function. Use the new fitness function for selection.*

Report the mean and maximum of the new fitness function, the old fitness function at each of the 6 sampling points*, and the survival time. Perform 100 simulations. Does the new fitness function aid in increasing performance as measured by the original fitness function? Do the data suggest an answer to Problem 12.48?*

Problem 12.52 asks you to speculate on the value of a gene pool evolved for one size of board for another size of board. We will now look at the answer, at least for the North Wall Builder task.

Experiment 12.19 *Modify the software from Experiment 12.17 to use* 11×11 *and* 13×13 *boards. Use the gene pool generated in Experiment 12.17 for population seeding. Create initial populations either by uniting 10 of the genes from the gene pool, selected uniformly at random, with 390 random genes, or by generating 400 random genes. Using the NWB fitness function* lex *survival time, run 100 simulations for at most 400 generations for each of the two new board sizes and for each of the initial populations. Report mean, deviation, and maximum of fitness and survival time. Did the genes from the gene pool help?*

As has happened before, the combinatorial closure of the experiments we *could* have performed is enormously larger than those we did perform. You are encouraged to try other experiments (and please write us, if you find a good one). Of especial interest is more study of the effect skill at one task has on gene-pool quality for another task. This chapter is not the last we will see of ISAc structures; we will look at them in the context of epidemiology and ecological modeling in future chapters.

Problems

Problem 12.29 *On page 320 it is asserted that, for the Vacuum Cleaner task, the average fitness over the set of all initial placements and headings and the average fitness over all initial placements with a single heading are the same. Explain why this is so.*

Problem 12.30 *For an* $n \times n$ *board is* $n^2 + 2n$ *a reasonable number of moves for the Vacuum Cleaner task? Why or why not?*

Problem 12.31 Short Essay. *Given the way fitness is computed for the Vacuum Cleaner task, what use is the stand still action? If it were eliminated, would solutions from a evolutionary algorithm tend to get better or worse? Explain.*

Problem 12.32 *Are the vacuum's sensors any use? Why or why not?*

Problem 12.33 *Would removing the walls and permitting the board to wrap around at the edges make the Vacuum Cleaner task harder or easier? Justify your answer in a few sentences.*

Problem 12.34 *Is the length of an ISAc list a critical parameter, i.e., do small changes in the lengths of the ISAc lists used in an evolutionary algorithm create large changes in the behavior of the evolutionary algorithm, on average? Justify your answer in a few sentences.*

Problem 12.35 *For a* 9×9 *board, give an algorithm for a perfect solution to the Vacuum Cleaner task. You may write pseudo-code or simply give a clear statement of the steps in English. Prove, probably with mathematical induction, that your solution is correct.*

Problem 12.36 *In other chapters, we have used neutral mutations, mutations that change the gene without changing fitness, as a way of creating population diversity. Suppose we change all turn lefts to turn rights and turn rights to turn lefts in an agent. If we are considering fitness computed over all possible starting configurations, then is this a neutral mutation for (i) Tartarus, (ii) Vacuum Cleaner, (iii) Herbivore, or (iv) North Wall Builder? Justify your answers in a few sentences.*

Problem 12.37 *Experiment 12.11 samples 20 boards to estimate fitness. How many boards are in the set from which this sample is being drawn?*

Problem 12.38 *Suppose instead of having walls at the edge of the board in the Herbivore task we have the Herbivore board wrap around, left to right and top to bottom, creating a toroidal world. Would this make the task easier or harder? Would the resulting genes be better or worse as models of foraging herbivores? Explain.*

Problem 12.39 *Suppose that we were to use parse trees to code cowdozers for the Herbivore task. Let the data type for the parse trees be the integers (mod 4) with output interpreted as 0=turn left, 1=turn right, 2=go forward, 3=eat. Recalling that 0=empty, 1=box, 2=wall, explain in plain English the behavior of the parse tree (+* x_0 *2), where* x_0 *is the front middle sensor.*

Problem 12.40 Short Essay. *How bad a local optimum does the parse tree described in Problem 12.39 represent? What measures could be taken to avoid that optimum?*

