# Genetic Evolution of Machine Language Software

**Ronald L. Crepeau**
NCCOSC RDTE Division
San Diego, CA 92152-5000
e-mail: crepeau@nosc.mil
July 1995

**Abstract**

Genetic Programming (GP) has a proven capability to routinely evolve software that provides a solution function for the specified problem. Prior work in this area has been based upon the use of relatively small sets of pre-defined operators and terminals germane to the problem domain. This paper reports on GP experiments involving a large set of general purpose operators and terminals. Specifically, a microprocessor architecture with 660 instructions and 255 bytes of memory provides the operators and terminals for a GP environment. Using this environment, GP is applied to the beginning programmer problem of generating a desired string output, e.g., "Hello World". Results are presented on: the feasibility of using this large operator set and architectural representation; and, the computations required to breed string outputting programs vs. the size of the string and the GP parameters employed.

## 1 INTRODUCTION

Traditional Genetic Programming (GP) -- as invented by [Koza 1992a, 1992b, 1994] -- is implemented through a process that involves the specification and use of a small set of operators and terminals germane to the problem. The work of [Angeline 1993], [Angeline and Pollack 1994], [Kinnear 1991], [Rosca and Ballard 1994], [Teller 1993, 1994] and [Nordin 1994] are typical of this approach. Except for the work of Teller, the aforementioned GP implementations have not involved the use of memory elements beyond those implicit in the terminals. Nordin's implementation is somewhat similar to the work reported herein, in that he worked with various terminals and 24 assembly language instructions.

The use of a small set of problem specific operators and terminals, all without explicit memory elements, has a decided advantage: It limits the search space for potential solutions, thereby making it computationally practical to solve even complex problems. Contrarily, the use of problem specific operators and terminals requires their re-definition for each new problem. Moreover, traditional GP is not Turing complete, and is thus limited in the problems it can solve [Teller 1994].

It is commonly assumed that the use of a large set of generalized operators and terminals would make the GP process less efficient and potentially impractical. But if this assumption were erroneous it would open GP to the following advantages:

- The same GP environment could be applied to a variety of problems without re-programming for the fundamental operators or terminals.
- The addition of memory would provide the potential for Turing completeness and the solution of a large set of problems that traditional GP precludes.
- The use of evolutionary leveraging, i.e., the incremental evolution of new agents (programs) from the products of prior GP evolutions.

These advantages warrant investigation of GP using a large and generic set of operators augmented with memory. The remainder of this paper reports on such an investigation.

## 2 EXPERIMENTAL ARCHITECTURE

In lieu of developing an *ad hoc* set of operators and a related architecture for experimental use, the author took a more pragmatic approach, viz., the use of an existing microprocessor architecture with its related set of operators (instructions). This approach has the advantage of providing a proven, well defined and well documented architecture with a comprehensive and general purpose set of operators proven to be suitable for virtually any problem.

It was further decided to work at the machine language (ML) level. Use of this level of representation precludes the need to compile or assemble the programs involved. Because the selected ML is not native to the computers used in testing, an ML emulator is required. An emulator or some other execution control mechanism would have been required in any event, as it would be neither prudent nor practical to run programs in their native environment without controls on progress, addressing, jumps, register operations, stack operations, inputs, outputs, etc.

The overall design of the experimental architecture, hereafter referred to as the Genetic Evolution of Machine-language Software (GEMS) system, is comprised of three parts: the microprocessor emulator; a pool of ML programs; and, the Genetic Process Controller. These parts are integrated as shown in the top level architecture of Figure 1. The next two sections detail the characteristics and implementations of these parts. All implementation is in the C language. The current GEMS version runs under the Sun OS 4.1.3 and Solaris 2.4.

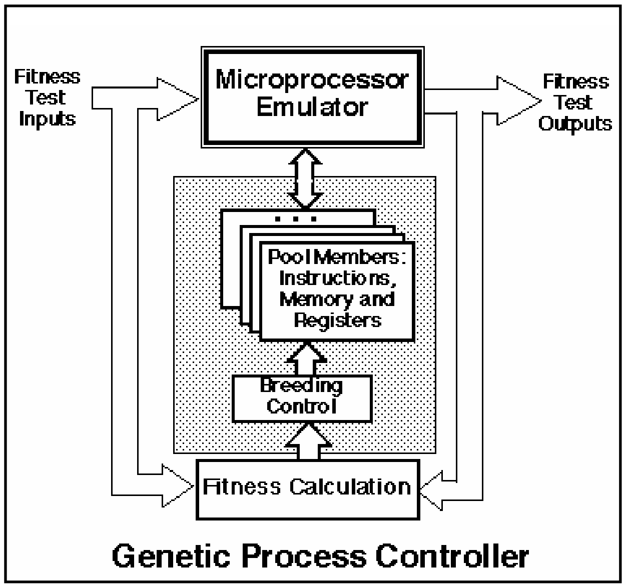


Figure 1. Top Level GEMS System Architecture.

## 2.1 MICROPROCESSOR EMULATOR AND ML PROGRAMS

A key and central element of the GEMS architecture is the microprocessor emulator. The emulator is a relatively static part of the GEMS system, in that it varies little from GP problem to GP problem.

The function of the microprocessor emulator is to execute the program (or a designated portion thereof) assigned to it from the pool. This execution proceeds for a specific number of instruction cycles as dynamically defined by the Genetic Process Controller. When, during program execution, the emulator encounters an input instruction, it receives the appropriate value from the Genetic Process Controller. Likewise, when an output instruction is encountered, its value is trapped by the Genetic Process Controller.

The microprocessor architecture emulated by GEMS is that of the Z80$^{TM}$. (Z80 is a registered trademark of ZILOG Inc.) The Z80$^{TM}$ is a circa-1980 16-bit microprocessor that was utilized as the central processing unit (CPU) for a variety of personal computers and controllers. Its large, low level instruction set, 64K addressing range and input-output features allowed this microprocessor to power a diverse set of applications including operating systems, word processors, spread sheets, simple graphics, etc. The Z80$^{TM}$ is of sufficient size and power for use on the planned GP problems yet was not too large for the author to program and check out its emulator within a three month period.

Figure 2 shows a typical system architecture for employing the Z80$^{TM}$. It is composed of:

- The Z80$^{TM}$ microprocessor itself
- Read Only Memory (ROM) - Not specifically emulated
- Random Access Memory (RAM)
- Input and output (I-O) ports
- Three buses

The threes buses are: data, address and control. The 8 bit wide data bus carries the byte of information being moved to or from memory or the I-O ports. The data is transferred based upon the 16 bit address on the address bus. The Control Bus provides synchronization signals needed in a hardware implementation, e.g., clock, read, write, etc. The data and address buses were implicitly implemented in the GEMS emulator.

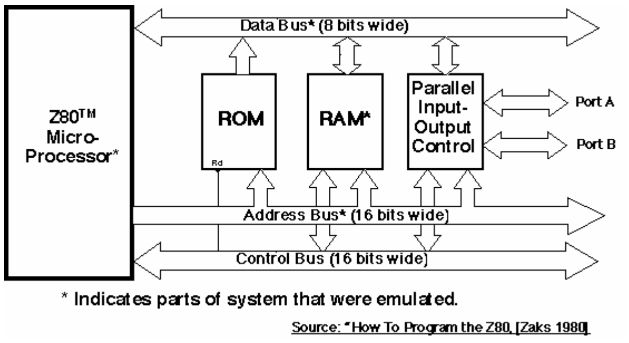

Data Bus* (8 bits wide)
Z80TM Micro-Processor*
ROM
Rd
RAM*
Parallel Input-Output Control
Port A
Port B
Address Bus* (16 bits wide)
Control Bus (16 bits wide)
* Indicates parts of system that were emulated.
Source: "How To Program the Z80, [Zaks 1980]

Figure 2. Typical Z80$^{TM}$ Architecture

### 2.1.1 Z80$^{TM}$ Characteristics and Emulation

The Z80$^{TM}$ does not include any internal cache or similar temporary memory. It does have seven 8-bit wide registers called the A (or accumulator), B, C, D, E, H and L. For some instructions, the combination of the B-C, D-E and H-

L registers are treated as effectively 16 bit registers. All of these registers are emulated.
The Z80$^{TM}$ incorporates four 16-bit wide registers called the IX, IY, SP (for Stack Pointer) and PC (for Program Counter). All of these registers are explicitly implemented except for the PC. Its function is implemented in a manner more germane to the GEMS needs for controlling program execution.

