Fast and Flexible Genetic Algorithm Processor

Pourya Hoseini, Abdollah Khoei, Khayrollah Hadidi, and Sajjad Moshfe
Microelectronics Research Laboratory
Urmia University
Urmia, Iran
st_p.hoseini@urmia.ac.ir ; a.khoei@urmia.ac.ir ; kh.hadidi@urmia.ac.ir ; st_s.moshfe@urmia.ac.ir

Abstract—**In this paper a generic genetic algorithm processor (GAP) with high flexibility in parameter tuning is introduced. The proposed processor utilizes pipeline structure to have low processing time. In order to further increase in the speed, genetic population has been duplicated, one for replacement stage of genetic algorithm (GA) and another for selection phase. Additionally, parallel processing method in the selection stage boosts GA processor's speed.**

The proposed GA has been designed so that it can work in online controlling circumstances. It supports for constraints in search space and changing environments. Also, a large bit number of chromosomes can be achieved by connecting the proposed 32-bit processors to work as one *n*-bit chip. Ability to work with two fitness function chips, supporting pipelined fitness functions, and capability of distributed processing are other factors that increase the speed in our design.

Index Terms—**Hardware GA, Genetic Algorithm, Fast GA Processor, Two Population Scheme**

I. INTRODUCTION

Genetic Algorithm is a stochastic global search method which is based on the survival of the fittest and is one of the major branches in signal processing. It generates a random population of solutions and then selects individuals (chromosomes) in which fitter ones have more probability of selection. The algorithm performs GA operations on the selected ones and produces new offspring which replace some individuals. This operation continues until a predefined condition has been reached so computational complexity of genetic algorithm is high. This makes software based genetic algorithms unsuitable for real time applications. Another notorious problem of GA is the need for tuning of many parameters. Considering these problems, hardware implementation of GA can be a solution for its computational cost whilst preserving flexibility in parameter tuning.

In [1] a generic GA processor with adaptive strategy for crossover has been proposed. Distributed processing and working with two fitness units are presented in [2]. In [3] a soft intellectual property for GA has been proposed. User selects processor's parameters and then a software application synthesizes a chip according to them. Disadvantage of this method is that after design or fabrication user cannot change parameters of the GA. Shackleford et al. [4] realized a survival based steady-state GA on FPGA and tested it with pipelined fitness functions. In [5] a general purpose soft intellectual property for GA with emphasis on flexibility is presented. A very programmable GA engine which is an application-specific instruction-set processor (ASIP) is presented in [6].

Among the works existing in the literature some suffer from pipeline stalls (roulette wheel selection and generational genetic algorithm) as described in [7]. Non flexibility in GA parameters is another problem in many previous works. Also application specific processors introduced in many papers, lose generality. It is obvious that for a generic GAP it should be capable of co-working with fitness units with various functionalities. The ability of working in online controlling environments is another concern in majority of the GA processors.

To have no pipeline stalls we have implemented a steady state GA with tournament operator as selection scheme. Our GAP communicates with external chips/computers in order to compute the fitness value of the chromosomes; hence it is a generic GA processor. The user can set many parameters of the GA such as mutation rate, crossover rate, mutation type, crossover type, tournament size, population size, GA iterations and so on. In this manner our GAP possess high flexibility. Supporting online control circumstances with a Discard pin for improper chromosomes, in problems with constraints, is another feature of the proposed GAP. It also supports changing environments with random immigrant strategy [8] for the steady state GA. The processor also can be restarted as a change in the control environment detected. We have chosen delete-the-oldest scheme for replacement stage of the GA as it performs better than delete-the-worst scheme in non-stationary systems [9] and is more suitable for hardware implementation.

We classify the effective factors on the speed of the proposed GA processor to five groups:

1- Chromosomes' bit string
2- Master-slave processing (multi fitness units work with one GAP)
3- Distributed (parallel) processing
4- Pipelined fitness unit (FU)
5- Employ boosting techniques in the body of GAP
    a. Pipeline structure
    b. Two population scheme
    c. Parallel processing of selection module.

All the above factors are supported in our GAP, but we only focus on the last one (circuit boosting techniques employed in the GAP) in this paper.

