Genetic Programming and AI Planning Systems[1]

Lee Spector

School of Communications and Cognitive Science
Hampshire College, Amherst, MA 01002
lspector@hamp.hampshire.edu

Abstract

Genetic programming (GP) is an automatic programming technique that has recently been applied to a wide range of problems including blocks-world planning. This paper describes a series of illustrative experiments in which GP techniques are applied to traditional blocks-world planning problems. We discuss genetic planning in the context of traditional AI planning systems, and comment on the costs and benefits to be expected from further work.

Introduction

Genetic programming (GP) is an automatic programming technique developed by Koza that extends the genetic algorithm framework of Holland (Holland 1992). Whereas the conventional genetic algorithm uses evolution-inspired techniques to manipulate and produce fixed-length chromosome strings that encode solutions to problems, GP manipulates and produces computer programs. Koza shows how programs can be "evolved" to solve a wide range of otherwise unrelated problems (Koza 1992).

Several of the problems that Koza describes are of interest to AI planning research. These include control programs for artificial ants, box-moving robots, wall-following robots, and block-stacking systems. The block-stacking problems are closest to the classic problems in the literature of AI planning systems, but Koza uses an unusual variant of blocks-world, making it difficult to relate his results to those of mainstream AI planning research (Tate et al. 1990).

In this paper we apply GP to the block-stacking problems that have been central in the literature of AI planning. In particular, we describe experiments in using GP techniques to 1) find a plan to achieve a single goal from a single initial state, 2) find a "universal plan" for achieving a single goal from a range of initial states, 3) find a domain-dependent planning program, capable of producing action sequences to achieve different sets of goals from a variety of initial states. We conclude that while GP has much to offer to AI planning research, more work must be done to determine exactly how it can be best applied.

Genetic Programming

GP works with a large population of candidate programs and uses the Darwinian principle of "survival of the fittest" to produce successively better programs for a given problem. To use GP one must choose the primitive elements (*functions* and *terminals*) out of which the programs will be constructed.[2] Every terminal in the terminal set and every value that may be returned by any function in the function set must be acceptable as an input for every argument position of every function in the function set; this is called the *closure* property.

The programmer wishing to employ GP must also produce a problem-specific fitness function. This function must take a program as input, producing a number that indicates the "fitness" of the program as output. This describes "how good" the program is at solving the problem under consideration, and determines the likelihood that the program and its offspring will survive to subsequent generations. In this paper all fitness values are "standardized fitness" values, for which *lower* fitness values indicate *better* programs (Koza 1992, p. 96).

Fitness is normally assessed by running the program on some number of *fitness cases*, each of which establishes inputs to the program and describes the corresponding output that the individual program should produce. One is often interested in producing a program that works over a very large, perhaps infinite, set of inputs; but the fitness of individual programs is assessed only with reference to a usually small, finite set of fitness cases. A program is said to be *robust* if it produces proper results for inputs that were not used in assessing fitness during the GP process.

The GP process starts by creating a random initial population of programs. The closure property ensures that each of these programs, unfit though it may be, will execute without signalling errors. Each of the programs is assessed for fitness, and fitness-sensitive *genetic operations* are then used to produce the subsequent generation. These may include reproduction, crossover, mutation, permutation, and others (Koza 1992); we use only reproduction and crossover here. The reproduction operator selects a highly fit individual and simply copies it into the next generation. Selection for re-

[1]The author acknowledges the support of the Dorothy and Jerome Lemelson National Program in Invention, Innovation, and Creativity.

[2]The description of genetic programming that follows covers only the simplest variant of the technique. See (Koza 1992) for more sophisticated variants.

production is random but biased toward highly fit programs. The crossover operation introduces variation by selecting two highly fit *parents* and by producing from them two *offspring*. The crossover operation selects random fragments of each of the two parents and swaps them; the resulting programs are copied to the next generation.

If GP is "working" on a given run then the average fitness of the population will tend to improve over subsequent generations, as will the fitness of the best-of-generation individual from each generation. After a preestablished number of generations, or after the fitness improves to some preestablished level, the best-of-run individual is designated as the result and is produced as the output from the GP system.