An 8-bit wide Flag (F) register is a part of the Z80$^{TM}$ and implemented in GEMS. The Flag register has individual bits assigned to be set or reset based on such things as an operation with a zero result (verses non-zero), carry, overflow, parity, etc.

The most significant aspect of the Z80$^{TM}$ is the instruction set it is capable of executing. The Z80$^{TM}$ has 691 unique instructions each of length 2, 4, 6 or 8 bytes, where each byte is 8 bits long. These instructions can be classified as:

- Arithmetic: Add, subtract, increment and decrement. The Z80$^{TM}$ has no hardware based multiply or divide instructions.
- Boolean logic: AND, OR, XOR
- Bit: Set bit, restore bit, get bit, rotate bits right/left, shift bits right/left.
- Load: direct, indirect, block
- Jump: Conditional, un-conditional; direct, relative
- Subroutine: Jump and return
- Stack: Push and pop
- Input and output
- Interrupt
- Halt

Of the 691 total, GEMS implements 660 instructions. It made no sense to implement the interrupt instructions at this time. Block input-output and block move instructions are not implemented as their effects can be achieve via the implemented instructions. The second (mirror) set of registers are not implemented as not currently needed.

The individual instructions of the pool members are implemented using a structure called "insts", which is defined as:

```
struct    insts
{ int              opCodeSize;
  Byte             instHexValues[8];
 };
```

The opCodeSize specifies the number of bytes in the instruction. The array instHexValues holds the actual instruction bytes. A GEMS pool member's set of instructions are stored as an array of the above structures. By using this approach the emulator stays synchronized with instruction boundaries during execution. Moreover, breeding and mutation can be made to take place at clean instruction boundaries, obviating a critical problem with using assembly language for GP.

The GEMS pool members are composed of more than just the program instructions. Rather, they consist of pseudo-machine images. Thus, for each pool member, GEMS stores not only the instructions of the program, but also other relevant execution data. Figure 3 shows the data structure, z80, used for the pool members. As may be seen, the data structure holds the status of the:

- Registers, including the flag and stack pointer.
- The number of the last instruction executed.
- The most recent fitness value.
- An array of instruction structures (“codeList”) with maxProgLength being the number of instructions in the pool members.
- An array of values (“data”) representing the memory contents, with critterDataSize being the amount of memory allocated to the pool members.
- Cumulative subroutine calls and pushes to the stack.

```
struct z80  {
     Byte          A;     /* Accumulator */
     Byte          B;     /* B register */
     Byte          C;     /* C register */
     Byte          D;     /* D register */
     Byte          E;     /* E register */
     Byte          H;     /* H register */
     Byte          L;     /* L register */
     int           IX;    /* IX register */
     int           IY;    /* IY register */
     Byte          F;       /* Flag  register */
     int           SP;    /* Stack pointer */
     int           inst;   /* Instruction # */
     long           strength;  /* Latest fitness */
     long           age;   /* Age since birth */

     struct insts   codeList[maxProgLength];
                          /* Program instructions */

     int           data[critterDataSize];
                          /* Memory contents */

     int           callsMade;
                            /* Net subroutine calls */

     int            pushesMade;
                            /* Net stack pushes */
};
```

Figure 3. Data Structure for Pool Members.

When a pool member, i.e., z80 data structure, is processed for its fitness, the registers, memory, last instruction, and calls/pushes are dynamically updated as the instructions are executed. Upon completion of a run, the fitness is calculated and written into the structure. In this way, the pool members' states are constantly up to date.

### 2.1.2 Memory Implementation

The Z80$^{TM}$ has a 64K addressing capability. This value is deemed larger than needed and somewhat impractical for emulation. As such, the emulations of this paper are limited to 255 instructions and 255 memory locations. While these may seem exceptionally modest numbers in today's world of megabyte memories, it should be noted that a good programmer can create some rather powerful capabilities with these few instructions, e.g., early machine boot-strapping programs consisted of one to two dozen machine language instructions.

The GEMS design deals with the incompatibility between the 64K of addressing potential of its randomly generated instructions and the upper limits of memory by using modulo arithmetic in calculating addresses accessed. In this manner, addressing circulates backward and forward through memory as if the end of memory were adjacent to the beginning. No detriment has been found in operating GP under this expediency.

The reader may have noted that the z80 data structure of Figure 3 segregates the program memory from the data memory. This is not typical of a microprocessor implementation. Normally, the instructions, stack and data reside in a contiguous memory space and grow towards each other, hopefully avoiding a collision and resultant system crashes. Because of the addressing limitations and the way instructions are implemented in GEMS it is expedient to treat the two memories separately.

### 2.1.3 Input - Output

An important and highly useful aspect of the Z80$^{TM}$ architecture is its input and output capabilities. The Z80$^{TM}$ architecture can address up to 64K of input and output ports via one of twelve input or output instructions (Only ten are implemented. Block input and output are not implemented.)

Seven of the input/output instructions execute a read from/write-to the port address given by the eight bits of the B register (MSBs) and the eight bits of the C register (LSBs) and exchange the eight bits of data with the A, B, C, D, E, H or L registers. An additional instruction reads from/writes-to the port given by the last byte of the instruction and exchanges the data with the A register.

In a Z80$^{TM}$ hardware implementation, doing input or output would require the electronics to decode the address and place the appropriate data on the I-O bus lines. It is much more convenient to deal with the large number of ports in software, thus resulting in a large number of ports being easily accessible for entering and extracting information from the Z80$^{TM}$. These ports also provide a buffer between the emulation process and the external world or Genetic Process Controller as defined in the next section.

## 2.2 GENETIC PROCESS CONTROLLER

As shown in Figure 1, the Genetic Process Controller of the GEMS system generates new pools of microprocessors, links the pool of microprocessors with the emulator, controls the inputs to the fitness tests, accepts the fitness test outputs, evaluates pool members' fitnesses and controls the breeding, mutation and survival (or otherwise) of the pool members. All of this is managed by the user via an interactive user control function.

