Proceedings of the 1995 ACM/SIGAPP Symposium on Applied Computing, February 26-28, 1995, Nashville, TN, pp. 319-323, ACM Press.

A SIMULATION OF ADAPTIVE AGENTS IN A HOSTILE ENVIRONMENT*

Thomas D. Haynes and **Roger L. Wainwright**

Department of Mathematical and Computer Sciences, The University of Tulsa
[haynes,rogerw]@euler.mcs.utulsa.edu

Keywords– Genetic Programming, Variable Fitness Function, Parallel Evaluation of Fitness, Autonomous Agent

Abstract

In this paper we use the genetic programming technique to evolve programs to control an autonomous agent capable of learning how to survive in a hostile environment. In order to facilitate this goal, agents are run through random environment configurations. Randomly generated programs, which control the interaction of the agent with its environment, are recombined to form better programs. Each generation of the population of agents is placed into the Simulator with the ultimate goal of producing an agent capable of surviving any environment. The environment that an agent is presented consists of other agents, mines, and energy. The goal of this research is to construct a program which when executed will allow an agent (or agents) to correctly sense, and mark, the presence of items (energy and mines) in any environment. The Simulator determines the raw fitness of each agent by interpreting the associated program. General programs are evolved to solve this problem. Different environmental setups are presented to show the generality of the solution. These environments include one agent in a fixed environment, one agent in a fluctuating environment, and multiple agents in a fluctuating environment cooperating together. The genetic programming technique was extremely successful. The average fitness per generation in all three environments tested showed steady improvement. Programs were successfully generated that enabled an agent to handle any possible environment.

Introduction

Genetic Programming (GP) is an algorithmic paradigm intended to artificially construct computer programs to solve problems. Genetic programming is an extension of Genetic Algorithms (GA). The programmer provides a set of terminals and functions which enables the genetic programming technique to evolve a program to solve a given problem. The functionality given to the genetic program is derived from the problem under consideration. After a random population of programs is constructed, it is evolved into successive generations hopely with a better average fitness. After the population converges, or after a preset number of generations, the individual program with the best fitness is selected to be the solution. It is assumed that the reader is familiar with the fundamentals of genetic programming. Koza [7], provides an excellent overview of the fundamentals of genetic programming.

In this research, genetic programming is used as the foundation of a simulation of the interactions of autonomous agents in a hostile environment. Each agent is controlled by a program consisting of elements of functionality designed to allow the agent to interact with the Simulator. The Simulator is a map of a two dimensional grid that contains mines and energy. Each cell of the grid may have a mine, energy, both a mine and energy or neither.

The environment that an agent is presented consists of other agents, mines, and energy. In a multiple agent environment, each agent is allowed to interact with other agents to the extent of exchanging information about their world. This information includes the cell coordinates of mines and/or energy, and the item tallies of visited cells. More than one agent can occupy a given cell. Mines are to be avoided as they are capable of exploding and killing every agent occupying the cell. Energy is collected by the agent for sustenance. It is possible that lack of energy could cause an agent to die of hunger.

Genetic Programming is used to control the generation of new populations which are then run through the Simulator. In a fluctuating environment, each generation is presented with a different grid, and each agent is initially placed on the grid in a random location. The goal of the simulation is to breed an agent program which can survive in any environment which is presented to it.

In nearly all of the research to date in genetic programming, static fitness functions were used to calculate the fitness of candidate programs. An example of this is the Autonomous Ant problem [7], in which programs are constructed to navigate through a fixed environment in search of energy. Every generation of this problem is presented with the same trail of energy. In contrast, our Simulator randomly generates a new environment in which to judge each generation.

* Research partially supported by OCAST Grant AR2-004 and Sun Microsystems, Inc.

The dynamic environment presents several difficulties in comparing the performance of two generations in the Simulator. These difficulties for the most part can be dismissed due to the differing objectives of the two problems. For the Autonomous Ant problem, the intent is to breed a program to handle a specific environment. Thus, each generation can be compared to each other. In our Simulator, the goal is not to encode knowledge of a specific environment into an agent's program, but to encode algorithms for handling any environment into the agents program. Thus, each generation can not be compared to any other, unless the environments happen to be fixed. This concept is a relatively new area of research with regard to genetic programming. Other researchers using genetic programming to solve complex tasks include Angeline *et al.* [1], Husbands [3], Kinner [4], Koza [5], [6], [7], and Tackett [9].

