The Automatic Programming of Agents that Learn Mental Models and Create Simple Plans of Action

David Andre
Visiting Scholar, Computer Science Department, Stanford University
860 Live Oak Ave, Apt 4, Menlo Park, CA 94025
andre@flamingo.stanford.edu

Abstract

An essential component of an intelligent agent is the ability to notice, encode, store, and utilize information about its environment. Traditional approaches to program induction have focused on evolving functional or reactive programs. This paper presents MAPMAKER, a method for the automatic generation of agents that discover information about their environment, encode this information for later use, and create simple plans utilizing the stored mental models. In this method, agents are multi-part computer programs that communicate through a shared memory. Both the programs and the representation scheme are evolved using genetic programming. An illustrative problem of 'gold' collection is used to demonstrate the method in which one part of a program makes a map of the world and stores it in memory, and the other part uses this map to find the gold The results indicate that the method can evolve programs that store simple representations of their environments and use these representations to produce simple plans.

1 Introduction

The ability to notice, encode, store, and utilize information about the environment is an essential component of intelligent behavior. Storing a model or map of the environment increases the problem solving capacity of an intelligent agent. However, much of the research on the artificial induction of computer programs has focused on reactive programs with no use or only a minimal use of state. These programs, although generated by an artificial process, do not themselves learn or produce plans of action.

Genetic programming (Koza 1992) is a variant of the genetic algorithm in which the genetic population consists of computer programs rather than of fixed length bitstrings or other fixed data structures. The initial population of programs consists of randomly generated programs represented as parse trees that are composed of the available simple programmatic ingredients. Genetic programming then breeds these programs using the Darwinian principle of survival of the fittest and the crossover operation, which is similar to sexual recombination in nature. Tackett (1994) provides analysis that genetic programming can be viewed as a method of stochastic beam search.

This paper presents MAPMAKER, a method for the automated generation of computer programs that discover information about their environment, encode this information, store it, and then utilize this information to produce plans of action. These programs are evolved using genetic programming and the structures of the evolving programs are constrained so as to facilitate the development of learning and the use of memory. The method evolves solutions to the gold collection problem, where the agent must learn the positions of gold in each of several worlds, store this information in a usable fashion, and then later utilize this information to produce simple plans to collect the gold. Several evolved solutions to the problem are discussed that generalize perfectly to worlds on which they have not been trained. In addition, the mental models created by these successful individuals are understandable and clearly represent models of the world. Also, evidence is presented that random search could not find even partial solutions to the problem in any reasonable time.

2 Background on Genetic Programming

As described in John Koza's seminal work (1992), genetic programming is a method that breeds populations of computer programs (such as those shown in Fig. 1). The genetic programming performed in this research employs steady-state selection (Syswerda 1989), a minor variant on Koza's methods (Koza 1992). Genetic programming with steady-state selection consists of the following steps:

(1) Create an initial population by randomly generating programs composed of the primitive functions.
(2) Execute each program in the population and determine its fitness based on its ability to solve the problem.
(3) Loop over the following until either a complete solution or a satisfactory result is found, or a limit on the number of reproductions is exceeded.
    (a) Create two new offspring by applying the crossover operation. Crossover creates two new programs by swapping randomly chosen subtrees of two existing programs (the parents - Fig. 1). The parents are selected probabilistically based on high fitness. Programs with high fitness will be selected often, programs with low fitness will seldom be selected.
    (b) Kill two members of the population to provide space for the two new children. Choose these individuals probabilistically based on poor fitness.
    (c) Evaluate both children and determine their fitness.

Although the programs expressed in Fig. 1 are simple, genetic programming can evolve much more complex programs, utilizing complicated programmatic structures. Genetic programming can evolve programs utilizing iteration and subroutine calls, as discussed in Koza (1994). Automatically defined functions (ADFs) are subroutines that are co-evolved with the main program, and can increase the power of genetic programming (Koza 1994). For more information on genetic programming, see Kinnear (1994), which reviews advances in genetic programming.