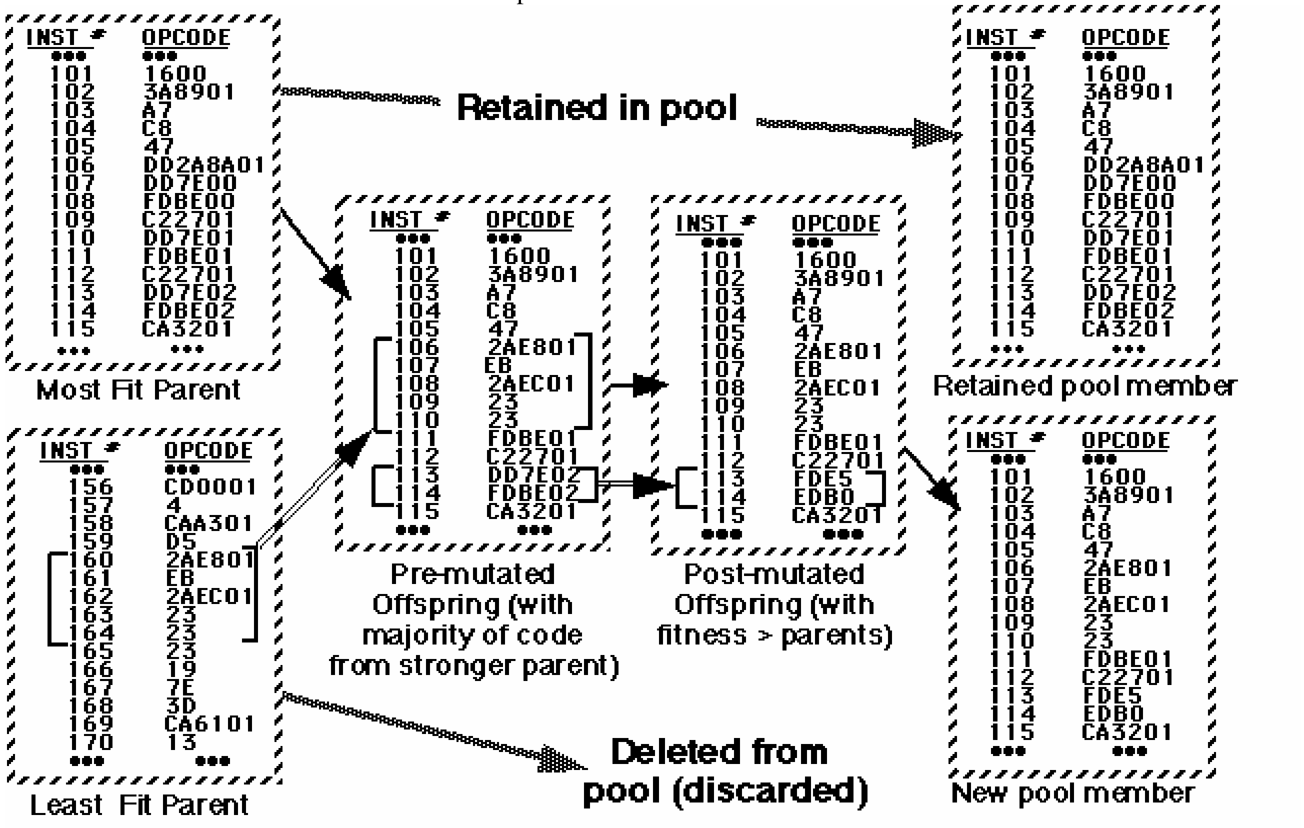
### 2.2.1 Breeding Overview

GEMS employs the breeding approach the author calls "asynchronous". This approach is similar to the Steady State GP of [Reynolds 1993] and [Kinnear 1991], except that there is no attempt to guarantee the uniqueness of each individual in the population -- thus the difference in nomenclature. This approach stands in contrast to generational GP wherein simultaneous breeding (conceptually, as the processes are actually implemented in serial machines) takes place such that the n pool members are pairwise bred, yielding n/2 off-spring and a new pool of 3n/2 members. These are pared back to n by some fitness related selection process.

In the asynchronous breeding process employed by GEMS, two parents are selected from the pool and bred one or more times. As each off-spring is bred, it is evaluated for insertion in the pool using a modification of the process which [Altenberg 1994] calls "upward mobility" selection. Thus, if the off-spring is more fit than both of its parents it replaces the weaker of the parents in the pool. If an off-spring fails to perform better than both parents, it does not survive. The selected parents breed up to k off-spring (k a small number, e.g., less than 12) or until one of the parents is replaced. The rational for multiple off-spring is its analogy to nature, wherein parents of high order creatures typically have litters of numerous off-spring.

### 2.2.2 Breeding Process

The breeding process within GEMS involves a variant of double cross-over with preference given to the most fit of the parents. Figure 4 illustrates the process. It begins by copying all of the code, memory and registers contents from the stronger parent into the potential off-spring. Next, a random number is selected which is greater than 0 and less than half the total number of instructions in the pool programs (maxProgSize). This represents the size of the code block to be removed from the weaker parent and inserted into the stronger code initially in the off-spring. A second random number is generated to define where to begin removing the block of code from the weaker parent.



Retained in pool
Most Fit Parent
Least Fit Parent
Pre-mutated Offspring (with majority of code from stronger parent)
Post-mutated Offspring (with fitness > parents)
Retained pool member
Deleted from pool (discarded)
New pool member

Most Fit Parent

| INST # | OPCODE |
|---|---|
| ●●● | ●●● |
| 101 | 1600 |
| 102 | 3A8901 |
| 103 | A7 |
| 104 | C8 |
| 105 | 47 |
| 106 | DD2A8A01 |
| 107 | DD7E00 |
| 108 | FDBE00 |
| 109 | C22701 |
| 110 | DD7E01 |
| 111 | FDBE01 |
| 112 | C22701 |
| 113 | DD7E02 |
| 114 | FDBE02 |
| 115 | CA3201 |
| ●●● | ●●● |

Least Fit Parent

| INST # | OPCODE |
|---|---|
| ●●● | ●●● |
| 156 | CD0001 |
| 157 | 4 |
| 158 | CAA301 |
| 159 | D5 |
| 160 | 2AE801 |
| 161 | EB |
| 162 | 2AEC01 |
| 163 | 23 |
| 164 | 23 |
| 165 | 23 |
| 166 | 19 |
| 167 | 7E |
| 168 | 3D |
| 169 | CA6101 |
| 170 | 13 |
| ●●● | ●●● |

Pre-mutated Offspring (with majority of code from stronger parent)

| INST # | OPCODE |
|---|---|
| ●●● | ●●● |
| 101 | 1600 |
| 102 | 3A8901 |
| 103 | A7 |
| 104 | C8 |
| 105 | 47 |
| 106 | 2AE801 |
| 107 | EB |
| 108 | 2AEC01 |
| 109 | 23 |
| 110 | 23 |
| 111 | FDBE01 |
| 112 | C22701 |
| 113 | DD7E02 |
| 114 | FDBE02 |
| 115 | CA3201 |
| ●●● | ●●● |

Post-mutated Offspring (with fitness > parents)

| INST # | OPCODE |
|---|---|
| ●●● | ●●● |
| 101 | 1600 |
| 102 | 3A8901 |
| 103 | A7 |
| 104 | C8 |
| 105 | 47 |
| 106 | 2AE801 |
| 107 | EB |
| 108 | 2AEC01 |
| 109 | 23 |
| 110 | 23 |
| 111 | FDBE01 |
| 112 | C22701 |
| 113 | FDE5 |
| 114 | EDB0 |
| 115 | CA3201 |
| ●●● | ●●● |

Retained pool member

| INST # | OPCODE |
|---|---|
| ●●● | ●●● |
| 101 | 1600 |
| 102 | 3A8901 |
| 103 | A7 |
| 104 | C8 |
| 105 | 47 |
| 106 | DD2A8A01 |
| 107 | DD7E00 |
| 108 | FDBE00 |
| 109 | C22701 |
| 110 | DD7E01 |
| 111 | FDBE01 |
| 112 | C22701 |
| 113 | DD7E02 |
| 114 | FDBE02 |
| 115 | CA3201 |
| ●●● | ●●● |

New pool member

| INST # | OPCODE |
|---|---|
| ●●● | ●●● |
| 101 | 1600 |
| 102 | 3A8901 |
| 103 | A7 |
| 104 | C8 |
| 105 | 47 |
| 106 | 2AE801 |
| 107 | EB |
| 108 | 2AEC01 |
| 109 | 23 |
| 110 | 23 |
| 111 | FDBE01 |
| 112 | C22701 |
| 113 | FDE5 |
| 114 | EDB0 |
| 115 | CA3201 |
| ●●● | ●●● |

Figure 4. The GEMS Breeding and Mutation Process.

This latter number is selected so as to ensure that the whole block of code can be removed without exceeding the end of the weaker parent's code.

All of the selected block of code from the weaker parent is randomly inserted into the off-spring to replace that from the stronger parent. Thus, as shown in Figure 4, instructions 160 through 164 have been randomly selected to replace instructions 106 through 110 of the stronger parent's code that was given to the offspring.

In a manner similar to the above, a segment (less than half) of the weaker parent's memory contents replaces a comparable amount of the stronger parent's memory contents in the off-spring.

### 2.2.3 Mutation

GEMS employs mutation of two types. In one case, mutation is used to replace a random amount of contiguous program and memory values in the off-spring. For example, in Figure 4 instructions 113 and 114 of the off-spring are replaced by totally new instructions. This is done prior to any fitness evaluation.

A second aspect of mutation is to randomly and totally replace a weak pool member. Each of these measures attempts to ensure that genetic diversity is maintained in the pool.

### 2.2.4 Fitness Evaluation

The fitness evaluation process in GEMS depends upon the specific problem. The following section elaborates how fitness was evaluated for this initial GEMS experiment.

## 3 THE "Hello World" PROBLEM

The initial problem used to evaluate genetic programming with machine language and memory was that typical of a beginning programmer, viz., output a string -- specifically, the eleven character string "Hello World".