The rest of this paper is organized as follows. Section 2 introduces the Simulator. Section 3 discusses the problem of one agent in a fixed environment and the problem of one agent in a fluctuating environment. Section 4 discusses multiple agents in a fluctuating environment. Conclusions are given in Section 5.

The Simulator

As part of this research, we wrote a genetic programming package in C to control the construction of programs to guide agents in navigating through a minefield. The fitness function of the GP is an interpreter which maps these programs into actions in the minefield. This interpreter is called the Simulator. The minefield, along with the agents, is called the Environment. The basic algorithm for the Genetic Program and the Simulator is shown below:

```
Generate an initial population of Agents
For each Generation do
    Generate an Environment
    For the number of allowable moves do
        Move each Agent in parallel
        Determine any mine explosions
    End
    Calculate the fitness of each Agent
    Track the best Agent of any Generation
    Calculate ave. fitness of the Generation
    Track the best average fitness of all Generations
    Perform reproduction based on fitness
    Perform Mutation
End
Determine the viability of the best Agent
```

The goal of this research is to construct a program which when executed will allow an agent (or agents) to correctly sense, and mark, the presence of items (energy and mines) in any environment. Any agent who can learn to apply a general algorithm to a specific instance of an environment will be successful. Any agent who tries to apply an algorithm designed for a specific environment to a changing environment will not be successful.

Each agent consists of an S-expression, a supply of energy, and a structure used to hold state information, or memory, for each run through the Simulator. The S-expression is stored as a parse tree, and is generated following the rule of initial tree generation with types. See Montana [8] for an excellent discussion on strongly typed genetic programming. The energy supply is the agents store of energy, and is initially randomly generated. The state information consists of the current cell locations, the planned move location, the desired direction to move in, the running raw fitness, a map of the minefield from the agent's perspective, and a linked-list representing the previous cells visited by the agent. The state information is initialized with the starting cell location.

In the multiple agent environment, agents may communicate information to each other. The rules governing this process is that in order to share information two agents must occupy the same cell. This restriction means they can only transfer cell coordinates of cells they have actually visited.

The minefield is a two dimensional grid of cells. In all of our simulations, the grid is either 10 by 10, or 20 by 20. Each cell represents an area which an agent can enter. Each cell can contain two items: mines and energy. Mines kill an agent, while energy provides sustenance and allows agents to continue operating. Also associated with each cell are tallies for each item. These tallies include the number of surrounding cells which contain each item. For example when an agent visits a cell the tally for that cell indicates how many of the eight neighboring cells have mines and how many of the eight neighboring cells have energy. Each agent can be thought of as carrying a sensor which can detect these tallies. An agent is not aware of the direction of the items, however. By examining the counters in surrounding cells, the agent is able to deduce which cells contain which items. This is similar to the game, minesweeper, except our model is more complicated.

As an agent moves through the Simulator (minefield) its movements are recorded. This recording allows an agent to apply deductive reasoning to the area of the Simulator for which it is knowledgeable. It can evolve algorithms to deduce which cells have either mines or energy. As it enters new cells, it might gain knowledge to cause a chain reaction of detection back through the cells it has previously visited.

Each agent has state information or memory. This memory takes the form of a dynamically allocated linked list. As an agent successfully enters a cell, it adds the new cell to the list of visited cells. There are two methods which can be applied to determine how the new cell is added to the linked list. The first simply adds cells so that the agent maintains a continuous trail of the cells it visits. The second method adds only new cells to the list. This problem and several other subtle problems has made the development of the Simulator and interesting project to work on. As a result, we developed several "tricks" to solve these issues. These and other issues are explained in more detail in [2].

Fitness Function

The Function and Terminal sets for this problem are described in [2]. In the Simulator, the fitness function does change from generation to generation. Indeed, the Simulator can be thought of as the fitness function. The main reason stock test cases were not used to evaluate individual agents and populations, was to allow for the development of an agent who could handle any valid environment in which it was placed. Since an agent is placed on the map in a cell without a mine, the hostile conditions that it can experience range from being totally surrounded by mines in all eight directions, to having no mines in its immediate vicinity. The globally optimal agent will be able to handle these conditions.

Each agent is assigned a total raw fitness based on its performance for meeting certain criteria in each run of the Simulator. These criteria include the detection of mines and energy, the collection of energy, the entering of cells which have not been visited, and surviving. All of these criteria modify the total raw fitness available in any run of the Simulator.