II. THE PROPOSED GENETIC ALGORITHM PROCESSOR

Hardware implementation of GA processors usually enjoys pipelining of their modules. In our design, we break the normal follow of the GA into five modules and pipeline them. Thus it adds a latency of four GA iterations. Fig. 1 illustrates the block diagram of the proposed GAP. After initialization, the normal follow of GA starts which consists of selection, crossover & mutation, delay, fitness estimation, and replacement modules. Fitness estimation module is responsible for communicating with auxiliary fitness units through a two-way handshake interface. Emigration and immigration modules play roll in distributed processing. Best found register saves the best individual and at the end sends it to finalization module to send out of the processor. If enabled by user random immigrants module replaces some chromosomes by new randomly created ones in the specified frequency that set by user.

Pipeline structure shown in Fig. 1 reduces the duration of every iteration to the time taken by the slowest module. Note that the speed of the fitness estimation module depends on auxiliary FUs. Also in the subsection B we describe the reason of placing a delay in the pipeline structure. In the following we present detailed explanations about the major modules and the utilized speed up topologies.

A. Replacement Module

Replacement module has responsibility of replacing newly generated offspring in the population. As said the replacement scheme is delete-the-oldest which means the oldest individuals will perish and new ones replace them. Fig. 2 shows the replacement module. At the start of the module a control signal, which comes from the master control module, orders the clocked inverters to load the offspring (and also immigrant chromosomes come from other GAP chips) and their fitness values into the pre-loadable register arrays. At the same time the ready flag of each offspring is loaded into its register. Ready flag specifies that each offspring should be written in the memory or not (for example in the case of a discarded chromosome it prevents writing that one).

After that, replacement module's clock starts and the register arrays send out the chromosomes and their fitness values serially to the memory core and to the best found register. A 32-bit digital comparator compares the new offspring's fitness with the best found solution's fitness and when the fitness of the offspring is greater than the previous best found fitness it takes it and its chromosome as the new best found solution. Right side of the Fig. 2 shows that if the ready flag of an offspring be zero it cannot replace the best found register in any conditions.

At the bottom of the Fig. 2 the two counters are building blocks of the delete-the-oldest scheme. They mark the address of the oldest individuals. Each time an offspring is written to the oldest chromosome (at the address specified by the counters) in the memory, the circuit clocks the counter in the next cycle and increases the address of the oldest individual by one. So the next address in the memory will be perished in the following replacement. Note that in every run of the replacement module only one counter counts and in the next run the other one counts. This is due to the two population scheme introduced in the following section.

B. Memory Core

We have used 16 Kbit RAMs for storing each population. But a problem arises in the pipeline structure when both replacement and selection modules want to make use of the RAM, one for reading from it and one for writing in it. Therefore, maintaining two copies of the population, one for reading and one for writing, seems a good idea. But having two exact populations requires complicated and time consuming procedure (approximately doubles the execution time of replacement module) because RAMs cannot read and write at the same time. Hence we designed a two population scheme in which, an offspring only replaces in one population. Two populations periodically switch between replacement and selection modules. For example if in the current iteration of GA, first population is being replaced and the other population is being selected, in the next GA's iteration, the first will be used for selection module and the secondary for replacement module. Therefore, each population of chromosomes has participants that are not in the other population. But this does not mean that we have two distinct and parallel populations, because new chromosomes which come out from one population goes (replaced) in another one. This is accomplished by a delay module that shown in Fig. 1. Delay module causes that new parents selected from one population replace in the other population. Our experiments show that this method does not decrease the GA performance but increases it. Calculations show that this method increases the speed of the GAP by 34%. This is in consequence of that replacement and selection modules can now work concurrently. The use of two counters is also because of the two mentioned populations. For each population a counter holds the oldest chromosome's memory address.