The approach employed in generating the "Hello World" string was used to generate all of the results presented in this paper. The approach required three adaptations of the basic GEMS capability involving the INPUT and OUTPUT instruction processing and the fitness evaluation.

## 3.1 THE PROBLEM AND CONDITIONS

Succinctly stated the goal of the "Hello World" Problem (HWP) is to evolve a machine language agent which after completion of one run will have output the ASCII codes for the characters of "Hello World" to the virtual ports 1 through 11, respectively. Additional conditions on the problem are:

- The agent may output more than one character to a port, but the last character output to that port is the one upon which fitness is based. Prior outputs cause no penalties.
- The agent can output the characters in any order.
- The agent's run is always initiated at the first instruction and runs until a HALT instruction is encountered or 255 instruction cycles have been executed.
- An agent's run always begins with the registers and memory set at the states they were in at the completion of that agent's most recent run.

## 3.2 TREATMENT OF INPUT INSTRUCTIONS

Normally, the use of GEMS for a GP application would require that the input instructions be modified to accommodate training or situational data inputs. While the HWP does not require any inputs, it can be expected that in the course of evaluating a pool member the situation will occur wherein the agent looks for an input at one of the ports. When this happens in the HWP implementation, a random eight bit value is provided, except for port number 1. When port 1 is accessed, one of the characters of the desired string is randomly input. No testing was performed to determine whether or not this had any positive effect.

## 3.3 TREATMENT OF OUTPUT INSTRUCTIONS

Normally, the use of GEMS for a GP application would require that the output instructions be monitored in order to simulate interaction with the environment. Additionally, at least some of the outputs must be monitored in order to evaluate the fitness of the run.

For the HWP, the GEMS program code is modified to monitor the outputs of the agent and record the latest valid output to each of ports 1 through 11. First, the output is checked to see if it represents a printable ASCII character. If so, the value is recorded (replacing any prior output to that port) for use in the run's fitness evaluation. If the output is not a printable ASCII character, it is ignored.

## 3.4 FITNESS EVALUATION

Two fitness phases, with related criteria and scores, are employed for the HWP. They are:

- Phase 1: This fitness scoring is strictly based upon the correctness of the output string.
- Phase 2: In so long as the ML agent is outputting the correct string, the fitness score is augmented by a value related to the shortness of the agent.

Phase 1 Fitness Scoring: The fitness score of a given output from a pool member was heuristically derived from some early tests. It is calculated from two parts:

1. The Hamming distance (i.e., number of bit mis-matches) between the binary ASCII representation of the goal string and the output from the run. Both the output and goal strings are treated as 88 bit vectors formed from concatenation of the 11 8-bit binary representations of the ASCII characters.
2. The actual number of correct characters (max. = 11).

The fitness score for this phase is calculated as:

3 * (88- Hamming_distance) + 8 * (# of correct characters outputted)

The maximum score for all letters correct is 352.

Phase 2. Agent Brevity Fitness Scoring: If a pool member outputs the correct string and in less than 255 instructions, it's fitness score is increased by a value of: 255 minus the length of the agent in instructions.

Thus, the maximum possible fitness for the HWP occurs when the shortest ML agent puts out a correct string. The minimum length agent, including the HALT instruction, is 17 instructions so the best possible HWP fitness score is 590.

## 3.5 SAMPLE TEST RESULTS

The GEMS process did indeed generate an agent that output "Hello World" within a practical amount of computing time. For purposes of orientation and illustration, the results of one HWP run are shown in Figures 5 and 6. This run used a pool of 1500 members, a 20% mutation rate and uniform random selection of parents.

Figure 5 illustrates the normal progression curve for the GP process. As with most GP progression curves, the pool average, best in pool and best-to-date fitness values are plotted. In this case, because GEMS uses asynchronous breeding, the x-axis progress number is "spawns" versus the traditional GP “generation” number. A spawn represents the selection of one set of parents which generate off-spring until one of the parents is replaced by a more fit off-spring or a maximum litter is spawned. The relationship between a spawn and a traditional GP generation is not exact, as it depends upon the average number of off-spring generated. Assume that litters average r off-spring, then the equivalency of generations (G) to spawns (S) is: G = (r*S)/P, where P is the pool size. For the HWP problem, the maximum litter size was 12, so the number of equivalent generations per spawn is somewhere between G=S/1500 and G=S/125.

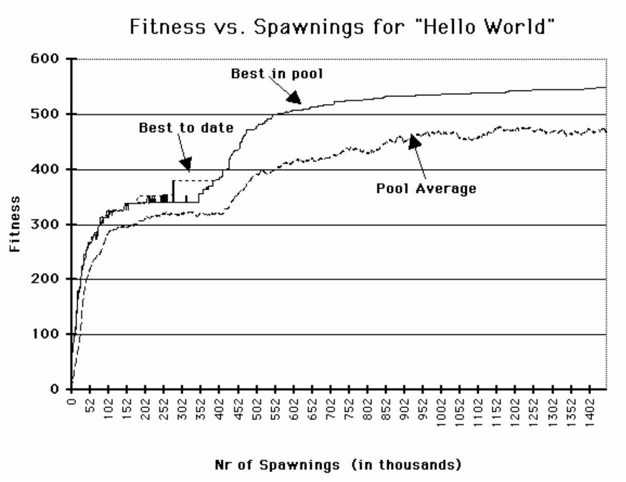


Figure 5. Graph of Fitness vs. Spawnings.

From Figure 5 it can be seen that this run achieved a correct output (fitness = 352) at about 150,000 spawnings (100 to 1200 generations). By about 450,000 spawnings, the agent was composed of less than 100 instructions. Ultimately, the agent size reduced to 58 instructions before the process was terminated.

Figure 6 shows the 58 instructions of the Figure 5 run in their assembly language format. The register contents are also shown. The far right column gives the state of the output ports (1-11 reading left to right) when they change.

Interestingly, the agent uses only the OUT instruction that transfers the A register's value to the port given in the OUT instruction's second byte. Section 5 discusses why.

The GP solution also took advantage of the latitude for outputting the characters in any sequence. It should be noted that the three l's and two o's are output before any change is made to the A register. This has been noted in other runs of different strings.

An analysis of the program indicates that it is clearly using memory contents to get the appropriate A register values, albeit in a very indirect manner. Thus, instruction 1 reads memory location 0xAA into the DE register. At instruction 6, register D is subtracted from register A and the value left in A. At instruction 11, DE is subtracted from HL (with carry) and subsequently the H register is OR'd with register A leaving the ASCII value for the letter l in register A, which is output at instructions 14, 15 and 19.

Instruction 22 is another case where memory is used. Here the content of memory location 0x8C is accessed to get a value to subtract from the current A register content which yields the ASCII value for the delimiting character. (Note: A period was used as a word delimiter versus a space for programming expediency.)