GP appears to be a powerful technique with wide applicability. It is CPU intensive, but there are ample opportunities for parallelism (e.g., in the assessment of fitness across a large population). We believe that it has great promise, but as Dewdney wrote of genetic algorithms more generally, "The jury is still out on a method that (a) claims to solve difficult problems and (b) is suspiciously painless." (Dewdney 1993, p. xiii) In order to understand the strengths and weaknesses of the technique we must apply it to areas in which prior research has mapped the computational territory. This strategy is being pursued by many, and is evident in (Koza 1992); in the remainder of this paper we endeavor to lay the foundations for such work in the mainstream of AI planning.

Genetic Planning

One can apply the techniques of GP to AI planning problems in a variety of ways.[3] GP systems produce programs; AI planning systems produce plans. Insofar as a plan is a program for an execution module, one can use a GP system *as* a planning system—one can use a GP system to evolve a plan which, when executed in the context of a given initial state, achieves a given set of goals.

A traditional AI planning system takes as input an initial state, a goal description, and a set of operator schemata, and produces as output a sequence of operator schemata, along with any necessary variable bindings. One can use a GP system in a similar way; given an initial state, a goal description, and a description of the actions that the execution module can perform, one can produce a program for the execution module that will achieve the goals from the initial state. The first of the experiments described below uses GP in this way.

The parallel between the traditional planning system and the genetic planning system need not be exact; whereas most planning systems require that the available actions be described declaratively (using, e.g., STRIPS operators (Fikes & Nilsson 1971)), purely procedural "operators" will suffice for the genetic planning system.[4] This is because the genetic planning system can assess the utility of action sequences by *running* them in simulation, rather than by analyzing declarative structures that describe operator effects. The cost of simulation can be high, both in runtime and in simulation development time, but the simulation approach obviates the need for declarative action representation. Since declarative action representation is an active research area with many outstanding problems (Ginsberg 1990), the availability of a planning methodology that does not require such representations is interesting for this reason alone. In addition, the way that simulation is used in GP is clearly parallelizable; the fitness of each program can be assessed in an independent simulation.

A more ambitious approach to genetic planning is to evolve control programs that can achieve some given set of goals from a variety of initial conditions. If one augments the function set to allow for decision-making and iteration in the evolved plans, one can actually evolve such "universal plans" (in the sense of (Schoppers 1987)). Koza's work on blocks-world planning takes this approach, as does the second of the experiments described below.

A third approach to genetic planning is to evolve complete domain-dependent planners. The function set must in this case include functions that access the system's current goals; given such a function set one can evolve programs that can achieve a range of goal conditions from a range of initial states. The third of the experiments described below uses GP in this way.

A fourth approach to genetic planning is to evolve complete domain-*independent* planners. The function set would in this case presumably include functions that have proven to be useful in existing domain independent planners; e.g., functions for constructing partial orders of plan steps. We have not yet conducted any experiments using this ambitious approach.

Koza's Genetic Blocks-World Planner

Koza has described the use of GP for a set of planning problems in a variant of blocks-world (Koza 1992, sec. 18.1). In this domain the goal is always to produce a single stack of blocks. The domain never contains more than one stack; every block is always either part of the stack or on the table (and clear). He considers the example of producing the 9-block stack that spells "**UNIVERSAL**" from a variety of initial configurations. Note that this is an instance of the second approach to genetic planning outlined above; we seek a single program that transforms a range of initial states to satisfy a single, prespecified goal condition.

Koza's blocks-world is unusual both because it is limited to a single stack of blocks and because it uses an unusually powerful set of functions and terminals (defined by (Nilsson 1989)). The terminal set consists of the following "sensors": **CS**, which dynamically specifies the top block of the stack; **TB** ("Top Correct Block"), which specifies the highest block on the stack such that it and all blocks below it are in the correct order; and **NN** ("Next Needed"), which

[3]Note, however, that although Holland's seminal work on genetic algorithms (Holland 1992) contains much of interest to planning researchers, its use of the phrase "genetic plan" has no relation to "planning" as used in the literature of AI planning systems.

[4]Some "traditional" planners use operators that include procedural components as well, e.g. NOAH (Sacerdoti 1975).

specifies the block that should be on top of **TB** in the final stack. The functions are: **MS** ("Move to the Stack"), which takes a block as its argument and, if it is on the table, moves it to the stack and returns **T** (otherwise it returns **NIL**); **MT** ("Move to the Table"), which takes a block as its argument and, if it is *anywhere* in the stack, moves the *top* block of the stack to the table and returns **T** (otherwise it returns **NIL**); **DU** ("Do Until"), which is actually a macro that implements a control structure—it takes two bodies of code, both of which are evaluated repeatedly until the second returns non-**NIL**; **NOT**, which is the normal LISP boolean negation function; and **EQ**, which is the normal LISP equality predicate.