3 Related Work

The acquisition of mental models and the automatic synthesis of agents that learn are not new areas for the field of evolutionary computation. Neural network learning methods have often been combined with genetic algorithms that specify the layout of the network and/or the initial weights (Belew et. al. 1991; Ackley and Littman 1991). However, these methods do not use explicit representations of state - the 'memories' learned are stored in the weights and are thus closed both to introspection and to human understanding. Additionally, the role that the neural net plays in the individual is often pre-specified, and thus the learned representations can only be used in limited ways. In genetic programming, the use of branching operators that depend upon the state of the environment is common (Kinnear 1994). When an evolved program combines several actions and branching statements through the use of progn statements it incorporates an implicit use of state. However, this state represents at best only an implicit representation of the world, is not available for introspection, and often can be difficult to comprehend. Occasionally, GP applications allow the evolving programs to use a few variables of state (Andre 1994b; Koza 1994), but this is hardly representational memory.

One successful use of evolved representational structures is Teller's (1994a) on work on using indexed memory in genetic programming. Teller evolved programs that could solve a simple problem - that of pushing blocks up against the boundaries of a world. Teller used an interesting strategy to facilitate the use of memory in his evolving programs; he strictly limited the function sets so that the evolved programs could move only once per evaluation and receive only limited sensory feedback. Without using memory, only minimal fitness was possible. In addition, Teller has proved that his indexed memory paradigm is Turing complete (Teller 1994b). Although it is valuable work, Teller's indexed memory scheme poses several flaws for the study of the evolution of agents utilizing mental models. First, the evolved representations developed by his programs are difficult to interpret. Second, Teller's representation allows individuals to perform well using only indirect models of their world, such as simple counters.

In some preliminary work with the MAPMAKER method, we demonstrated that agents could evolve to solve the gold collection problem when programs had access to only one memory cell for each world location (Andre 1994a). This previous work used the same multi-module MAPMAKER architecture used in the current work, but the evolved agents utilized a two-dimensional memory that simplified the computation. The function set was too complex: movement in memory was hard-coded to movement in the world to reduce the demands on evolution. The evolving representations were constrained so that they exactly matched the structure of the world.

The present research addresses these issues by extending Teller's (1994a) indexed memory scheme. Indexed memory allows for a wide variety of representations. In addition, because the system uses a multi-module architecture for the evolving programs and a multi-phasic fitness environment in which the input and output processes of the individual programs are kept separate, the evolved representations of the world are easily available and comprehensible.

4 The MAPMAKER Architecture

One problem inherent in investigating the use of memory and internal representation in program induction is that many problems can be solved without state. Solutions using memory may be less complex than those not using memory, but may be harder to evolve. Genetic programming is known for exploiting loopholes, and thus to evolve the use of memory, one must constrain the fitness environment in order to promote its evolution. Teller (1994a) required the use of memory by restricting the building blocks and the sensory inputs. The MAPMAKER architecture, (Fig. 2a), depends on multi-modularity and sensory deprivation to force the evolution of memory. Each individual consists of two modules, each of which is executed separately with different inputs and outputs. The first module, the map-maker, can examine sensory information and may store and read information in memory, but may not act in the world. The second module, the map-user, is blind with respect to the world; it must use only its stored representation to produce a plan. The plan is then evaluated and its fitness determined. Assuming that the task requires specific knowledge about the current world, memory representations of the world are required to achieve good fitness, because of the separation of perception and action into the two modules.

The MAPMAKER architecture facilitates in understanding the representations used by the evolving agents because it provides an opportunity to examine the exact internal state that represents the world: the information contained in memory at the moment after the map-maker is executed is all the information the map-user may use in creating a plan, and thus represents the entire representation of the world. Another aspect of the MAPMAKER approach is that it parallels the stages often used in psychological studies of memory - stimulus, delay, and retrieval.

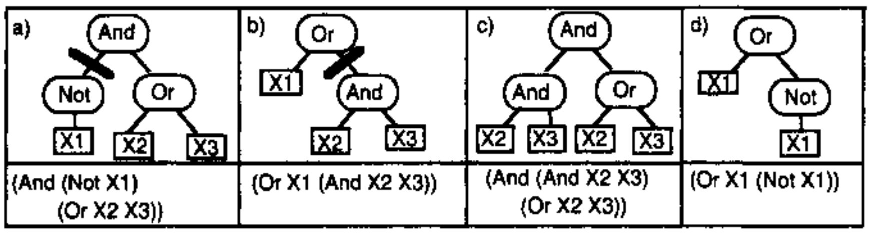


Figure 1. An example of crossover with programs for a Boolean regression problem. The programs in parts a and b are the parents for the crossover operation. The crossover operation exchanges a subtree from one parent with a subtree from the other parent to create the two offspring (parts c and d). The heavy lines indicate the subtrees that were exchanged.