| **Inst** | | **Register Contents** | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| **#** | **Instruction** | **A** | **B** | **C** | **D** | **E** | **H** | **L** | **Output** |
| 0 | LD SP, HL | 65 | 5D | 5D | FA | F5 | 1 | 24 | "-----------" |
| 1 | LD DE, (0xAA) | 65 | 5D | 5D | F3 | 6F | 1 | 24 | |
| 2 | LD C to H | 65 | 5D | 5D | F3 | 6F | 5D | 24 | |
| 3 | LD H to A | 5D | 5D | 5D | F3 | 6F | 5D | 24 | |
| 4 | SRA H | 5D | 5D | 5D | F3 | 6F | 2E | 24 | |
| 5 | DEC H | 5D | 5D | 5D | F3 | 6F | 2D | 24 | |
| 6 | SBC A, D | 6A | 5D | 5D | F3 | 6F | 2D | 24 | |
| 7 | RES 1, A | 68 | 5D | 5D | F3 | 6F | 2D | 24 | |
| 8 | LD L to E | 68 | 5D | 5D | F3 | 24 | 2D | 24 | |
| 9 | POP IY | 68 | 5D | 5D | F3 | 24 | 2D | 24 | |
| 10 | RLC D | 68 | 5D | 5D | E7 | 24 | 2D | 24 | |
| 11 | SBC HL, DE | 68 | 5D | 5D | E7 | 24 | 44 | FF | |
| 12 | RES 0, H | 68 | 5D | 5D | E7 | 24 | 44 | FF | |
| 13 | OR H | 6C | 5D | 5D | E7 | 24 | 44 | FF | |
| 14 | OUT (N=0xA), A | 6C | 5D | 5D | E7 | 24 | 44 | FF | "---------l-" |
| 15 | OUT (N=0x4), A | 6C | 5D | 5D | E7 | 24 | 44 | FF | "---l-----l-" |
| 16 | POP DE | 6C | 5D | 5D | 6F | 6A | 44 | FF | |
| 17 | ADD IY, IX | 6C | 5D | 5D | 6F | 6A | 44 | FF | |
| 18 | LD C to B | 6C | 5D | 5D | 6F | 6A | 44 | FF | |
| 19 | OUT (N=0x3), A | 6C | 5D | 5D | 6F | 6A | 44 | FF | "--ll-----l-" |
| 20 | SUB A, B | F | 5D | 5D | 6F | 6A | 44 | FF | |
| 21 | LD D to C | F | 5D | 6F | 6F | 6A | 44 | FF | |
| 22 | SBC A, (IY+0x3E) | 2E | 5D | 6F | 6F | 6A | 44 | FF | |
| 23 | LD (HL=0x43) to D | 2E | 5D | 6F | F7 | 6A | 44 | FF | |
| 24 | OUT (N=0x6), A | 2E | 5D | 6F | F7 | 6A | 44 | FF | "--ll-.---l-" |
| 25 | SRA D | 2E | 5D | 6F | FB | 6A | 44 | FF | |
| 26 | AND B | C | 5D | 6F | FB | 6A | 44 | FF | |
| 27 | RES 0, D | C | 5D | 6F | FA | 6A | 44 | FF | |
| 28 | OR B | 5D | 5D | 6F | FA | 6A | 44 | FF | |
| 29 | ADD A,D | 5D | 5D | 6F | FA | 6A | 44 | FF | |
| 30 | OUT (N=0x7), A | 57 | 5D | 6F | FA | 6A | 44 | FF | "--ll-.W--l-" |
| 31 | BIT 3,D | 57 | 5D | 6F | FA | 6A | 44 | FF | |
| 32 | LD C to A | 6F | 5D | 6F | FA | 6A | 44 | FF | |
| 33 | OR B | 7F | 5D | 6F | FA | 6A | 44 | FF | |
| 34 | ADD A, n=0xE5 | 64 | 5D | 6F | FA | 6A | 44 | FF | |
| 35 | OUT (N=0xB), A | 64 | 5D | 6F | FA | 6A | 44 | FF | "--ll-.W--ld" |
| 36 | SRL A | 32 | 5D | 6F | FA | 6A | 44 | FF | |
| 37 | OR B | 7F | 5D | 6F | FA | 6A | 44 | FF | |
| 38 | SRA E | 7F | 5D | 6F | FA | 35 | 44 | FF | |
| 39 | RES 4, A | 6F | 5D | 6F | FA | 35 | 44 | FF | |
| 40 | OUT (N=0x8), A | 6F | 5D | 6F | FA | 35 | 44 | FF | "--ll-.Wo-ld" |
| 41 | OUT (N=0x5), A | 6F | 5D | 6F | FA | 35 | 44 | FF | "--llo.Wo-ld" |
| 42 | SBC A, A | 0 | 5D | 6F | FA | 35 | 44 | FF | |
| 43 | ADD A, n=0x48 | 48 | 5D | 6F | FA | 35 | 44 | FF | |
| 44 | SET 6, E | 48 | 5D | 6F | FA | 75 | 44 | FF | |
| 45 | SLA E | 48 | 5D | 6F | FA | EA | 44 | FF | |
| 46 | OUT (N=0x1), A | 48 | 5D | 6F | FA | EA | 44 | FF | "H-llo.Wo-ld" |
| 47 | LD A, 0x39 | 39 | 5D | 6F | FA | EA | 44 | FF | |
| 48 | LD HL, nn=0x124 | 39 | 5D | 6F | FA | EA | 1 | 24 | |
| 49 | SLA A | 72 | 5D | 6F | FA | EA | 1 | 24 | |
| 50 | OUT (N=0x9), A | 72 | 5D | 6F | FA | EA | 1 | 24 | "H-llo.World" |
| 51 | IN C, (C=Ox5D6F) | 72 | 5D | 3A | FA | EA | 1 | 24 | |
| 52 | SRA E | 72 | 5D | 3A | FA | F5 | 1 | 24 | |
| 53 | LD B to C | 72 | 5D | 5D | FA | F5 | 1 | 24 | |
| 54 | ADD A,E | 72 | 5D | 5D | FA | F5 | 1 | 24 | |
| 55 | RES 1, A | 65 | 5D | 5D | FA | F5 | 1 | 24 | |
| 56 | RES 4, A | 65 | 5D | 5D | FA | F5 | 1 | 24 | |

```
57  OUT   (N=0x2), A        65  5D  5D  FA  F5  1    24  "Hello.World"
58  HALT@ Inst 58           65  5D  5D  FA  F5  1    24
(Note: IX register was 0x14D for all 58 instructions.)
```

Figure 6.  Code from Hello World Run.

Clearly, there is little that is elegant about the code that evolved in this run. None the less, the agent is functional and robust.

## 4 EXPERIMENTAL RESULTS

Performing GP with a large number of machine language operators and memory, as reported herein, calls into question the applicability of prior traditional GP results. For this reason, experiments were performed in order to determine how the GEMS GP computational requirements are effected by the:

- Pool size.
- Length of the target string.
- Mutation rate
- Parent selection process.

The remainder of this section discusses the experiments and their results.

### 4.1 PROTOCOL

A standard protocol was used in testing the effects of various parameters. Firstly, the same GEMS version of software was used for each GP run. Next, in order to get statistically meaningful data for each set of parameters, multiple runs -- called GPSets -- were made using those parameters. Some of the GEMS GP runs required an extensive amount of time, e.g., several days on a Sun SPARC 20 for a single run of the HWP. [Kinnear 1994] recommends twenty runs for meaningful statistical results - more if the GPSets results aren't substantially different. As a necessary comprise, the author elected to use GPSets comprised of sixteen runs.  Where more or less than 16 runs formed a GPSet, it is noted in the data below.

Each run of a GPSet began with the generation of a new random pool. Additionally, a new and different random number generator seed was used for each run.

GPSets were normally started as background processes on a single machine. As the GP runs progressed, they periodically recorded pool statistics to a disk file. When each GP run reached its goal, it would spawn the next process. For completed GP runs, the pool statistics files were examined to obtain the desired figures concerning the run. In this way, a GPSet could be processed with a minimum of external control.

Not all GPSets (or even GP runs within a GPSet) were necessarily run on the same machine or machine type. Various Sun SPARC machines, from SPARC 1's to SPARC 20's were used.

Three benchmark points were used for each GP run. These are the number of spawnings at which:

1. A pool member first put out the correct string, hereafter called “first seen”.
2. Correctly performing agents were considered stable in the pool, hereafter called “stable-in-pool”.
3. The shortest agent in the pool was of less than 100 instructions, referred to herein as “<100 Insts”.

The resolution of the results was always the rate at which the pool statistics were sampled and written to the disk. This value varied from 100 spawnings (for short strings) to 1000 spawnings for long strings.

It is important to note that the appearance of a properly functioning agent in the pool can occur as a spurious event. In fact, pool members can produce the correct output string numerous times and then fail to be able to do it again correctly. For example, consider the following possible code segment:

```
    DEC (IX+11)     /* Decrement content of memory
                       location with  address = Contents
                       of IX register+11. If result is
                       zero, set Zero Flag   */
    LD   A, 6B      /* Load A register with ASCII for
                       'k' */
    JP NZ, nn       /* Jump to instruction nn if Zero
                       Flag not set */
    HALT            /* End program execution  */
    • • •           /* Other miscellaneous code  */
nn  OUT 1, A        /* Output A register to port 1 */
    • • •           /* Code to print the remainder
                      of the string */
```

The DEC instruction may be a code remnant from the original randomly generated program and totally unrelated to the current performance of the program. None the less, if this agent is run enough times the DEC instruction will, at some time, result in a zero in the memory location, the Zero Flag will be set, the jump instruction (JP NZ, nn) will not execute and the agent will halt pre-maturely.