Note that the function and terminal sets are carefully tailored to the specialized nature of the domain (O'Reilly & Oppacher 1992). **CS** would not generalize in any obvious way to a domain with multiple stacks. **TB**, though described as a "sensor," depends on the goal and must perform computation to match several elements in the world to components of the goal. Goal-sensitivity in the function and terminal sets is not necessarily to be avoided; indeed, in some cases it is necessary, and we use goal-sensitive functions below. But it is important to note that **TB** is goal-sensitive in a highly specialized, domain-dependent way. **TB** also depends on the fact that the domain can contain only one stack. **NN** is domain-specific in much the way that **TB** is. **MS** and **MT** make sense only in a single-stack world.

Koza ran his GP system on the "**UNIVERSAL**" problem for 51 generations with a population size of 500 individuals. He assessed fitness with respect to 166 of the millions of possible initial configurations. Fitness for an individual program was calculated as 166 minus the number of fitness cases for which the stack spelled "**UNIVERSAL**" after the program was run. A 100% correct program emerged in generation 10. It was: **(EQ (DU (MT CS) (NOT CS)) (DU (MS NN) (NOT NN)))**

Although this program is correct, it is not particularly efficient. It used 2,319 block movements to handle the 166 fitness cases, whereas it is possible to use only 1,641. By factoring the number of block movements into the fitness function Koza was able to produce a correct and maximally efficient program. That program, however, was longer than it needed to be. By factoring the number of symbols in the program into the fitness function (a "parsimony" measure) he was able to produce a correct, maximally efficient, and maximally parsimonious program.

Blocks-World Experiment #1

We have performed several experiments to assess the applicability of GP techniques to more traditional AI planning domains. The three that we describe here are all blocks-world experiments. Koza's GP code was used in all cases.[5]

Our first experiment was to use GP to produce a single correct plan that achieves a particular (conjunctive) goal condition from a particular initial state. We chose the problem known as the Sussman Anomaly as a representative problem from the blocks-world domain. The goal in this problem is to start with a world in which **C** is on **A**, and in which **A** and **B** are both on the table, and to produce a state in which **A** is on **B**, **B** is on **C**, and **C** is on the table. We will refer to the resulting state as an {**ABC**} tower.

We built a simple blocks-world simulation environment and wrote **NEWTOWER** and **PUTON** functions that are procedural versions of the following **STRIPS**-style operators. In these operators distinctly named variables must bind to distinct blocks:

Operator: (NEWTOWER ?X) ;; move X to the table if clear
Preconditions: (ON ?X ?Y) (CLEAR ?X)
Add List: ((ON ?X TABLE) (CLEAR ?Y))
Delete List: ((ON ?X ?Y))

Operator: (PUTON ?X ?Y) ;; put X on Y if both are clear
Preconditions: (ON ?X ?Z) (CLEAR ?X) (CLEAR ?Y)
Add List: ((ON ?X ?Y) (CLEAR ?Z))
Delete List: ((ON ?X ?Z) (CLEAR ?Y))

Our functions check that the required preconditions hold and change the world according to the add and delete lists if they do. Each function returns its first argument (the top of the resulting stack) upon success, or **NIL** if passed **NIL** or if the preconditions do not hold. We used a function set consisting of **NEWTOWER**, **PUTON**, and two sequence-building functions, **PROGN2** and **PROGN3**, which are versions of LISP's **PROGN** that take 2 and 3 arguments respectively. The resulting programs may have a hierarchical structure since the functions in the function set can be nested in many ways. The terminals used for this experiment were the names of the blocks: **A**, **B** and **C**.

We calculated fitness with respect to a single fitness case:

INITIAL: ((ON C A) (ON A TABLE) (ON B TABLE) (CLEAR C) (CLEAR B))
GOALS: ((ON A B) (ON B C) (ON C TABLE))

Our fitness function had three components: a correctness component, an efficiency component, and a parsimony component. The correctness component was calculated as 70 times the number of achieved goals divided by the total number of goals (in this case 3). This produces a number between 0 and 70, with higher numbers indicating better programs. The efficiency component was calculated from the number of **NEWTOWER** and **PUTON** actions actually executed in running the program. All executions were counted, even if the action was not successful. The number of actions was scaled to produce a number between 0 and 15, with higher numbers indicating more efficient programs. The parsimony component was calculated from the number of symbols in the program, scaled to produce a number between 0 and 15, with higher numbers indicating more parsimonious programs. The values of the correctness, efficiency, and parsimony clauses were summed and subtracted from 100, producing an overall fitness value between 0 and 100, with lower numbers indicating better programs.