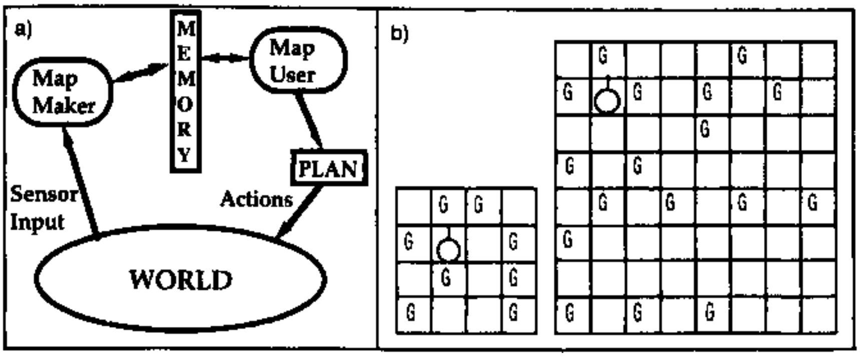


Figure 2 a) The MAPMAKER architecture, b) Toroidal worlds for the gold collection problem

In this architecture, the fitnesses for both modules of the individual are determined only by the output of the map-user. This is an extension of the credit assignment problem inherent in program induction - which part of the program caused the high or low fitness? Although this sort of indirect fitness is normally encountered when multi-part programs are evolved, it is especially salient in this approach because the two modules are executed separately, and the behavior of the map-maker only affects fitness if the map-user utilizes information from memory.

5 The Gold Collection Problem

The goal in the gold collection problem is to dig up all the gold, without digging on squares that have no gold. The agent operates in an *NxN* toroidal world (Fig. 2b). The squares in the world can be identified by the vector of integers modulo (7V-1) of the form (i,j) where $0 \le i,j < N$. These can be stored and manipulated as a single integer when N < 10. Thus, 11 is the square (1,1), and 43 is the square (4,3). This representation was used to allow the programs access to their position and still allow the functions to always return integers. The agent initially starts off in square 11. An individual interacts with the world as follows (Fig. 2a) . First the map-maker module of the individual is iteratively allowed to observe the positions of each gold in the world, one gold at a time. This module can store this information in memory. Then, the map-user module is allowed to examine the contents of memory, and to output actions into the plan This plan is then executed, and the numbers of golds collected and erroneous digs performed are calculated and used to determine the fitness of the individual. Each evolving individual is evaluated on several worlds with different arrangements of gold to promote solutions that will generalize well to worlds not used in evolution.

Though many world sizes and gold densities are possible, we used world sizes of 2x2, 4x4, and 8x8. The number of gold in the worlds is approximately half the number of squares in the world, with the exception that in the 8x8 world, gold densities of 1, 2, and 3 golds per world were also investigated. Although it may seem that these world sizes are trivial to map, such is not the case. In order to succeed, the map-maker and the map-user must evolve to agree on a representation. The map-maker must evolve to store all gold positions in memory in some fashion, and the map-user must evolve to use this information to constrain and guide its digging and movement actions. Although in the 2x2 world with 1 gold there are only 4 possible configurations of gold, there are infinitely many ways to encode this information and the map-user and the map-maker must evolve a shared representation. Evolving a map-user to select what the plan for gold collection should be is not trivial because there are infinitely many plans that attempt to dig only once, infinitely many that dig N times on the same square, etc. The map-user must also evolve the code to move around the world to the squares with gold. If the map-user takes the simplest option and avoids path planning by visiting every square, then it must solve Koza's Lawnmower problem (Koza 1994). Koza found this problem to be difficult when no ADFs were used, and straightforward, although not trivial, when ADFs were used. Solutions are thus not easily generated, even for the tiniest worlds. The 2x2 world is also similar to Square World, a problem in the AI literature used to illustrate planning strategies (Genesereth and Nourbakhsh 1993).

6 Algorithmic Details

There are four steps in preparing to use genetic programming on any problem, namely specifying the architecture of the programs to be evolved, the set of primitive programmatic ingredients, the fitness function, and the parameters for controlling the run.