While the above is a trivial example, examination of the convoluted code of Figure 6 should convince the reader of the potential for latent disasters.

It is for this reason, as much as esthetics, that code should be evolved to the minimum size practical. While this won't necessarily guarantee unerring performance, it will raise confidence in the product.

GP runs were normally not terminated until they had concluded all three benchmarks. On rare occasions, a run would pass benchmarks 1 and 2, but not reach 3. Sometimes, GP runs were manually terminated if they got to 2-3 million spawnings without any sign of shortening. In other cases, the computer would (for a variety of reasons) shut down or be re-booted. Whenever this happened, it is noted below, but never effected more than one run of a GPSet and only the benchmark 3 result.

## 4.2 POOL SIZE TEST AND RESULTS

In traditional GP processing, pool size represents a compromise between computational requirements and sufficient genetic diversity for having a good chance at reaching the problem solution. [Kinnear 1994] states that the population size must be larger than a critical minimum size in order to generate a solution reliably and that this size is different for each problem. Similar conditions should apply to the GEMS process.

GEMS tests were conducted with pool sizes of 150, 250, 500, 1000, 1500 and 2000 members and using "Hello" as the target string. All the tests involved GPSets of sixteen runs, except for the 150 member pool which had 31 runs and the 500 member pool which had 15 runs.

Figure 7 shows the results for the number of spawnings required to get programs of less than 100 instructions. The curves for the “first seen” and “stable-in-pool” benchmarks are similarly shaped.

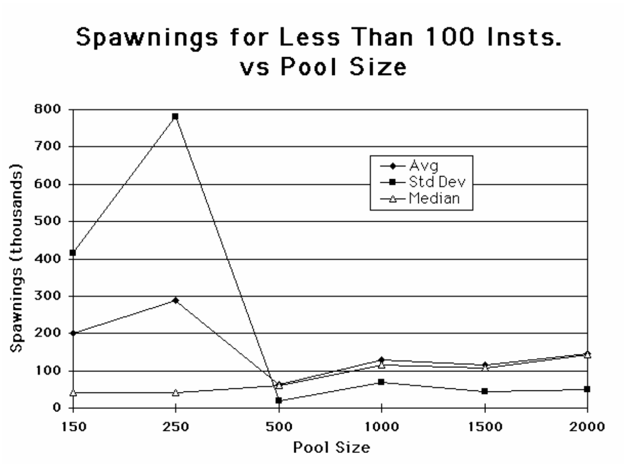


Figure 7. Spawnings to Generate “Hello” vs. Pool Size

The salient result shown in Figure 7 is that for pools of less than 500, the standard deviation increases significantly. This reflects the frequency and sizes of the outliers. For pools over 500, the standard deviation levels out and the median closely approaches the average, i.e., the GP process is more well-behaved. These results confirm that GEMS follows Kinnear's statement on pool size effects as given above.

Figure 8 illustrates the median values for the three benchmarks. The number of spawnings needed to get a solution stable in the pool appears to be rising at a linear rate or slightly faster. The cause of this rise is discussed in Section 5.

## 4.3 COMPUTATIONS VS. STRING LENGTH TESTS AND RESULTS

It should not be unexpected that as a GP problem becomes more difficult, the computations to obtain its solution increases. The rate of increase is an important issue. If the GEMS computations rise too rapidly with problem difficulty, e.g., exponentially or combinatorially, the prospect for GEMS will be poor or -- at the least -- more restricted or difficult to advance.

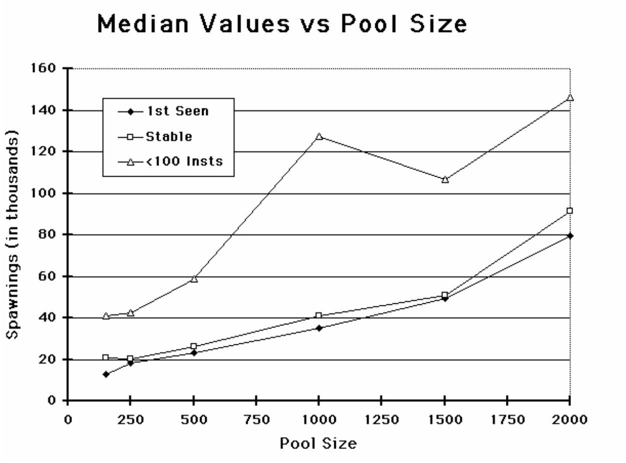


Figure 8. Median “First seen”, “Stable-in-Pool” and “<100 Insts” Values vs. Pool Size for “Hello” String.

The length of the string to be generated by the GEMS process undoubtedly affects problem difficulty. An indication of the growth in problem difficulty with string length can be gained by examining what would be required to solve this problem via a random process. Specifically, how does the probability of obtaining the desired string via random value selection vary with string length. To examine this, let n be the number of characters in the string to be generated. Assume that the string is to be produced by randomly drawing (with replacement) from a uniformly distributed pool of valid alpha-numeric characters. Assume this pool is composed of the fifty-two upper and lower case English characters, the ten digits and the space character for a total of sixty-three characters. The probability of drawing the correct characters in one set of n draws is 1 in $63^n$. If the string length is increased to m=(n+k), the probability becomes 1 in $63^{(n+k)}$. Thus, the problem increases in difficulty by a factor of $63^k$, i.e., exponentially with string length. Moreover, if all printable ASCII characters are included, the problem becomes about $2^k$ times harder yet. Clearly, the GEMS process must be much better behaved than this to be viable.

The tests conducted involved the generation of strings of various lengths from 2 through 11 characters. The specific strings (including case) are: to, Ron, Marc, Hello, stored, uniform, San Diego and Hello World.

In each case, a GPSet of sixteen GP runs was performed and the three benchmark points previously defined were measured. These runs all used the following conditions:

- Pool size of 1500 members
- Uniform parent selection
- 20% mutation rate.

Figure 9 shows a semi-logarithmic plot of the spawn count median values for "first seen", "stable in pool" and "< 100 insts" bench marks for each string length. The dashed line in this figure shows a plot of the equation $400c^3+10000$, where c is the string length in characters. This latter curve is presented in order to give some indication of the growth rate over the range of the string lengths. It indicates that the problem difficulty, over the range of the test and in terms of required spawns, while increasing rapidly, does not appear to be combinatorial or exponential.

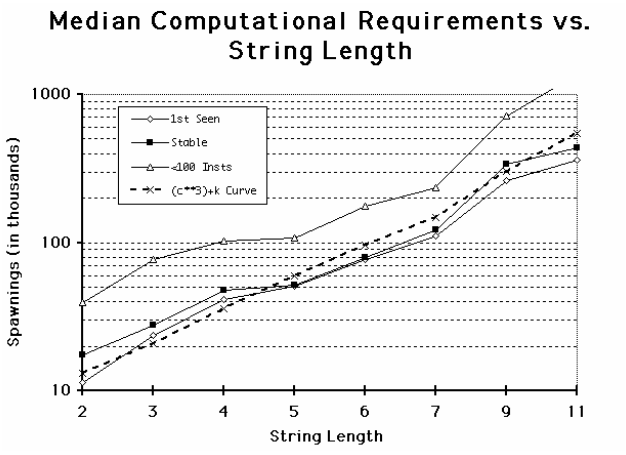


Figure 9. Computational Requirements for Various String Lengths ( Pool Size = 1500, uniform parent selection and 20% mutation rate.)

## 4.5 PARENT SELECTION AND MUTATION TESTS AND RESULTS

A limited amount of testing was performed to determine if the parent selection criteria and mutation rate had any significant influence on the computations required for string generation. Due to the potential for interaction between these factors, they are considered together.

The GEMS process allows the user to control how parents are selected for breeding from the pool. Three selection processes are available:

- Uniform selection where each pool member has an equal probability for selection irrespective of its fitness ranking.
- Selection probability based upon the member's fitness rank to the 1.5 power.
- Selection probability based upon the square of the member's fitness rank.