Problem 12.41 *Compute the expected score of the parse tree described in Problem 12.39 on a* 16×16 *board with 32 boxes.*

Problem 12.42 *Give the operations and terminals of a parse tree language on the integers (mod 4) in which the parse tree described in Problem 12.39 could appear. Now write a parse tree in that language that will score better than (+ x_0 2). Show on a couple of example boards* why *your tree will outscore (+ x_0 2). Advanced students should compute and compare the expected scores.*

Problem 12.43 *For the four grid-robotics tasks we've looked at in this chapter (Tartarus, Vacuum Cleaner, Herbivore, and North Wall Builder), rate the tasks for difficulty (i) for a person writing a controller, and (ii) for an evolutionary algorithm. Justify your answer. Is the number of possible boards relevant? The board size?*

Problem 12.44 *Invent and describe a new grid-robotics task with at least one action not used in the grid-robotics tasks studied in this chapter. Make the task interesting and explain why you think it is interesting.*

Problem 12.45 *In Experiment 12.14, we used 4 copies each of the Vacuum Cleaner gene pool members to create an initial population. Given the three possible classes of outcomes (listed on page 324) that the experiment was attempting to distinguish amongst, explain why we did* not *include random genes in the initial population.*

Problem 12.46 *In Experiment 12.14, we tried using a Vacuum Cleaner gene pool as a starting point for an evolutionary algorithm generating Herbivore controllers. For each pair of the four tasks we study in this chapter, predict if using a gene pool from one task would be better or worse than starting with a random population for another. Give one or two sentences of justification for each of your predictions.*

Problem 12.47 *Does using survival time in a lexical fitness for the North Wall Builder create the potential for a bad local optimum? If so, how hard do you estimate it is to escape and why, if not, explain why not.*

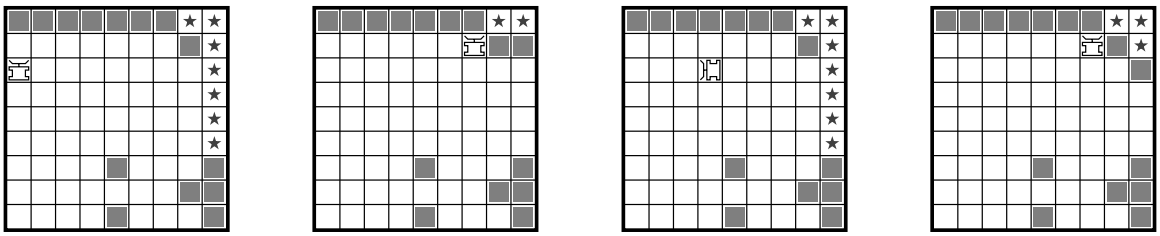
Problem 12.48 *Is the allotment of* $6n^n$ *moves for a constructor to complete the North Wall Builder task generous or tight fisted? Why? (see Experiment 12.18)*

Problem 12.49 Short Essay. *In Experiment 12.18, we examine the use of time-averaged fitness to encourage the constructor to act efficiently. On page 326, it is asserted that for this to help, it must be possible to reach high fitness configurations without intermediate low fitness ones. In other words, fitness should pretty much climb as a function of time. First, is the assertion correct? Explain. Second, are there solutions to the North Wall Builder task in which the fitness does not increase as a function of time?*

Problem 12.50 *Hand code, in the language of your choice or in pseudo-code, a perfect solution to the North Wall Builder task. Beginning students work on a* 9×9 *board; advanced students provide a general solution.*

Problem 12.51 *A* macromutation *is a map that takes a gene to a single other gene making potentially large changes. Give, in a few sentences, a method for using gene pool members in a macromutation operator.*

Problem 12.52 Essay. *For the Tartarus, Vacuum Cleaner, Herbivore, and North Wall Builder tasks, we can vary the board size. With this in mind, cogitate on the following question: would a gene pool created from experiments on one board size be a good starting point for experiments on another board size? Feel free to cite evidence.*



Problem 12.53 Short Essay. *The 4 boards above are final states for the North Wall Builder task. They came from simulations initialized with gene pools, in the manner of Experiment 12.14. What can you deduce from these boards about the process of evolution from a gene-pool-derived initial population containing no random genes? Is anything potentially troublesome happening?*

12.4 Return of the String Evolver

This section includes some ideas not in the main stream of the ISAc list material, but suggested by it and valuable. Thus far, we have done several string baseline experiments: 10.2, 10.3, 10.4, 10.5, 12.8, 12.11, and 12.16. The 4 experiments from Chapter 10 developed the following methodology for a string baseline of a virtual robotics experiment.