Following a suggestion of Koza, we staged the introduction of the efficiency and parsimony components into the

[5]Koza's code can be found in the appendix to (Koza 1992), and can also be obtained by anonymous FTP.

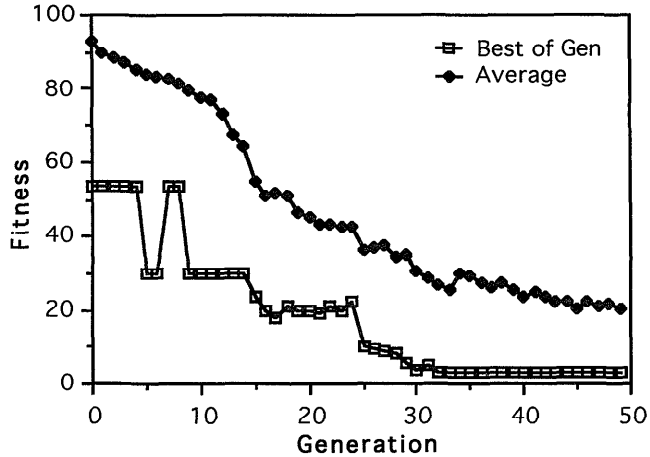

Figure 1. Best-of-generation and average fitnesses for exp. #1.

fitness function. In generations 0–14 only the correctness component of the fitness function was used. The efficiency component was introduced at generation 15 and was used thereafter. The parsimony component was introduced at generation 25 and was used thereafter.

We ran the GP system for 50 generations with a population size of 200. The overall performance of the GP system on this problem is summarized in Figure 1. In the initial generation of random programs the average fitness was 92.88. The best individual program of the population had a fitness measure of 53.33. It was: **(PUTON (PROGN2 C B) (NEWTOWER C))**. This gets **C** on the table and **B** on **C**, achieving 2 of the 3 goals. The average fitness of the population improved over the subsequent generations, but there was no improvement in the best-of-generation program until generation 5, when the following program was produced with a fitness measure of 30.0:

```
(PROGN3 (PROGN2 (NEWTOWER C)
                (NEWTOWER (NEWTOWER A)))
        (NEWTOWER (PROGN2 B B))
        (PROGN3 (PUTON B C) (PUTON B C) (PUTON A B)))
```

This program solves the Sussman Anomaly, but it is neither efficient nor elegant. The average fitness of the population continued to increase through the subsequent generations, although no improvement of best-of-generation individual was possible until generation 15, when the efficiency component of the fitness function became effective and allowed for differentiation among the correct plans. At generation 25 the parsimony clause became effective as well, and by generation 32 a maximally efficient, parsimonious, and correct plan had evolved with a fitness measure of 3.15: **(PROGN3 (NEWTOWER C) (PUTON B C) (PUTON A B))**.

Blocks-World Experiment #2

The best-of-run plan from experiment #1 solves the Sussman Anomaly, but it is not useful in many other cases. In our second experiment we wanted to evolve a "universal plan" for achieving a single goal condition from a range of initial states. To achieve greater generality we changed the terminal and function sets:

FUNCTION SET: (NEWTOWER PUTON PROGN2 PROGN3 TOPOVER DO-ON-GOALS)
TERMINAL SET: (TOP BOTTOM)

The **TOPOVER** function takes one argument, a block, and returns the top of the stack of which that block is a part. It returns its argument if it is something that is currently clear, or **NIL** if it is **NIL**. **DO-ON-GOALS** is actually a macro that implements a limited iteration control structure. It takes one argument, a body of code, that it evaluates once for each of the system's unachieved "**ON**" goals. During each iteration the variables **TOP** and **BOTTOM** are set to the appropriate components of the current goal. Note that we have removed **A**, **B** and **C** from the terminal set; programs can refer to blocks only via **TOP** and **BOTTOM**. **TOP** and **BOTTOM** are both **NIL** outside of any calls to **DO-ON-GOALS**, and calls to **DO-ON-GOALS** can be nested. The **DO-ON-GOALS** macro was developed for experiment #3, below, in which the need for access to the system's goals is more obvious.