Early GEMS experiments involved a set of runs to examine the effects of parent selection and mutation. The tests had a target string of “Hello”, employed GPSets of sixteen runs, measured the standard benchmarks and used a pool size of 150. The results are shown in Figure 10 as the median values for each of the benchmarks vs. the test parameters. In this case, a uniform selection and 20% mutation rate (labeled “U-20%”) appears to be substantially better than the other two parameter choices of: parent selection based on rank to the 1.5 power and 20% mutation (1.5P-20%); or, uniform parent selection and 2.5% mutation (U-2.5%). It was noted that the high variance associated with this set of runs made it unwise to dogmatically state a conclusion based upon these results. None the less, because it could not be demonstrated that uniform parent selection and a 20% mutation rate was any worse than the other options, and might well be better, these factors were used in most of the testing discussed on the foregoing sections.

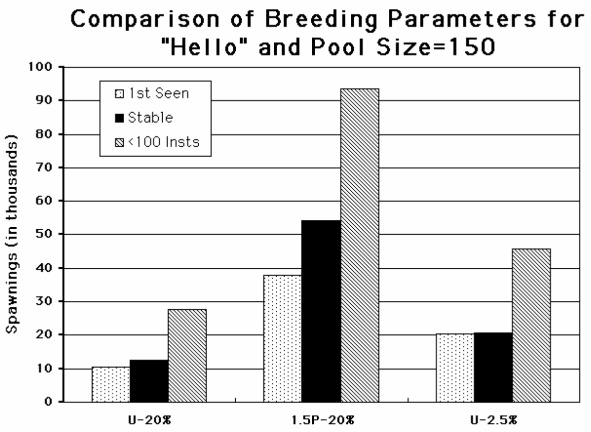


Figure 10. Benchmarks Compared for Mutation Rates (20% vs. 2.5%) and Parent Selection Criteria (uniform vs. 1.5 Power of rank) for “Hello” and pool size of 150.

It should be noted that the 20% mutation rate may not be as large as it sounds considering the way in which GEMS performs mutations. To wit, 20% of the time an off-spring is selected for mutation. During mutation, a random amount (up to 20%) of the off-spring’s instructions are randomly regenerated. In that the latter 20% is uniformly distributed, on the average only 10% of an off-spring’s code is changed. In the net, then, the mutation amounts to an average of 2% of the population contents. An exact comparison of GEMS mutation to traditional GP mutation is made difficult by the differences in sizes of the instruction/operation sets.

Further experiments were conducted on mutation and parent selection factors using the string "Marc" and a pool of 1500 members. Again GPSets of sixteen runs were used. The results are summarized in Figure 11 which shows the median benchmark values for each of the parameter sets. These results appear to indicate that, for the values chosen, neither the mutation rate nor the parent selection criteria produce any significant computational advantages in obtaining stable solutions in the pool. If anything, use of parent selection based upon rank to the 1.5 power and 20% mutation appears slightly better.

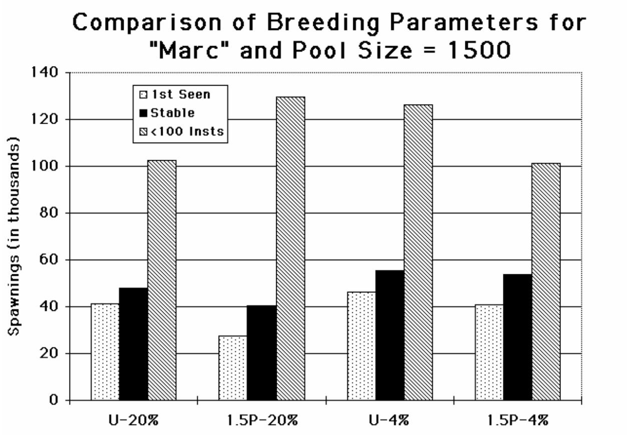


Figure 11. Benchmarks Comparison for Mutation Rates (20% vs. 4%) and Parent Selection Criteria (uniform vs. 1.5 Power of ranking) for “Marc” and pool size of 1500.

## 5 DISCUSSION

### 5.1 BASIC CONSIDERATIONS

The experimental results given in Section 4 demonstrate that a GP process employing a large set of general purpose operators along with a relatively large memory can be used to generate modest programs. While this contradicts conventional views held on the matter, the author has a potential explanation that involves the problem chosen, the way GEMS is implemented and the flexibility of the GEMS operator set.

The explanation for why GEMS produces results is in part due to the fact that OUT instructions of the type needed to get some fitness score, albeit small, are not extraordinarily uncommon. To understand this, consider the fact that the GEMS implementation has ten output instructions. Given this number, if the instructions of the new pools and mutations were all equally probable, then 1.5% of the instructions (10 of the 660 implemented) will send some output to some port.

The author recognized early on that the input and output instructions were critical to the viability of any software generated by GEMS. This stems from the fact that the only way of interacting with the GEMS agents is via input and output instructions. There is no other visibility into the GEMS pool members’ behavior. As such, it was decided that in new random pool generation 5% of the instructions would be specifically and equiprobably generated as input or output instructions. The remaining 95% contain the normal 1.5% output instructions. Thus, output instructions represent about 3.9% of all randomly generated instructions in the new pools.

For nine of the ten output instruction types, the port addresses for an 11 character string represent 11 of the $2^{16}$ possible port values (or about 0.02%). The remaining instruction is "OUT (n), A", where n is in the range of 0 to 255. For this instruction the 11 ports represent 11 of the 256 possible values or about 4.3%. (This high value probably accounts for this instruction being the only OUT instruction in the code of Figure 6. Other OUT instructions have been observed in other GEMS generated programs, so they do occur). In the net, the probability of a randomly selected "OUT" instruction writing to ports 1 through 11 is about 0.0045.

A pool of 1500 members, each having 255 instructions, has a total of 382,500 instructions. On average 14,917 of these are output instructions of which, on average, 67 write to a desired output port.

Writes to the ports are always 8 bit values. Nearly half of the 256 possible outputs are printable ASCII characters. Thus, at initiation of the GP process, an average of 30 output instructions are available in the pool that can generate a non-zero fitness value. These then become the seeds for the evolution of the successful pool members.

The second element that may account for the GEMS process' evolutionary successes is the low level, flexibility and redundancy of the GEMS operators. Figure 6 shows the code that evolved to generate "Hello World". It illustrates some very convoluted ways of getting the A-register loaded with the required values to be outputted. This is possible because the GEMS operators are numerous, diverse and low level in operation. Thus, achieving a desired register value can derive from the process of: accessing register or memory contents, adding, subtracting, bit shifting, bit setting or re-setting, logical ANDing or ORing, incrementing or decrementing or any suitable combination of the above.

The implication of the previous paragraph is that in lieu of a single instruction or set of instructions, there exists many functionally equivalent code blocks that can achieve the requisite goal of getting a needed value in the A (or other output) register. Moreover, these code blocks can be inter-related in what they do in a manner no human programmer would dream of doing (even if that person could conceive of and implement such an approach).

It is impractical to enumerate the ways that the GEMS operators can achieve any particular complex function. So it is open to speculation as to the probability of achieving that function. Yet, the results of this paper imply that, for the string generation problem, there is a sufficiently large number of ways of achieving the desired register contents to make the problem tractable for GP.

## 5.2 MEMORY USAGE

It is clear from the code of Figure 6 that in solving the string generation problem GEMS employed the available memory, albeit awkwardly by human programming standards. The relatively large amount of memory, vis-à-vis the needs of this problem, did not preclude obtaining a solution program. In fact, it may have enhanced the process.

The string generation process requires the binary ASCII values for each character to be outputted. With 255 memory locations initially filled with random 8 bit values, it is highly probable that the needed values are somewhere in memory. It is nearly certain that the values can be obtained by some manipulation of one or more of the values in memory.

## 5.3 PROGRAM LENGTH OPTIMIZATION

Section 4.1 discussed the importance of reducing the length of a GEMS agent product for purposes of ensuring enduring proper behavior. Figures 8 and 9 show that as much computational power was required to get a 100 instruction agent as was required to obtain a stable agent in the pool. Sometimes 2-3 times as much computational power was required. Not infrequently, a GEMS run did not reach this benchmark in 2-3 million spawnings and was thus terminated.