The simplest string baseline evolves fixed-length strings of a length sufficient to supply a character for each move desired in the simulation (Experiment 10.2). The next step is to use shorter strings, cyclically, as in Experiment 10.3. The advantage of this is not that better solutions are available - it is elementary that they are not - but rather that it is much easier for evolution to *find* tolerably good solutions this way. Experiments 10.4 and 10.5 combine the two approaches, permitting evolution to operate on variable-length strings. This permits discovery at short lengths (where it is easier), followed by revision at longer lengths, reaping the benefits of small and large search spaces, serially.

In the baseline experiments in the current chapter, we simply mimicked Experiment 10.3 to get some sort of baseline, rather than pulling out all the stops and getting the best possible baseline. If your experiments went as ours did, this string baseline produced

some surprising results. The difference between the string baseline performance and the adaptive agent performance is modest, but significant, for the North Wall Builder task. In the Herbivore task, the string baseline substantially outperforms the parse tree derived local optimum described in Problem 12.39.

The situation for string baselines is even more grim than one might suppose. In preparing to write this chapter, we first explored the North Wall Builder task with a wall at the edge of the world, using GP-automata and ISAc structures. Much to our surprise, the string baseline studies, while worse on average, produced the only perfect gene (fitness 81 on a 9×9 board). The string baseline showed that, with walls, the North Wall Builder task was too easy. With the walls removed, the adaptive agents, with their ability to not walk off the edge of the board, outperformed our string baseline population. This is why we present the wall-free version of the North Wall Builder task.

At this point, we will introduce some terminology and a point of view. An agent is *reactive*, if it changes its behavior based on sensor input. The parse trees evolved in Experiment 10.7 were purely reactive agents. An agent is *state conditioned*, if it has some sort of internal state information. The test for having internal state information is: does the same input result in different actions at different times (neglect the effect of random numbers). An agent is *stochastic*, if it uses random numbers.

Agents can have any or all of these three qualities and all of the qualities can contribute to fitness. Our best agents thus far have been reactive, state conditioned, non-stochastic agents. When used individually, we find that the fitness contributions for Tartarus have the order:

$$\text{reactive} < \text{stochastic} < \text{state conditioned}.$$

In other words, purely reactive agents (e.g., parse trees with no memory of any sort) perform less well than tuned random number generators (e.g., Markov chains), which in turn achieve lower fitness than purely state conditioned agents (e.g., string controllers).

Keeping all this in mind, we have clear evidence that the string baselines are important. Given that they are also a natural debugging environment for the simulator of a given virtual robotic world, it makes no sense not to blood a new problem on a string baseline. In the remainder of this section, we will suggest new and different ways to perform string baseline studies of virtual robotics tasks.

Our first attempt to extend string controllers involves exploiting a feature like the NOP instruction in ISAc lists. In Experiments 10.3, 10.4, and 10.5, we tinkered with varying the length of the string with a fairly narrow list of possibilities. The next experiment improves the granularity of these attempts.

Experiment 12.20 *Start with either Experiment 12.11 (Herbivore) or Experiment 12.16 (North Wall Builder). Modify the string evolver to use a string of length 30 over the alphabet consisting of the actions for the virtual robotics task in question, together with the null character "*". Start with a population of 400 strings, using the string cyclically to generate*

actions, ignoring the null character. Perform 100 simulations and compare with the results for the original string baseline. Plot the fraction of null actions as a function of time: are there particular numbers of null actions that seem to be desirable? Now redo the experiment, but with a 50% chance of a character being null, rather than a uniform distribution. What effect does this have?