We used 20 fitness cases and averaged their results; they were constructed from the following lists by pairing each initial state with each goal list:

INITIAL:
1. **((ON A TABLE)(ON B TABLE)(ON C TABLE) (CLEAR A)(CLEAR B)(CLEAR C))**
2. **((ON A B)(ON B C)(ON C TABLE)(CLEAR A))**
3. **((ON B C)(ON C A)(ON A TABLE)(CLEAR B))**
4. **((ON C A)(ON A B)(ON B TABLE)(CLEAR C))**
5. **((ON C A)(ON A TABLE)(ON B TABLE)(CLEAR C)(CLEAR B))**
6. **((ON A C)(ON C TABLE)(ON B TABLE)(CLEAR A)(CLEAR B))**
7. **((ON B C)(ON C TABLE)(ON A TABLE)(CLEAR B)(CLEAR A))**
8. **((ON C B)(ON B TABLE)(ON A TABLE)(CLEAR C)(CLEAR A))**
9. **((ON A B)(ON B TABLE)(ON C TABLE)(CLEAR A)(CLEAR C))**
10. **((ON B A)(ON A TABLE)(ON C TABLE)(CLEAR B)(CLEAR C))**

GOALS:
1. **((ON A B)(ON B C)(ON C TABLE))**
2. **((ON B C)(ON A B)(ON C TABLE))**

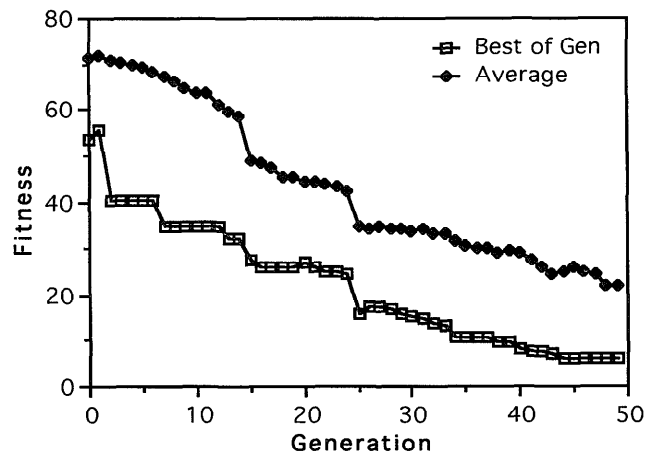

Figure 2. Best-of-generation and average fitnesses for exp. #2.

Note that the 10 fitness cases using goal list 2 are duplicates of the those using goal list 1 but with the order of the goal clauses changed; since **DO-ON-GOALS** loops through the goals in the order that they are presented, this helps to ensure that the resulting program is not overly dependent on goal ordering. All other GP parameters were set to the values used in experiment #1. The overall performance of the GP system in this experiment is summarized in Figure 2. In the initial generation of random programs the average fitness was 71.20. The best individual program of the population had a fitness measure of 53.33 and correctly handled 6 of the 20 fitness cases. It was:

```
(NEWTOWER (DO-ON-GOALS
            (PROGN3 (PUTON TOP BOTTOM)
                    (PUTON BOTTOM BOTTOM)
                    (DO-ON-GOALS TOP))))
```

The best-of-run individual program for this run was found on generation 48. It had a fitness measure of 5.91 and correctly handled all 20 fitness cases. It was:

```
(PROGN2
  (DO-ON-GOALS
    (DO-ON-GOALS
      (PROGN3 (NEWTOWER (DO-ON-GOALS
                          (TOP-OVER TOP)))
              (PROGN2 (TOP-OVER TOP) TOP)
              (PUTON TOP BOTTOM)))))
  (DO-ON-GOALS (PUTON TOP BOTTOM)))
```

Note that the program is robust over initial states that were not in the set of fitness cases. The program correctly builds an {**ABC**} tower from all three of the possible configurations that were not used as fitness cases: the towers {**CBA**}, {**BAC**}, and {**ACB**}. Because many problems are isomorphic, the use of function and terminal sets that refer to blocks only by their positions in goals, and not by their names, is helpful in achieving this robustness.

The robustness of the solution program does not extend to changes in goal sets. For example, the program will not achieve the unary goal list **((ON B A))** from an initial state consisting of a {**BCA**} tower.