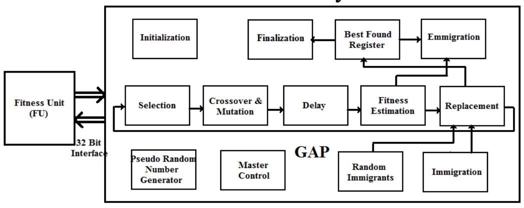

Figure 1. Block diagram of the proposed GA processor

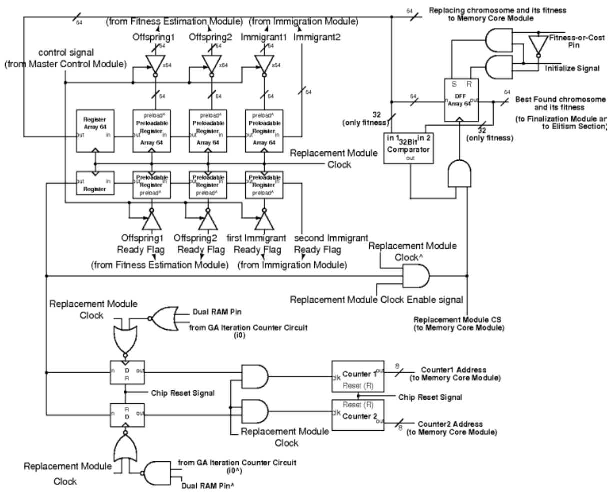
(from Fitness Estimation Module)
(from Immigration Module)
Offspring1
Offspring2
Immigrant1
Immigrant2
control signal
(from Master Control Module)
Replacing chromosome and its fitness to Memory Core Module
Fitness-or-Cost Pin
Initialize Signal
Register Array 64
Preloadable Register Array 64
32 (only fitness)
32Bit Comparator
Best Found chromosome and its fitness (to Finalization Module and to Elitism Section)
Replacement Module Clock
Offspring1 Ready Flag
Offspring2 Ready Flag
first Immigrant Ready Flag
second Immigrant Ready Flag
(from Fitness Estimation Module)
(from Immigration Module)
Replacement Module Clock Enable signal
Replacement Module CS (to Memory Core Module)
Dual RAM Pin
from GA Iteration Counter Circuit (I0)
Chip Reset Signal
Counter 1
Counter 2
Counter1 Address (to Memory Core Module)
Counter2 Address (to Memory Core Module)
Dual RAM Pin^

Figure 2. Replacement module

So far, we should have two RAMs. But as demonstrated in Fig. 3 we have twice. As mentioned before we have used steady state GA that selects two parents in the same iteration. Hence, two simultaneous reading operations are needed in the selection module. The memory core shown in Fig. 3 replaces two copies of chromosomes in two RAMs and then in the next iteration uses one RAM for selection of first parent and the other for selection of second parent.

So far we have quadruplicated the memory space, but addition of three 16Kbit RAMs brings 100% boost in the selection module's speed and 34% faster run of overall GAP due to simultaneous run of replacement and selection modules. The memory (RAMs) is tightly coupled with the rest of the system and receives addresses at least 200ps prior to its operation. In this case RAMs utmost take 600ps to operate.

The top part of Fig. 3 switches the left and right RAMs for selection and replacement in each iteration. Note that the proposed design supports the normal follow of GA (i.e. replacement module writes in RAMs after reading operations in the selection module have been done). When the Dual RAM Pin is high only first counter is used and memory core looks for completion of the selection module to switch between two module's clocks. Note that in the initialization phase all the RAMs are in the write mode and the initialization clock (CS) goes to RAMs as illustrated at the top of Fig. 3.

The address changer circuitry in the bottom of Fig. 3 are for switch between counters' addresses and random addresses come from pseudo random number generator (PRNG) which is needed for the selection module. We chose linear cellular automata (LCA) method for implementing PRNG. Also as shown in the bottom circuitry of Fig. 3 the memory core can import addresses from address bus of the chip to use in the selection phase. It is necessary for connecting the GAP chips which is out of scope of this paper. In the initialization phase, all the RAMs receive the first counter's address since in this phase the two populations grow exactly (selection module is not running in the initialization phase to conflict with replacement). After the completion of initialization module, one of the two counters' addresses passes to the two left/right side RAMs. Also note that, the size of addresses proceed to the RAMs, alters based on the population size that has been set by user, as illustrated in the bottom frame of Fig. 3.