The architecture of the evolving programs is shown in Fig. 3. The map-maker module has an easier task than the map-user; because it is directly shown the positions of the gold, it must only store the gold locations in some fashion that the map-user can comprehend. Therefore, ADFs (subroutines) are not needed in the map-maker, but are in the map-user. The ADFs in the map-user are arranged hierarchically; the Result Producing Branch for the map-user (RPBU) can call both of the ADFs, and ADFU2 can call ADFU1. Both ADFs take a single argument.

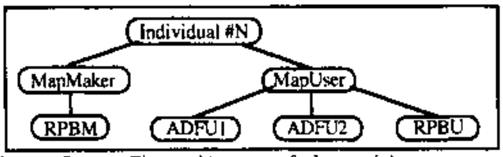


Figure 3. The architecture of the evolving programs: consisting of a map-maker module with one result-producing branch and a map-user module with two automatically defined functions and one result-producing branch.

World Size	Num Gold	#Fit Cases	Population Size	Max Num Generation Equivalents
2x2	2	4	14,000	130
4x4	8	15	18,000	110
8x8	1	30	10,000	200
8x8	2	30	12,000	165
8x8	3	30	12,000	165
8x8	20	30	15,600	130

Table 2. Parameters for several of the runs.

The set of primitive programmatic ingredients must be specified prior to a run of genetic programming. These ingredients are the basic building blocks for the programs to be evolved. For the gold collection problem, we choose the functions shown in Table 1.

Each module and ADF of the program has a slightly different function set to account for the varied behaviors allowed in each. The map-maker has no access to functions for motion, for example, but has access to view the world through the GoldPos function. Additionally, some functions are constrained to occur only in the ADFs to engender their widespread, consistent use. Previous work (Andre 1994a) indicated that runs that had no such restrictions were largely unsuccessful. Restricting key functions to an ADF forces that ADF to be used when the behavior produced by those functions is needed. When the ADF is changed, the modifications occur in all places where the functionality is used.

The function sets for each of the different branches for the gold collection problem are shown in Table 1 0, 01,10, and 11 are constants chosen for their usefulness in this problem because, for example, 10 represents the vector (1,0) which is one step in the x direction. R refers to the ephemeral random α-$10 \leq \mathcal{R} \leq 10$. Ephemeral random constants provide a method for creating constants in GP (Koza 1992). ADFU1 and ADFU2 are one-argument functions that call the appropriate ADF (subroutine). Arg_l is a zero-argument function that contains the dummy variable (formal parameter) passed to an ADF.

The add and sub functions act on a pair of integer vectors. These operations use digit by digit arithmetic modulo the size of the world. For example, in a 4x4 world, (add 33 12) = 01. Not returns a 1 if its argument evaluates to 0, otherwise it returns a 0. Progn evaluates both its arguments and returns the value of the second argument. Self returns the individual's current location in the world. Dignow outputs a 'Dig' command to the plan. Jump (a) outputs a command to the plan that changes the agent's position by a displacement of a in the world. For example, if an individual were at position 1,1 in the world, (Jump 12) would move the individual to position (2,3) in the world.

GoldPos(), used by the map-maker, returns the position of the gold that the map-maker is currently allowed to view. Repeat(a,b) takes the result of its first argument a, modulo the size of the world, and iteratively executes the second argument b that many times. Repeat returns the value of the last execution of the second argument b. Repeati(a,b) is much like Repeat, excepting that Repeat_Index () is set within Repeat 1. The Repeat_Index() is equal to the current iteration number. PutMexn(a,b) puts b into memory cell a and returns the previous value of cell a. ReadMem(a) returns the value of cell a. IncMem(a) increases cell a by 1.

The fitness function in the gold collection problem is straightforward. The goal is to collect all of the gold in each of several worlds without making any incorrect digs. Thus, to attain a perfect score, the individual acting in the 2x2 world must collect all 4 golds - one from each of 4 worlds - without any false-digs, and the individual acting in the 8x8 world with 20 golds per world must collect 600 golds - 20 from each of 30 worlds. The golds were distributed randomly in each world. The fitness of an individual is a weighted sum of the number of golds *not* picked up and the number of false digs. In addition, there is a large penalty for picking up no golds. Thus the fitness function is equal to the following expression, where lower fitness is better.

5*Gold_Remaining+2 0*Num_of_FalseDigs+ (10,000 if Gold=0).