The author conjectures that the problem with breeding for shorter agents resulted from the use of sequential evolutions, i.e., first breed for the correct output, then breed for shorter programs. This conjecture is based upon the following considerations. First, GEMS has only one instruction that can be used to shorten an agent -- the HALT instruction. The HALT instruction is one of 660. Thus, the initial randomly generated pool members of 255 instructions each contain, on the average, approximately 0.4 HALT instructions. During the process of breeding the correct output, it is conceivable that most of the HALT instructions in the pool could, due to their lack of utility and even potential detriment, be bred out or at the least relegated to code locations that are not reached. This would account for the difficulties with breeding short agents.

The author plans to investigate this matter further. One solution is to increase the frequency of the mutational generation of HALT instructions once agents generating the proper output appear in the pool. Another solution is to attempt to breed shorter agents at the same time as the correct output is being bred. [Kinnear 1993] reports that this approach may even result in more efficient agent implementations. The author has encountered difficulties with this dual fitness approach. Specifically, the process tends to evolve short and poorly performing agents. Further work is needed in this area.

## 5.4 POOL SIZE EFFECTS

The size of the pool is an important determining factor in the computational requirements for achieving the breeding goal. On one hand, small pool sizes have a low mean computational requirement, i.e., spawnings, but a larger variance. This implies that the process can sometimes converge rapidly to the target agent, but at a significant risk that periodically it won't converge for a large number of spawnings.

With a large pool, there appears to be a smaller variance of the computational requirements to achieve a target agent, but with a higher mean as illustrated by Figure 8.

In order to explain the growth in mean computations required for a larger pool size, especially with the GEMS asynchronous breeding process, the author proposes the following hypothesis: The probability that an off-spring will have a fitness improvement of $\partial$ over its best parent's fitness decreases as $\partial$ increases. This hypothesis is not unreasonable if one accepts the concept -- discussed in the next paragraph -- of agents being composed, to some extent, of functionality elements and related fitness.

The breeding of GEMS, as shown in Figure 4, takes code blocks from two parents. Each code block represents some amount of functionality which reflects its contribution to the fitness of the parent from which the code block originated. When the two code blocks are combined, their respective functionality's can complement, overlap or undo each other. If they complement each other more than they undo, the resultant functionality, and thus fitness, of the off-spring is greater than either parent. The amount of new functionality, or added fitness, of the off-spring over its best parent would normally be small. Large jumps in fitness improvement are unlikely. The reasons for this are that, on the average:

- Some percentage of the stronger parent's functionality (given to the off-spring) is being deleted to make room for functionality from the weaker parent.
- Up to 50% of the weaker parent's functionality is being given to the off-spring.
- The incestuous nature of the cross breeding within the pool mitigates against any radically different (or better) members, so the spectrum of functionality of the pool members is somewhat homogeneous.
- The probability of multiple "micro-functions" complementing each other decreases as the number of "micro-functions" increases. (Here "micro-functions" are the pieces of code that form functions and are such small program elements as register values, bit shifts, useful jumps, etc.)

Clearly, there will be those serendipitous occasions when the parents' code blocks complement each other quite well and generate a relatively large improvement in the off-spring's fitness. This though should be expected to be the rare exception.

Assuming that the above hypothesis is true, then the rate of improvement of the pool members is controlled by the density of the most fit members and thus, indirectly, the average of the pool members' fitnesses.

Consider the initial pool. It contains only poorly fit members. The selection of these early pool members for breeding will occasionally produce an off-spring of slightly improved fitness vis-à-vis the parents and per the above hypothesis. At first, the pool will contain very few improved members. The odds of two of these improved members being selected to breed, and thereby potentially generate an even more fit off-spring, is inversely proportional to the pool size. As such, and most important, *breedings in number proportional to the pool size must occur to raise the density of more fit pool members so that these can breed to generate yet more fit off-spring*. The net result is that the number of spawnings to reach the target agents increases with pool size irrespective of any other factor.

Another way to view this is that in the aggregate the pool members' individual fitnesses ride on a tide of the average fitness of all of the pool members with few being very far above that average.

## 6 CONCLUSIONS

Agents of simple functionality can be generated with a GP process that involves a large number of ML operators and memory as implemented in GEMS. The growth in computational requirements with problem complexity is high, but not formidably so.

Further investigations into ML agent generation are warranted. Specifically, GEMS should be applied to more difficult problems, e.g., sorting numbers. Additional research should be conducted to identify methods for improving the computational efficiency of ML agent generation. These improvements could come from changes to the internal breeding process implemented within GEMS and procedural approaches to the GP process.

GP processes are not smarter than humans. They are simply more patient and tenacious in exploiting the opportunities made available to them. They exhibit these characteristics while churning away 24 hours a day at megaflop rates. Therein lies their power to solve problems.

**Bibliography**

Altenberg, Lee (1994) "The Evolution of Evolvability in Genetic Programming." In *Advances in Genetic Programming*, K. Kinnear, Ed., Cambridge, MA: MIT Press.

Angeline, Peter (1993) "Evolutionary Algorithms and Emergent Intelligence." PhD Thesis, Computer Sciences Department, Ohio State University.

Angeline, Peter and Pollack, Jordan B. (1994) "Coevolving High-Level Representations." In *Artificial Life III. Proceedings of the Workshop in Artificial Life Held June 1993 in Sante Fe, New Mexico*, Christopher G. Langton, Ed., Reading, MA: Addison-Wesley.

Kinnear Jr., Kenneth E. (1991) "Generality and Difficulty in Genetic Programming: Evolving a Sort." in *Proceedings of the Fifth International Conference on Genetic Algorithms*, S. Forrest, Ed., San Mateo, CA; Morgan Kaufmann.

Kinnear Jr., Kenneth E. (1993) "Generality and Difficulty in Genetic Programming: Evolving a Sort." *In Proceedings of the Fifth International Conference on Genetic Algorithms*, S. Forrest, Ed., San Mateo, CA: Morgan Kaufmann.

Kinnear Jr., Kenneth E. (1994) "A Perspective on the Work in this Book." In *Advances in Genetic Programming*, K. Kinnear, Ed., Cambridge, MA: MIT Press.

Koza, John R. (1992a) "Genetic Programming: On the Programming of Computers by means of Natural Selection.", Cambridge, MA, MIT Press.

Koza, John R. (1992b) "The Genetic Programming Paradigm: Genetically Breeding Populations of Computer Programs to Solve Problems." *In Dynamic, Genetic and*

*Chaotic Programming: the Sixth-Generation*, Branko Soucek, Ed. New York, NY, John Wiley & Sons.

Koza, John R. (1994) “Introduction to Genetic Programming”, In *Advances in Genetic Programming*, K. Kinnear, Ed., Cambridge, MA: MIT Press.

Nordin, Peter. (1994) “A Compiling Genetic Programming System that Directly Manipulates the Machine Code.” In *Advances in Genetic Programming*, K. Kinnear, Ed., Cambridge, MA: MIT Press.

Reynolds, C. W. (1993) "An Evolved, Vision-Based Behavioral Model of Coordinated Group Motion," in *From Animals to Animats 2: Proceedings of the Second International Conference on Simulation of Adaptive Behavior*, J. A. Meyer, H. L. Roitblat and S. W. Wilson, Eds. Cambridge, MA: MIT Press.

Rosca, J. P. and Ballard, D. H. (1994) “Learning by Adapting Representations in Genetic Programming.” In *Proceedings of the 1994 IEEE World Congress on Computational Intelligence*, IEEE Press.

Teller, Astro (1993) “Learning Mental Models”, in *Proceedings of the Fifth Workshop on Neural Networks: An International Conference on Computational Intelligence: Neural Networks, Fuzzy Systems, Evolutionary Programming and Virtual Reality*.

Teller, Astro (1994) "Turing Completeness in the Language of Genetic Programming with Indexed Memory." in *Proceedings of the 1994 IEEE World Congress on Computational Intelligence*, IEEE Press.

Zaks, Rodnay [1980] “How To Program The Z80.” Berkeley, CA: SYBEX Inc.