C. *Selection Module*

Fig. 4 presents the overall schema of the selection module. Selection module has two identical circuits for choosing two parents from the memory. Contestant chromosomes which come from RAMs go to elitist circuit. The output of the elitist circuit is whether a randomly selected individual that comes from the RAM or the best found chromosome. At first, the "first contestant accept signal" loads the first contestant as the winner chromosome. In the advance of the selection phase, 32-bit digital comparators compare the winner one with the newly arrived contestants. In this way, when a fitter one (individual with higher fitness) is detected the 64-bit D flip-flop array saves the fitter chromosome and its fitness. The circuitry in the bottom of Fig. 4 is for sending selection addresses to other GAP chips (for connecting GAP chips and make them as one n-bit processor).

D. *Crossover & Mutation Module*

The crossover & mutation module is illustrated in Fig. 5. This module takes two 32 bit chromosomes and with a probability performs the crossover operation, then with a probability carries out the mutation operation. Uniform pattern generator in Fig. 5, produces a randomly valued bit string which probability of being one is related to the rate of crossover or mutation. When it produces random string for crossover cells it uses crossover rate and in the case of mutation cells it uses mutation rate. These random bits go to D flip-flops which then a control signal saves them. If the Uniform Crossover Pin be high, random bits then decide the switch between two bits of the parents in the crossover cell's multiplexer pair. Also, if the Uniform Mutation Pin be high, the output of the D flip-flop (random bit) can invert bit value of the offspring by the use of the XOR in mutation cell.

One point and two point crossovers also are supported in our design. If only one of the crossover type pins be high crossover cells perform one point crossover, and if both pins be high the two point crossover is performed. Crossover decoders indicated in top of the crossover cell in Fig. 5, are special 5-bit decoders which in its generated output, from a point all bits to the right are one or zero, and vice versa.

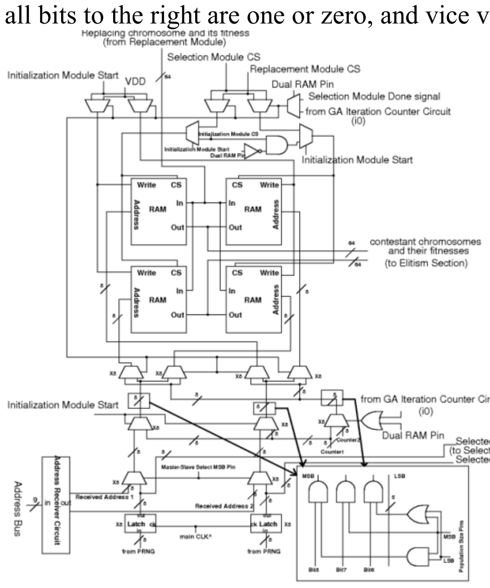
Replacing chromosome and its fitness (from Replacement Module)
Selection Module CS
Replacement Module CS
Initialization Module Start
VDD
Dual RAM Pin
Selection Module Done signal
from GA Iteration Counter Circuit (i0)
Initialization Module Start
Write
CS
RAM
Address
In
Out
contestant chromosomes and their fitnesses (to Elitism Section)
from GA Iteration Counter Circuit (i0)
Dual RAM Pin
Selected Address1 (to Selection Module)
Selected Address2
Master-Slave Select MSB Pin
Address Receiver Circuit
Address Bus
Received Address 1
Received Address 2
Latch
main CLK*
from PRNG
MSB
LSB
Population Size Pins

Figure 3. Memory core

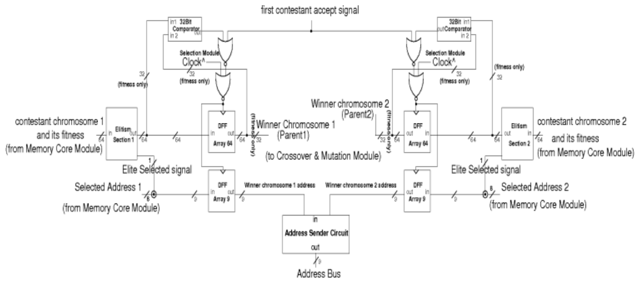
first contestant accept signal
32Bit Comparator
Selection Module Clock*
(fitness only)
contestant chromosome 1 and its fitness (from Memory Core Module)
Elitism Section 1
DFF Array 64
Winner Chromosome 1 (Parent1)
Winner chromosome 2 (Parent2)
(to Crossover & Mutation Module)
DFF Array 64
Elitism Section 2
contestant chromosome 2 and its fitness (from Memory Core Module)
Elite Selected signal
Selected Address 1 (from Memory Core Module)
DFF Array 9
Winner chromosome 1 address
Winner chromosome 2 address
Selected Address 2 (from Memory Core Module)
Address Sender Circuit
Address Bus