The next step in preparing to run genetic programming is to choose values for various parameters of the run. Tournament selection (Goldberg and Deb 1991) with a tournament size of 8 was used to choose parents for crossover. To choose parents to be removed from the population to make room for the newly created children, tournament selection with a tournament of size 2 was used. Larger tournament sizes for the removal operation result in overly greedy evolution. The population size and maximum number of generation equivalents for the presented research are shown in Table 2. One generation equivalent is defined as the number of reproductions necessary to create as many children as there are individuals in the population.

Module	Branch	Functions allowed
MapMaker:	RPBM	0, 01, 10, 11, $\mathcal{R}$, add, sub, not, GoldPos, IfPairEqual, progn, PutMem, ReadMem, IncMem
MapUser:	ADFU1	0, 01, 10, 11, $\mathcal{R}$, add, sub, not, SELF, IfPairEqual, arg_1, progn, Jump, PutMem, ReadMem, IncMem, DIG
	ADFU2	0, 01, 10, 11, $\mathcal{R}$, ADFU1, add, sub, not, SELF, IfPairEqual, arg_1, progn, Jump, PutMem, ReadMem, IncMem
	RPBU	0, 01, 10, 11, $\mathcal{R}$, ADFU1, ADFU2, add, sub, not, progn, Repeat, Repeati, Repeat_Index, IfPairEqual
Argument Structure:		0(), 01(), 10(), 11(), $\mathcal{R}$(), add(a,b), sub(a,b), not(a), Repeat(a,b), Repeati(a,b), arg_1(), progn(a,b), SELF(), dignow(), IfPairEqual(a,b,c,d), Jump(a), ADFU1(a), ADFU2(a), GoldPos(),Repeat_Index(), PutMem(a,b), ReadMem(a), IncMem(a)

Table 1. Function sets for the branches of the evolving programs, and the argument structure for each of the functions.

7 Results

The runs were performed on a variety of machines, including Sun Sparc 2's, a DEC Alpha, and 486-66 machines. Runs took an average of 2 days to complete. In all world sizes, approximately 1/3 of the runs produced solutions. On the non-successful runs, nearly correct individuals emerged that collected most of the gold and dug on very few squares with no gold. Successful results for each world size will be briefly discussed. On the 2x2 world, four solutions emerged out of ten runs. One such solution emerged after processing 1,100,742 individuals. The behavior of this individual was straightforward. The map-maker would fill memory elements directly corresponding to locations in the world with the value 1, and then would reset these cells to 0 if a gold was found there. Additionally, the symbol 13 was also used to signify a non-gold. Memory traces from four worlds after the map-maker has been executed are shown in Table 3. When golds were at (0,0) and (0,1), for example, memory cells 0 and 1 are set to 0, whereas cells 10 and 11 are set to 1. The map-maker also stores some redundant information that is not used by the map-user. For example, whenever there is a gold in position (1,0), memory cell 19 is filled with a 12. It seems that the process of development might be that the map-maker first evolves to store information about the world in many redundant ways, and then the map-user learns enough of one of these representations to be able to achieve a better fitness.

On other successful runs on the 2x2 world, representation schemes were similar, although different symbols were used. In one scheme, the map-maker stored the representation for locations in the memory cell corresponding to the square one square to the left of the location. This offset was learned by both the map-maker and the map-user, and changed the notion of what correspondence to the world meant.