The goal of the fitness function is to reward agents who will detect energy and mines. We decided that detecting mines was worth more than detecting energy. Thus we assigned 100 raw fitness points for mine detection, and only 50 points for detecting energy. It is desirable for agent programs to develop detection of items, regardless of the actual types. Once agents start to deduce item locations, the simple genetic operations will ensure that this functionality is dispersed into the population as a whole. This desire for general item detection is the reason why energy detection is still highly rewarded.

We designed the Simulator such that agents will need to collect energy in order to survive. In order to facilitate the learning of this rule, we award 15 points for picking up energy. We also rewarded for the exploration of cells that have not yet been visited. It is only by going where they have not yet been that agents can gather new information about their environment. By gathering more information, an agent can potentially increase its raw fitness. We awarded an agent one raw fitness point for entering a cell for the first time.

The last fitness consideration is the desire to survive. An agent who survives until the end of the Simulation is awarded 15 points. Note that agents who survive are more likely to have more time in which to explore the environment in the Simulator. This can lead to a higher raw fitness score. Thus, the total fitness is described by the following formula:

$$fitness = (\#Mines \;\; X \;\; M_{dm}) + (\#Energy \;\; X$$
$$(M_{de} + M_{pe})) + (\#Cells \;\; X \;\; M_{ec})M_l$$

Where M_{dm}, M_{de}, M_{pe}, M_{ec}, and M_l are respectively the fitness awards for detecting mines, detecting energy, picking up energy, entering new cells, and surviving. The number of mines and energy is randomly determined for the Simulator in the first generation, and then the same numbers are used in every generation. In the fluctuating environment, however, the location of the items changes with each new generation.

The awarding of points for meeting criteria other than just surviving, allows good subprograms to be positively reinforced. It also allows for the possibility that good genetic material can be passed on through poor agent programs. The earlier discussion on the desire to breed item detection into the population illustrates this point. In a population in which no item detection is present, the agent which is able to detect energy will have a higher fitness than the rest of the population. Then either crossover or mutation can change the item detection from energy to mine. The rate of consumption of energy is related to the number of moves allowed.

Fixed Versus Fluctuating Environment

We investigated the difference between a fixed environment versus a fluctuating environment. In both cases, each agent is run through the Simulator alone. The reason agents are run alone is to ensure that each agent starts at the same cell location. Also, each agent is assigned a randomly determined supply of energy. Note that this supply is equal for each agent in all the generations of each run. Also note the movement allowance is greater than the number of cells. This means that each agent can potentially reach the maximum fitness. It is also expected that the Fixed Run will produce agents who have their environment directly encode into their programs. The Fluctuating Run will not have this encoding.

Fixed Environment

An initialized environment is created, and each agent in each generation is run through this same environment. Table 1 shows the parameters for one agent in a fixed environment.

Table 1: Parameters for One Agent in a Fixed Environment

Number of Agents	75
Number of Moves	150
Number of Generations	100
Field Size	10
Maximum Possible Fitness	2349
Starting Energy Range	3 - 9
Award for Moving	1
Award for Detecting Mines	100
Award for Detecting Energy	50
Award for Picking Up Energy	5
Award for Surviving	15

Figure 1 displays the best and average fitness for each generation for one agent in a fixed environment. The average fitness per generation steadily improves from one generation to the next. The best fitness per generation has the same general shape as the average fitness. However, it is not as smooth. This run was very successful. Clearly by the 40th generation, the population as a whole is beginning to encode information about the environment. The resulting solution program for this simulation appeared at generation 55 with a fitness values of 1909. The program has 27 nodes and is shown below.

```
Dual( ConTack( SetStatus(N, IFTE(F, M, En), P),
               PickTack(NT, M, P)),
      Dual( ConTack( SetStatus
           ( PickTack(NT, M, P), M, P),
                 W),
           IFTE(T, DM, cSW)))
```

Fluctuating Environment

In this study a new environment is created for each successive generation. Each agent is then run through this envi-

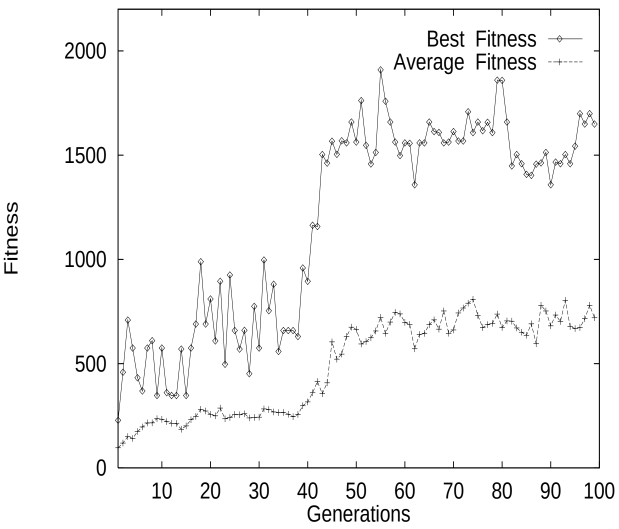

Figure 1: Best and Average Run per Generation for One Agent in a Fixed Environment