Figure 4. Selection module

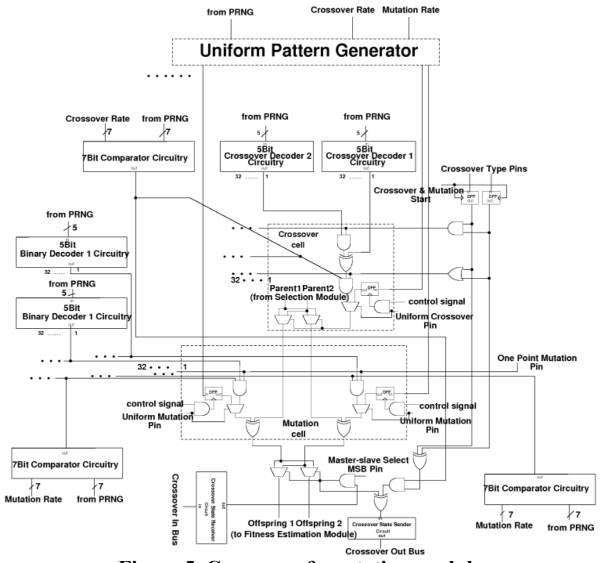
from PRNG
Crossover Rate
Mutation Rate
Uniform Pattern Generator
Crossover Rate
from PRNG
7Bit Comparator Circuitry
5Bit Crossover Decoder 2 Circuitry
5Bit Crossover Decoder 1 Circuitry
Crossover Type Pins
Crossover & Mutation Start
5Bit Binary Decoder 1 Circuitry
Crossover cell
Parent1 Parent2 (from Selection Module)
control signal
Uniform Crossover Pin
One Point Mutation Pin
control signal
Uniform Mutation Pin
Mutation cell
Master-slave Select MSB Pin
7Bit Comparator Circuitry
Mutation Rate
from PRNG
Crossover In Bus
Offspring 1 Offspring 2 (to Fitness Estimation Module)
Crossover State Sender Circuit
Crossover Out Bus

Figure 5. Crossover & mutation module

III. Simulation Results

The proposed GA processor supports all the 5 factors mentioned in section I. But we use only the fifth factor (i.e. speed up techniques designed in the body of the processor only) in this experiment as this factor has been explained in this paper. In this experiment a fitness unit test chip is connected to the proposed generic GAP chip as symbolically shown in Fig. 1. For the benchmark problem, we use a variation of bounded knapsack problem (which is one of the NP-complete problems), presented in (1).

$$Fitness = \sum_{i=9}^{11} i \times X_i + \sum_{i=17}^{19} i \times X_i + \sum_{i=25}^{27} i \times X_i + \sum_{i=41}^{43} i \times X_i + \sum_{i=57}^{59} i \times X_i \leq 450$$

$$Bound\colon\ X_i \in \{0,1,2,3\} \qquad (1)$$

In each sigma, X_i is the number of repetition of i. Because a fixed number of bits are used for each sigma, we added the following restriction:

$$In\ each\ sigma\colon\ X_{i_1} + X_{i_2} + X_{i_3} \leq 3 \qquad (2)$$

Detailed information about our test is shown in Table I. We have implemented the proposed GAP with CSMC's 0.18µm library and simulated the set of processor and its fitness unit with Hspice. For the fitness unit a discard pin has been designated to discard chromosomes generated in GAP which produce the fitness higher than 450 (as shown in (1)).