The use of null characters permits insertion and deletion, by mutation, into existing string controllers. It is a different way of varying the length of strings. We now will create reactive strings for the Herbivore environment and subject them to evolution. Examine Figure 12.6. This is an alphabet in which some characters stand for one of two actions, depending on information available to the cowdozer's sensors. We call a character *adaptive*, if it codes for an action dependent on sensory information. Non-adaptive characters include the traditional actions and the null character from Experiment 12.20.

Character	Meaning	Adaptive
L	Turn left	No
R	Turn right	No
F	Move forward	No
E	Eat	No
A	If box left, turn left, otherwise go forward	Yes
B	If box right, turn right, otherwise go forward	Yes
C	If wall ahead, turn left, otherwise go forward	Yes
D	If wall ahead, turn right, otherwise go forward	Yes
Q	If box ahead, eat, otherwise go forward	Yes
$*$	Null character	No

Figure 12.6: Alphabet for the adaptive Herbivore string controller

Experiment 12.21 *Rebuild your Herbivore board routines to work with the adaptive alphabet described in Figure 12.6,* except *for the null character. Run an evolutionary algorithm operating on a population of 400 adaptive strings of length 30, used cyclically during fitness evaluation. For fitness evaluation, use a sample of 20 boards to approximate fitness. Use* 9×9 *Herbivore boards. Use single tournament selection with tournament size 4, two point crossover, and 0-3 point mutations with the number of mutations selected uniformly at random.*

Perform 100 simulations and compare with other experiments for the Herbivore task. Save the fraction of adaptive characters in the population and plot this in your write up. Now, perform these simulations again with the null character enabled. Was the effect different from that in Experiment 12.20 (assuming comparability)?

The adaptive characters used in Experiment 12.21 are not the only ones we could have chosen. If we include failure to act, there are $\binom{5}{2} = 10$ possible pairs of actions. Each pair could be chosen between, based on information from any of 8 sensors with 3 return values. One is tempted to use a meta-evolutionary algorithm to decide which adaptive characters are the most valuable (but one refrains). Rather, we look at the preceding experiment's ability to test the utility of adaptive characters and note that stochastic characters can also be defined. A stochastic character is one that codes for an action dependent on a random number. In Figure 12.7, we give a stochastic alphabet.

Character	Meaning	Stochastic
L	Turn left	No
R	Turn right	No
F	Move forward	No
E	Eat	No
G	Turn right, turn left or go forward with equal probability	Yes
H	Turn right 20%, turn left 20%, go forward 60%	Yes
I	Turn left or go forward with equal probability	Yes
J	Turn right or go forward with equal probability	Yes
K	Turn left 30%, go forward 70%	Yes
M	Turn right 30%, go forward 70%	Yes
$*$	Null character	No

Figure 12.7: Alphabet for the stochastic Herbivore string controller

Experiment 12.22 *Rebuild your Herbivore board routines to work with the stochastic alphabet described in Figure 12.7,* except *for the null character. Run an evolutionary algorithm operating on a population of 400 adaptive strings of length 30, used cyclically during fitness evaluation. For fitness evaluation, use a sample of 20 boards to approximate fitness. Use* 9×9 *Herbivore boards. Use single tournament selection with tournament size 4, two point crossover, and 0-3 point mutations with the number of mutations selected uniformly at random.*

Perform 100 simulations and compare with other experiments for the Herbivore task. Save the fraction of stochastic and of each type of stochastic characters in the population

and plot this in your write up. Now, perform these simulations again with the null character enabled. Compare with other experiments and comment on the distribution of the stochastic characters within a given run.

We conclude with a possibly excessive experiment with a very general sort of string controller. We have neglected using string doubling and halving mutations on our adaptive and stochastic alphabets; these might make nice term projects for students interested in low level design of evolutionary algorithm systems. Other, more bizarre possibilities are suggested in the Problems.