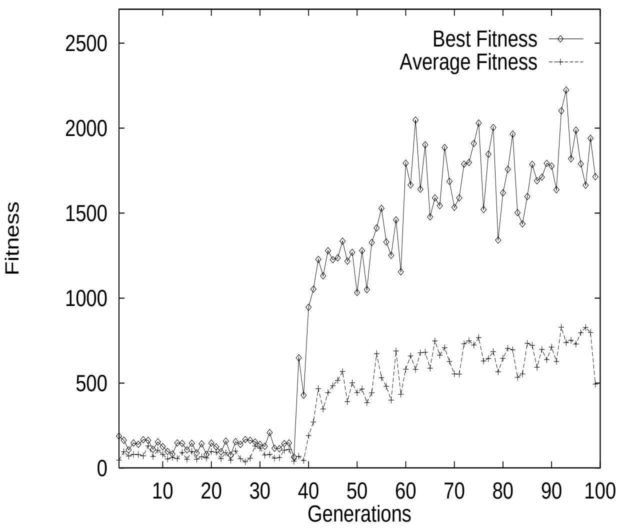

Figure 2: Best and Average Run per Generation for One Agent in a Fluctuating Environment

ronment. Note agents always start from the same initial cell, which was randomly determined for the first environment. The parameters to initialize this Simulation are shown in Table 2.

Table 2: Parameters for One Agent in a Fluctuation Environment

Number of Agents	75
Number of Moves	150
Number of Generations	100
Field Size	10
Maximum Possible Fitness	2404
Starting Energy Range	3 - 9
Award for Moving	1
Award for Detecting Mines	100
Award for Detecting Energy	50
Award for Picking Up Energy	5
Award for Surviving	15

Figure 2 depicts the results for the average and best fitness for each generation for a single agent in a fluctuating environment. Notice there is a distinct lack of detection of items until about the 40th to 50th generations. It is at this time that the average and best fitnesses per generation jump dramatically. The resulting solution program in this simulation appeared at generation 93 with a fitness values of 2224. The solution program has 32 nodes and is shown below.

```
Dual( IFTE(
       =( SetStatus( NT, En, P ),
          SetStatus( NT, En, P )),
       DM,
       DM),
      IFTE(
       =(Plan(SetStatus(NT, M, P)), DtM(NT)),
       DM,
       IFTE(=(ZN(NT, M), Plan(MH)), DM, DM)) )
```

Note in the fixed environment simulation, the agents managed to get the item detection functions earlier on in the simulation, since the curves are smoother. The average fitness curve for the fixed run gradually increased until a "jump" occurred associated with the functionality to detect mines (Figure 1). In contrast, the fluctuating run had an almost linear average fitness until it got the needed functionality to detect mines (Figure 2). Once that functionality was bred into the population, the average fitness jumped dramatically as expected. One of the goals for this research has been realized in that the excellent performance of the fluctuating runs indicates that a general solution had been found.

Multiple Agents – Fluctuating Environment

In this simulation multiple agents navigate through a fluctuating environment sharing information about the minefield and competing for a fixed amount of energy. The relevant parameters used to control this Simulation are listed in Table 3.

There is a slow but steady improvement in both the average and best fitness per generation for the multiple agent simulation in a fluctuating environment as shown in Figure 3.

Figure 4 shows the number of agents starving, surviving, and exploding per generation for multiple agents in a fluctuating environment. Each graph in Figure 4 changes rapidly until about the 10th generation or so, and then appears to become fairly stable throughout the rest of the simulation. Initially, the number of agents surviving increases rapidly indicating the agents learned quickly to interact with their environment. The agents in this simulation adapted quickly to detecting and avoiding mines as indicated by the initial sharp decline in agents exploding. The simulation shows that death by starvation rises to dominate the simulation. This phenomenon is simulation dependent, and is of little concern. In our case this occurred because there was too