One of the 100% correct solutions to the 4x4 world evolved after 487,068 individuals had been processed. The program code for this individual is shown below:

```
MapMaker RPB: (not (ReadMem (ReadMem (PutMem
(GoldPos) (PutMem (GoldPos) (IncMem (progn
(progn 30 1) (GoldPos))))))))
MapUser RPB: (ADFU1 (ADFU2 [Repeati (Repeat
(sub (add 10 (g_repeat_index)) (ADFU1
(Repeati (not 11) (Repeati (Repeati (ifpaireq
(ADFU2 (progn 0 1)) (ADFU1 (progn
[g_repeat_index) 11)) 1 (progn (ifpaireq (Repeat
(g_repeat_index)(add 1 (ADFU1 1))) 1 1 (ADFU1
(ADFU2 (sub 1 10))))) (Repeati (ifpaireq (ADFU2
(Repeati (ADFU1 (ADFU2 (Repeati (ifpaireq
(Repeat (g_repeat_index) (Repeati 1 11))
(g_repeat_index) 1 (ADFU1 (ADFU2 (Repeati
(ADFU1 (ADFU2 (Repeati(ifpaireq (Repeat
(g_repeat_index) (ADFU1 (ADFU2 (Repeati
(ifpaireq (Repeat (Repeati 1 1))
(g_repeat_index) 1 (add 1 (ADFU1 (ADFU2 (Repeati
(ifpaireq (Repeat (g_repeat_index) (ifpaireq
(Repeat (g_repeat_index) (Repeati 1 1))
(g_repeat_index) 1 (add 1 (ADFU1 (ADFU2
(Repeati (ifpaireq (Repeat (g_repeat_index)
(ifpaireq (Repeat (g_repeat_index) (Repeati 1
1)) (g_repeat_index) 1 (add 1 (ADFU1 (ADFU2
(Repeati (add (not g_repeat_index)) (sub 10
(g_repeat_index)) 1)))))) (g_repeat_index) 1
(add 1 (ADFU1 (g_repeat_index)))) 1)))))))
(g_repeat_index) 1 (add 1 (ADFU1
(g_repeat_index)))) 1))))) 1)))) 0 1 (add 1 (add
1 (ADFU1 (g_repeat_index))))) 1))) 1)))) 1)))
1)) (g_repeat_index) 1 (ADFU1 (ADFU2 (Repeati
(ifpaireq (Repeat (g_repeat_index) (Repeati 1
11)) 0 1 (Repeati (ifpaireq 10 (g_repeat_index)
1 (add 1 (ADFU1 (g_repeat_index)))) 1)) 1))))
1))) 10) 11))))) (ADFU1 (ADFU2 (Repeati (Repeat
(g_repeat_index) (ifpaireq 1 1 0 22)) 1)))) 1)))
MapUser ADF #0: (PutMem 11 (ifpaireq (ifpaireq
(PutMem (SELF) 11) 1 (Dig) 21) 1 (Dig) 21))
MapUser ADF #1: (Jump (Jump (Jump (Jump (Jump
(sub (Jump (ifpaireq (sub 11 1) 1 (progn 12
(Arg_1)) 11)) 11))))))
```

Although this individual looks complicated, its behavior is actually quite simple to understand, because of the MAPMAKER architecture. Not only can the behavior be largely understood from the output plan and the state of memory between the execution of the modules, the code can also be understood through analysis. This individual stores a 1 in the memory cells corresponding to gold positions in the world, a scheme similar to that shown in the 2x2 solution. However, the map-user in this world shows some other interesting uses of memory. The map-user in this case learned a single route over the 4x4 world and checked its memory at each square to see if it should dig. However, the route it learned overlapped, so it would pass over some squares more than once. To avoid digging on squares that had already been dug, the map-user stored a 11 in memory at each cell after having been there. In addition, it stored a '21' in the memory cell corresponding to the initial square.

The learned world representation not only allowed communication between map-maker and map-user, but allowed the map-user to use memory to avoid redigging at any locations. This individual's solution to the 4x4 world with 8 golds was completely general as well; although only trained on 15 different combinations of 8 golds per world, the individual collects without error any and all gold that is given to it in the 4x4 world. The ability of this individual to collect gold on all 4x4 worlds is shown in two ways: 1) it was tested experimentally on 100 worlds not included in the original evolution, and 2) is also a provable consequence of the code.

Successful individuals emerged at all four gold concentrations examined in the 8x8 world. When there was only one gold per world, one evolved individual - found after processing 840,296 individuals - utilized a quite different representation scheme than did the solutions to the 2x2 and 4x4 worlds. The individual stored the position of the gold in a hard coded memory location, cell 0, and then used this information to choose where to dig.

	Gold 1 Pos	Gold2 Pos	Memory: Cells:	0	1	2	3	4-8	9	10	11	12	13	14-18.	19	20	21
World 0	(0,0)	(0,1)		0	0	1	2	0	2	1	1	0	0	0	0	0	0
World 1	(0,1)	(1,1)		1	0	1	2	0	0	13	0	11	11	0	0	0	0
World 2	(1,0)	(0,0)		0	1	0	0	0	2	0	10	10	0	0	12	1	0
World 3	(0,1)	(1,0)		1	0	1	2	0	0	0	10	10	0	0	12	0	1

Table 3. *Memory traces from a successful individual on the 2x2 world. The bold cells are those that correspond directly to positions in the world. The memory cells contain the values that were in memory after the map-maker had finished execution.*

The actual code that evolved contained a great deal of code that was never executed, and the actual executed code is quite simple to understand; such a simplified version is shown in Fig. 4. The individual stores the correct gold position in cell 0, calculates the displacement from the starting position to the gold's location, jumps to that square, and digs.Thus, the individual evolved to be a planner that produces optimal plans for a single gold. Although the output behavior is simple, evolving such a behavior in the space of all possible plans is not.In addition, this individual was only trained on 30 worlds, with a gold at a different space in each world. This evolved individual generalized correctly to all 64 possible 8x8 worlds with 1 gold.