Blocks-World Experiment #3

Our third experiment was an attempt to evolve a blocks-world planner capable of achieving a range of goal conditions from a range of initial conditions. We used the same terminal and function sets as in experiment #2. We increased the population size to 500 and the number of generations to 201, with efficiency introduced into the fitness function at generation 33 and parsimony introduced at generation 66. We used 40 fitness cases, constructed by pairing each of the initial states from experiment #2 with each of the following goal lists:

1. **((ON A B) (ON B C) (ON C TABLE))**
2. **((ON B C) (ON A B) (ON C TABLE))**
3. **((ON C B) (ON B TABLE))**
4. **((ON B A))**

All other GP parameters were set to the values used in experiment #1. The performance of the GP system in this experiment is summarized in Figure 3. In the initial generation of random programs the average fitness was 77.02. The best individual program of the population had a fitness of 59.17 and correctly handled 14 of the 40 fitness cases. It was:

```
(TOP-OVER
  (PROGN3 (PUTON (NEWTOWER (DO-ON-GOALS BOTTOM))
                 (DO-ON-GOALS (PUTON TOP BOTTOM)))
          (DO-ON-GOALS
            (TOP-OVER (DO-ON-GOALS BOTTOM)))
          (DO-ON-GOALS
            (NEWTOWER (PROGN2 BOTTOM BOTTOM)))))
```

The first 100% correct solution emerged at generation 25. It had a fitness of 30.0, contained 49 symbols, and was messy; we do not show it here. The efficiency and parsimony components of the fitness function, introduced at generations 33 and 66 respectively, helped to improve the programs considerably. The best-of-run individual program was found on generation 168 and had a fitness of 6.54. It was:

```
(PROGN3
  (TOP-OVER
    (DO-ON-GOALS
      (NEWTOWER (DO-ON-GOALS (TOP-OVER TOP)))))
  (DO-ON-GOALS (NEWTOWER (TOP-OVER BOTTOM)))
  (DO-ON-GOALS
    (DO-ON-GOALS (PROGN2 (NEWTOWER (TOP-OVER TOP))
                         (PUTON TOP BOTTOM)))))
```

The planner evolved in experiment #3 is considerably more robust than that evolved in experiment #2. In fact, although it was evolved with only 40 fitness cases, it correctly solves all 13x13=169 possible 3-block problems. It even solves some 4-block problems: for example, it will correctly produce both an {**ABCD**} tower and a {**DCBA**} tower from an initial state containing an {**ABC**} tower and the additional block **D** on the table. We have not yet fully analyzed the program's robustness for 4-block and larger problems.

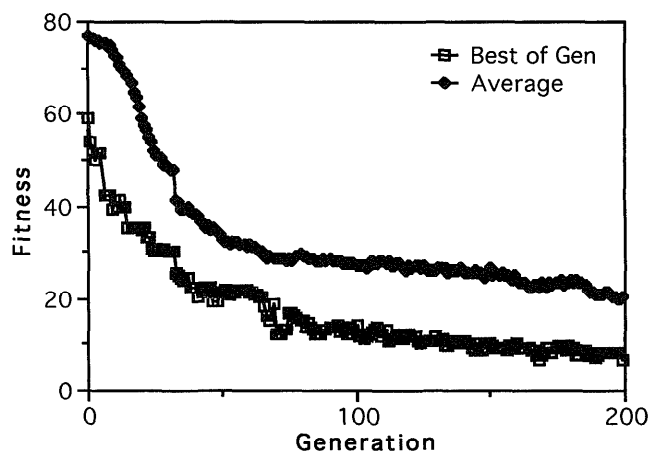

Figure 3. Best-of-generation and average fitnesses for exp. #3.

Discussion

In experiment #1 we wanted to see how well a GP engine could function in place of a traditional planner, which is generally invoked to produce a single plan that achieves a particular set of goals from a particular initial state. While we were able to evolve a correct, efficient, and parsimonious plan, one is lead to ask why a genetic technique should used in this case; traditional AI planning algorithms can solve such problems more reliably and more efficiently. Further, we should note that single blocks-world problems, at least with our fitness function (based on number of goals achieved), are not well suited to solution by genetic programming. This is because the coarseness of the fitness function provides little guidance to the evolutionary process. This can be seen in the first 10 generations of Figure 1, in which the coarseness of the fitness metric leads to large jumps in the best-of-generation fitness. We succeeded because the combinatorics of a 3-block world are manageable even with minimal guidance, especially with a program population size of 200. A more complex domain would demand a more informative fitness function. But GP may nonetheless be a good choice for solving some single-initial-state/single-goal planning problems. In particular, it can be appropriate when we have trouble representing the system's actions declaratively, or when the dynamics of the domain are best represented via simulation.