Table 3: Parameters for Multiple Agents in a Fluctuation Environment

Number of Agents	75
Number of Moves	150
Number of Generations	100
Field Size	20
Maximum Possible Fitness	25115
Mine Range	100 - 200
Energy Range	150 - 250
Starting Energy Range	3 - 9
Award for Moving	1
Award for Detecting Mines	100
Award for Detecting Energy	50
Award for Picking Up Energy	5
Award for Surviving	15

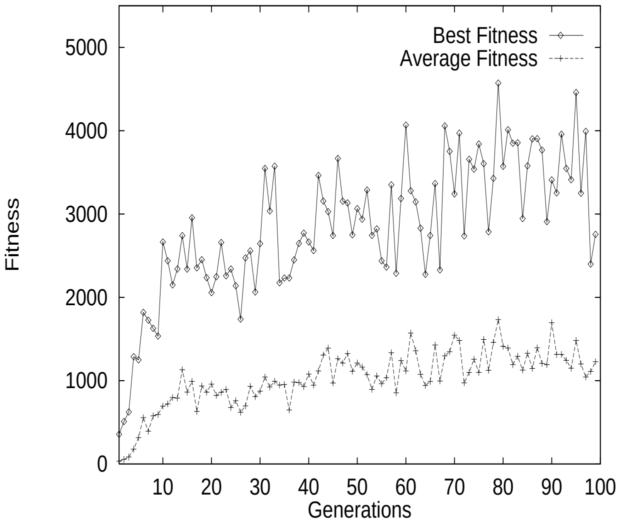

Figure 3: Best and Average Run per Generation for Multiple Agents in a Fluctuating Environment

much competition for the small amount of energy that we made available.

Note in Figure 5 the average number of new cells entered levels off after about ten generations. This can be linked to the detection of mines which is shown to be increasing in this same time period (Figure 4). It is also at this time that the death by explosion is decreasing (Figure 4). Thus the agents are not blindly entering new cells. An apparent difference between the detection of mines and energy is that mines are detected early on in the Simulation. This detection is clearly shown in Figure 5.

The resulting solution program in this simulation appeared at generation 79 with a fitness values of 4572. The program has 27 nodes and is shown below.

```
Dual( Dual( ConTack( SetStatus( NT, En, P ), NW ),
            ConTack( SetStatus( NT, En, P ), NT )),
      ConTack( Dual( Plan( DtM( NT ) ),
                     SetStatus( NT, En, P )),
               PickTack( NT, M, NP ))  )
```

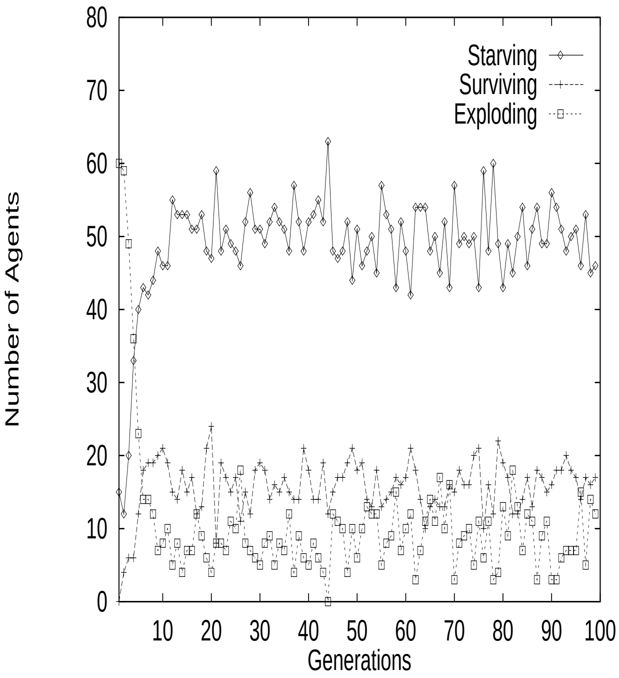

Figure 4: Number of Agents (Starving, Surviving, Exploding) per Generation for Multiple Agents in a Fluctuating Environment

The excellent performance shown in Figure 3 indicates that a general solution to the multiple agent problem in a fluctuating environment had been developed.

Conclusions

There are several possible applications for this research. One example is military aircraft entering territory which has target points and either radar sites or antiaircraft capabilities. It is the goal of the pilot to avoid the radar, and get to the target points.

Another application is in navigating terrain that has been contaminated with either nuclear, chemical or bacterial waste. Geiger counters measure the intensity of radiation, but they do not necessarily provide a direction of the source. By watching the increased activity of the meter, contaminated areas are avoided.