Successful individuals evolved for the 8x8 worlds with 2, 3, and 20 golds that used memory schemes similar to those used by the solutions to the 2x2 worlds. Andre (1994c) contains more detailed analysis of these results.. One successful individual on the 8x8 worlds with 20 golds evolved to use a memory scheme similar to that used by the solution discussed for the 4x4 worlds.It uses a '1' to represent a gold, and a '0' to represent a non-gold.After digging each gold, the corresponding memory cell is reset to a '0', so no multiple digs occur. The individual is provably a correct solution for all 8x8 worlds, with any gold distribution, even though it had been trained on only 30 worlds with 20 golds in each.

Overall, the experiments indicate that the MAPMAKER method can successfully evolve individuals to solve the gold collection problem. Many of the solutions were completely general - they extended perfectly to any number and arrangements of gold in their world, even though they were trained on a small subset of the possible worlds. In the case of the 8x8 world with 1 gold per world, an individual evolved that produced optimal plans of action for obtaining the gold. Many different representation systems emerged; these are shown in Table 4. All of these representation schemes are evolved models of the world; importantly, these models differ from the models evolved in the previous work with the MAPMAKER architecture (Andre 1994a) in that the exact mapping between memory and the world is evolved, rather than forced by the function set. Many of the solutions involve a direct mapping, but several utilize offsets, translations, or exchanges of cells that are more complex. In addition, preliminary research indicates that general solutions will evolve even if the direct mapping is prohibited by limited memory damage (Andre 1995).

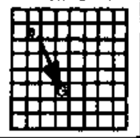
Thus, MAPMAKER can evolve programs to control a simple agent that uses indexed memory to store information and then uses this information to create simple plans.

8 Comparison to Random Search

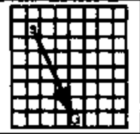
Many researchers, when first encountering genetic algorithms, often wonder if the results are due to a modified version of random search. Although Koza (1992) has shown that genetic programming performs much better than random search for many problems, it is important to rule out random search as a possibility for MAPMAKER's success.

To test this possibility, twenty million random individuals that followed the MAPMAKER architecture were created and evaluated on the 8x8 worlds with 20 golds per world. This version of the gold collection problem was chosen because it required the least computational effort to solve for MAPMAKER using genetic programming, and was thus felt to be the easiest of the problems (see Andre 1994c for more explanation). The best of these twenty million randomly generated individuals only collected 56 golds out of 600, and incurred 3 penalties for digging on squares with no golds. The individual did store some information about the world; an 11 was stored in some but not all squares with gold. . Although the map-user looked at several memory locations per world, the map-user used a different 'language' than the map-maker as to when it expected a gold, and thus , failed to correctly plan its actions even for the golds that were stored in its memory.

In comparison to the twenty million individuals that were evaluated for random search, which found only a 9% correct solution, MAPMAKER required processing fewer than six hundred thousand individuals to find a 100% correct solution on the 8x8 worlds with 20 golds per world. Random search thus seems unable to discover, in any reasonable amount of time, the cooperation between the map-maker and the map-user, the development of a shared representational system, and the programs to control both modules that are all required to solve the gold collection problem.



Memory Cell #0 : 35
Output Plan:
```
(Jump 24), Dig
```



Memory Cell #0 : 47
Output Plan:
```
(Jump 36), Dig
```

Simplified Code:

MapMaker:
```
(PutMem 0 (GoldPos))
```
MapUser:
```
(ADFU2 1)
```
ADFU #1:
```
(Dig)
```
ADFU #2:
```
(ADFU1 (Jump (sub (ReadMem 0) 11)))
```

Figure 4. Behavior and simplified code for successful individual on the 8x8 world with 1 gold.