Experiment 12.23 *Rebuild your Herbivore board routines to work with the union of the adaptive and stochastic alphabets described in Figures 12.6 and 12.7,* except *for the null character. Run an evolutionary algorithm operating on a population of 400 adaptive strings of length 30, used cyclically during fitness evaluation. For fitness evaluation, use a sample of 20 boards to approximate fitness. Use* 9×9 *Herbivore boards. Use single tournament selection with tournament size 4, two point crossover, and 0-3 point mutations, with the number of mutations selected uniformly at random.*

Perform 100 simulations and compare with other experiments for the Herbivore task. Save the fraction of stochastic, of adaptive, and of each type of character in the population and plot these in your write up. Comment on the distribution of the types of characters within a given run.

Problems

Problem 12.54 *On page 331, it is asserted that string controllers, like those from Experiments 10.2, 10.3, 10.4, 10.5, 12.8, 12.11, and 12.16 are purely state conditioned agents. Explain why they are not reactive or stochastic and identify the mechanism for storage of state information.*

Problem 12.55 *Give pseudo-code for transforming adaptive string controllers, ala Experiment 12.21, into ISAc lists with the same behavior. Hint: write code fragments for the adaptive characters and then use them.*

Problem 12.56 *Give a segment of an ISAc list that cannot be simulated by adaptive string controllers of the sort used in Experiment 12.21.*

Problem 12.57 *How many different adaptive characters of the type used in Experiment 12.21 are there, given choice of actions and test conditions?*

Problem 12.58 *Give and defend an adaptive alphabet for the Tartarus problem. Include an example string controller.*

Problem 12.59 *Give and defend an adaptive alphabet for the Vacuum Cleaner task. Include an example string controller.*

Problem 12.60 *Give and defend an adaptive alphabet for the North Wall Builder task. Include an example string controller.*

Problem 12.61 *Examine the adaptive alphabet given in Figure 12.6. Given the Q character is available what use is the E character? Do not limit your thinking to its use in finished solutions; can the E character tell us anything about the evolutionary algorithm?*

Problem 12.62 Essay. *Stipulate that it is easier to search adaptive alphabets for good solutions, even though they code for a more limited collection of solutions. Explain how to glean adaptive characters from evolved ISAc lists and do so from some evolved ISAc lists for one of the virtual robotics tasks studied in this chapter.*

Problem 12.63 *Either code the Herbivore strategy from Problem 12.39 as an adaptive string controller (you may choose the length) or explain why this is impossible.*

Problem 12.64 Short Essay. *Does the lack of stochastic actions involving eating represent a design flaw in Experiment 12.22?*

Problem 12.65 *Give and defend a stochastic alphabet for the Tartarus problem or explain why any stochasticity would be counter-indicated. Include an example string controller, if you think stochasticity could be used profitably.*

Problem 12.66 *Give and defend a stochastic alphabet for the Vacuum Cleaner task or explain why any stochasticity would be counter-indicated. Include an example string controller if you think stochasticity could be used profitably.*

Problem 12.67 *Give and defend a stochastic alphabet for the North Wall Builder task or explain why any stochasticity would be counter-indicated. Include an example string controller if you think stochasticity could be used profitably.*

Problem 12.68 *Reread Problems 10.7, 10.31, and 10.33. Now, reread Experiment 12.15. The thought in Experiment 12.15 was to transform a string baseline gene into an ISAc list. In the three problems from Chapter 10, we were using Markov chains as controllers for the Tartarus problem. Explain why a string controller is a type of (deterministic) Markov chain. Explain how to transform a string gene into a deterministic Markov chain that can lose its determinism by mutation, sketching an evolutionary algorithm for starting with string genes and evolving good Markov controllers.*

Problem 12.69 Short Essay. *Reread Experiment 12.22 and then answer the following question: does a Markov chain controller ever benefit from having more states than the number of types of actions it needs to produce? Explain.*

Problem 12.70 *Give an evolutionary algorithm that locates good adaptive or stochastic characters. It should operate on a population of characters* and *a population of strings using those characters, simultaneously, so as to avoid using a meta-evolutionary (multi-level) algorithm.*