The goal of this research is to evolve programs to control agents in a simulation of a minefield. This environment is hazardous to the survival of the agent. The bulk of the raw fitness points is awarded for detecting mines. This led to the generation of agent programs which survived the Simulation by detecting mines.

All of the runs of the Simulator exhibited the same common behavior. There was an initial high death rate caused by explosion. As the agents adapted by detecting mines, there was a corresponding increase in death by starvation. However, the agents slowly improved their average fitness despite this handicap. This indicates that the agents were able to evolve better algorithms for survival in spite of dying in the process. In essence, the successful agents passed good

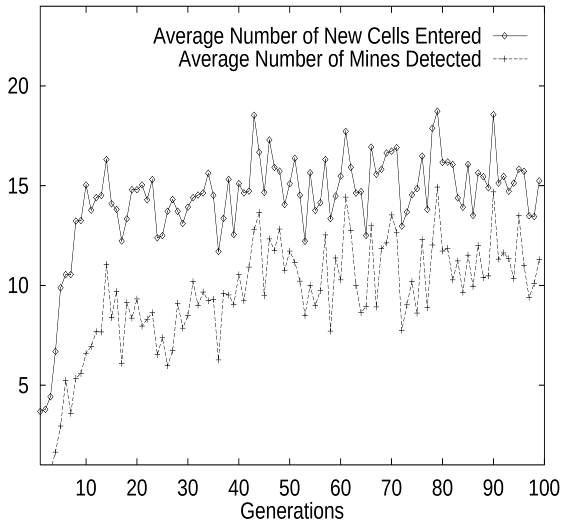

Figure 5: Number of Cells Entered and Mines Detected per Genetation for Multiple Agents in a Fluctuating Environment

genetic material to the next generation.

The Simulator did provide enough data to show that it is capable of creating agents who can survive in an hostile environment. As a result, we were very successful in achieving programs which enabled an agent to handle *any* possible environment. There was a definite repetition of certain program fragments across all the different runs of the Simulator. This supports the emergence for a general program to solve this problem. We obtained excellent performance in generating programs for one agent in a fixed environment, one agent in a fluctuating environment, and in the more complicated case of generating a general solution to the multiple agent in a fluctuating environment.

Acknowledgements

This research has been partially supported by OCAST Grant AR2-004. The Authors also acknowledge the support of Sun Microsystems, Inc.

References

[1] Angeline, Peter J., and Pollack, Jordan B., Competitive Environments Evolve Better Solutions for Complex Tasks, *Proceedings of the Fifth International Conference on Genetic Algorithms*, Morgan Kaufmann Publishers, Inc., pages 264 - 278, 1993.

[2] Haynes, Thomas D., "A Simulation of Adaptive Agents in a Hostile Environment", Master's Thesis, Department of Mathematical and Computer Sciences, University of Tulsa, Tulsa, OK., April, 1994.

[3] Husbands, Philip, Genetic Algorithms in Optimisation and Adaptation, *Advances in Parallel Algorithms*, Kronsjo, Lydia and Shumsheruddin, Dean, ed., Halstead Press, New York, 1992, pages 227-276.

[4] Kinnear, Kenneth E. Jr., Generality and Difficulty in Genetic Programming: Evolving a Sort, *Proceedings of the Fifth International Conference on Genetic Algorithms*, Morgan Kaufmann Publishers, pages 287 - 294, Inc., 1993.

[5] Koza, John R., Evolving a Computer Program to Generate Random Numbers Using the Genetic Programming Paradigm, *Proceedings of the Fourth International Conference on Genetic Algorithms*, Morgan Kaufmann Publishers, Inc., pages 37 - 44, 1991.

[6] Koza, John R., Simultaneous Discovery of Reusable Detectors and Subroutines Using Genetic Programming, *Proceedings of the Fifth International Conference on Genetic Algorithms*, Morgan Kaufmann Publishers, Inc., pages 295 - 302, 1993.

[7] Koza, John R., *Genetic Programming, On the Programming of Computers by Means of Natural Selection*, MIT Press, 1992.

[8] Montana, David J., BBN Technical Report No. 7866: *Strongly Typed Genetic Programming*, Bolt Beranek and Newman, Inc., May 7, 1993.

[9] Tackett, Walter A., Genetic Programming for Feature Discovery and Image Discrimination, *Proceedings of the Fifth International Conference on Genetic Algorithms*, Morgan Kaufmann Publishers, pages 303 - 309, Inc., 1993.