World Size	Gold Per World	Representation Method	Symbols
2x2	2	Store symbol in Mem[Location]	No Gold = '1' or '13', Gold = '0'
2x2	2	Store symbol in Mem [Location-to-Left-of-Current]	Gold = '1'
4x4	8	Store symbol in Mem[Location]	Gold = '1', No Gold = '0', BeenHere = '11', Starting-Location = '21'.
8x8	1	Store Location in Mem[0]	Gold's position is in Mem[0], No Gold = '0'
8x8	2,3,20	Store symbol in Mem[Location]	Gold = '1', No Gold = '0'
8x8	20	Store symbol in Mem[Location-1]	Gold = '14', No Gold = '0' or '4'

Table 4. *Several representations that emerged*

9 Conclusions

This paper has presented MAPMAKER, a method for the automatic generation of programs that observe their environment, store information, and then later use this information to formulate simple plans of action. The experiments with the gold collection problem indicated that MAPMAKER with genetic programming could generate solutions that were generalizable, robust, and in some cases optimal solutions to the problem. Future work will examine more complex representational schemes, more difficult problems, the possibility of memory structure that itself evolves, and cooperation among multiple agents.

The gold collection problem is, of course, a toy problem, and the solutions generated by MAPMAKER are not exemplary in and of themselves. However, what is noteworthy is that the solutions were created automatically through the artificial evolution of programs following the MAPMAKER architecture. The MAPMAKER architecture was successful at generating agents that discover information about their environment, encode this information for later use, and create simple plans using the stored mental models.

Acknowledgments

I would like to thank John Koza and Nils Nilsson for their help, advice, and many comments regarding this work.

References

Ackley, D. and Littman M. (1991). Interactions between learning and evolution. In Artificial Life II, SFI Studies in the Sciences of Complexity, vol. X, ed. C.G. Langton, C.Taylor, J.D. Farmer, & S. Rasmussen, Addison-Wesley.

Andre, D., (1994a). Evolution of Mapmaking: Learning, planning, and memory using genetic programming. Proceedings of the 1994 IEEE World Congress on Computational Intelligence. IEEE Press.

Andre, D., (1994b). Learning and upgrading rules for an OCR system using Genetic Programming. Proceedings of the 1994 IEEE World Congress on Computational Intelligence. IEEE Press.

Andre, D., (1994c). Artificial Evolution of Intelligence: Lessons from natural evolution: An illustrative approach using Genetic Programming. Unpublished BS Honors Thesis. Stanford University. Symbolic Systems Program.

Andre, D., (1995). The Evolution of Agents that Build Mental Models and Create Simple Plans Using Genetic Programming. To appear in Proceedings of the 1995 International Conference on Genetic Algorithms. Morgan Kaufmann.

Belew, R.K., Mclnemey, J., and Schraudolph, N.N. (1991). Evolving Networks: Using the genetic algorithm with connectionist learning. In Artificial Life II, SFI Studies in the Sciences of Complexity, vol. X, ed. C.G. Langton, C.Taylor, J.D. Farmer, & S. Rasmussen, Addison-Wesley.

Genesereth, M. and Nourbakhsh, I.. (1993). Time-saving tips for problem solving with incomplete information. In Proceedings of the 11th National Conference on Artificial Intelligence, AAAI-Press.

Goldberg, D.E., and Deb, K. (1991). A comparative analysis of selection schemes used in genetic algorithms. In Rawlins, G. (editor), Foundations of Genetic Algorithms. Morgan Kaufmann.

Kinnear, K. E. Jr. (ed). (1994). Advances in Genetic Programming. The MIT Press.

Koza, J.R., (1992). Genetic Programming: on the programming of computers by means of natural selection, Cambridge, Mass: MIT Press.

Koza, J.R., (1994) Genetic Programming II: Automatic Discovery of Reusable Programs . Cambridge, MA: The MIT Press.

Syswerda, G. (1989). Uniform crossover in genetic algorithms. Proceedings of the Third International Conference on Genetic Algorithms. (J. Schaffer, Ed,) San Mateo, CA: Morgan Kaufmann.

Tackett, W.A. (1994). Recombination, Selection, and the Genetic Construction of Computer Programs. Ph.D. dissertation, University of Southern California, Department of Electrical Engineering Systems.

Teller, A. (1994a) Turing Completeness in the language of genetic programming with indexed memory. Proceedings of the 1994 IEEE World Congress on Computational Intelligence. IEEE Press.

Teller, A. (1994b). The Evolution of Mental Models. Advances in Genetic Programming. (Kim Kinnear, Ed.). Cambridge, MA: MIT Press.