GP seems better suited, overall, to the construction of universal planners of the sort produced in our experiment #2, or complete domain-dependent planners of the sort produced in our experiment #3. There will always be problems in achieving robustness, however, and the genetic programming of universal planners is necessarily an iterative, experimental process. The success of a particular run of GP is highly sensitive to seemingly minor changes of parameters. Population size, crossover parameters, details of the terminal and function sets, choice of fitness cases, and variations in fitness metrics may all have large, difficult to predict effects. For example, we tried variations of blocks-world experiment #3 with identical parameters except for minor variations in the function and terminal sets (e.g., substituting a **DO-UNTIL** for **DO-ON-BLOCKS**, and providing other functions to access the goals). Many of these variations failed to produce fit programs. O'Reilly and Oppacher discuss the sensitivity of GP to this kind of variation and suggest modifications to the technique that they believe will lessen this sensitivity (O'Reilly & Oppacher 1992). But GP is an inherently experimental technique, and the resulting orientation may actually be quite welcome in some segments of the AI planning community; several planning researchers have recently called for just the sort of experimental framework that GP allows, and indeed requires (Hanks et al. 1993).

Genetic methods may also provide so-called "anytime" behavior (Dean & Body 1988), another feature of interest to the planning community: As can be seen in the fitness graphs in this paper, genetic programming starts by producing poor programs, and gradually improves the quality of its programs over time. The process can be stopped at any point to provide the current best-of-run program.

Conclusions

We conclude that GP has much to offer as an AI planning technology: freedom from declarative representation constraints, a methodology for building fast, domain-specific systems, a welcome experimental orientation, and anytime behavior during evolution. It also has many shortcomings: it is CPU-intensive, it is sensitive to minor changes in parameters, and it does not yet reliably produce robust results. Further work is clearly indicated.

Bibliography

Dean, T.; and Boddy, M. 1988. An Analysis of Time-Dependent Planning. In *Proceedings of the Sixth National Conference on Artificial Intelligence, AAAI-88*, 49–54.

Dewdney, A. K. 1993. *The New Turing Omnibus*. New York: W. H. Freeman and Company.

Fikes, R.E.; and Nilsson, N. 1971. STRIPS: A New Approach to the Application of Theorem Proving to Problem Solving. In *Artificial Intelligence* 2: 189–208.

Ginsberg, M.L. 1990. Computational Considerations in Reasoning about Action. In *Proceedings of the Workshop on Innovative Approaches to Planning, Scheduling and Control*, K. P. Sycara, ed. Defense Advanced Research Projects Agency (DARPA).

Hanks, S.; Pollack, M.E.; and Cohen, P.R. 1993. Benchmarks, Test Beds, Controlled Experimentation, and the Design of Agent Architectures. *AI Magazine* 14 (Winter): 17–42.

Holland, J.H. 1992. *Adaptation in Natural and Artificial Systems*. Cambridge, MA: The MIT Press.

Koza, J.R. 1992. *Genetic Programming*. Cambridge, MA: The MIT Press.

Nilsson, N. 1989. Action Networks. In *Proceedings from the Rochester Planning Workshop: From Formal Systems to Practical Systems*, J. Tenenberg, ed., Technical Report 284, Dept. of Computer Science, University of Rochester.

O'Reilly, U.; and Oppacher, F. 1992. An Experimental Perspective on Genetic Programming. In *Parallel Problem Solving from Nature, 2,* Männer, R.; and Manderick, B., eds. Amsterdam: Elsevier Science Publishers.

Sacerdoti, E.D. 1975. The Nonlinear Nature of Plans. In *Advance Papers of the Fourth International Joint Conference on Artificial Intelligence, IJCAI-75*, 206–214.

Schoppers, M.J. 1987. Universal Plans for Reactive Robots in Unpredictable Environments. In *Proceedings of the Tenth International Joint Conference on Artificial Intelligence, IJCAI-87*, 1039–1046.

Tate, A.; Hendler, J.; and Drummond, M. 1990. A Review of AI Planning Techniques. In *Readings in Planning*, Allen, J.; Hendler, J.; and Tate, A., eds., 26–49. San Mateo, California: Morgan Kaufmann Publishers, Inc.