The results show that the processor converged and found the fitness of 446 as illustrated in Fig. 6. Total power dissipation of our 127-pin chip (considering all the modules of the GAP) was 3.29W. Total runtime of the proposed GA processor was 698ns (with 450MHz clock). Conventionally, results of the hardware GAs are compared with software counterparts. In order to compare the speed of our hardware GA with software, we implemented a normal GA with the characteristics shown in Table I. On a machine with 2.4 GHz processor it takes 23ms to complete the algorithm. So our algorithm is approximately 32951 times faster than the software. To the best of authors' knowledge this speed up over software is far greater than the previous reported works. Although there is no standard suite for comparison of the hardware GAs, we review some recent works in Table II. We nearly obtained the same speed up over software with different GA parameters except in the case of bit lengths greater than 32 which requires connection of GAP chips.

Table I. Test details

Chromosome bits	Fitness bits	Crossover type	Mutation type	Crossover rate	
30	9	Uniform	One point	100/128	
Mutation rate	**Tournament size**	**Iteration numbers**	**Elitism**	**Random Immigrants**	**Population size**
7/128	4	32	yes	yes	32

IV. Conclusions

We described a genetic algorithm processor with pipeline structure and two population scheme. As explained in the paper, the boosting method reduces computational time of the selection phase to the half. Supporting control environments and flexibility in GA's parameter tuning are other features of our GAP. Simulation results show that the proposed hardware GA is about 33000 times faster than software. Execution time of 698ns shows that this GA processor can be used in real time applications. To the best of authors' knowledge the proposed genetic algorithm processor is the fastest hardware GA ever.

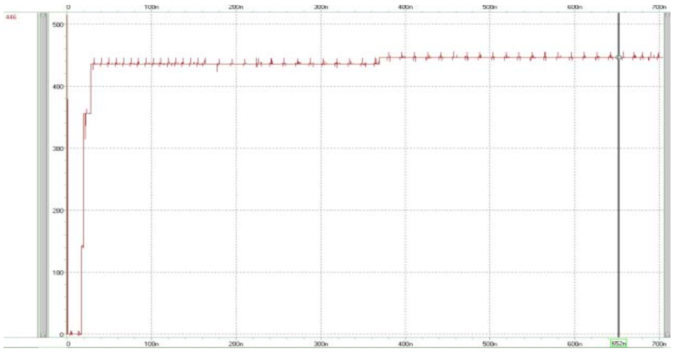

Figure 6. Convergence of the genetic algorithm processor

Table II. Comparison with other works

Work	Speedup over software	Software's host machine	Pipelined fitness function
[1]	10~20	Not defined	No
[3]	250	1.6 GHz	No
[4]	2200	100 MHz	Yes
[5]	5.16	PowerPC processor in the Xilinx Virtex2Pro	No
[6]	20	MIPS-I	No
Proposed	32951	2.4 GHz	Capable, but not in the test

References

[1] S. Wakabayashi, T. Koide, and K. Hatta, "GAA: A VLSI genetic algorithm accelerator with on-the-fly adaptation of crossover operators," IEEE International Symposium on Circuits and Systems, vol. 2, pp. 268–271, Monterey. CA, USA., May-June 1998.

[2] N. Yoshida, and T. Ysauoka, "Multi-GAP: Parallel and distributed genetic algorithms in VLSI," IEEE International Conference on Systems, Man, and Cybernetics, vol. 5, pp. 571–576, Tokyo, Japan, October 1999.

[3] P. Y. Chen, R. D. Chen, Y. P. Chang, L. S. Shieh, and H. A. Malki, "Hardware implementation of genetic algorithm," IEEE Transactions on Instrumentation and Measurement, vol. 57, no. 4, pp. 699–705, April 2008.

[4] B. Shackleford, G. Snider, R. J. Carter, E. Okushi, M. Yasuda, K. Seo, and H. Yasuura, "A high-performance, pipelined, FPGA-based genetic algorithm machine," Genetic Programming and Evolvable machines, vol. 2, no. 1, pp. 33–60, 2001.

[5] P. R. Fernando, S. Katkoori, D. Keymeulen, R. Zebulum, and A. Stoica, "Customizable FPGA IP core implementation of a general-purpose genetic algorithm engine," IEEE Transactions on Evolutionary Computation, vol. 14, no. 1, pp. 133–149, Febraury 2010.

[6] N. Kavvadias, V. Giannakopoulou, and S. Nikolaidis, "Development of a Customized Processor Architecture for Accelerating Genetic Algorithms," Microprocessors and Microsystems, Vol. 31, Issue 5, pp. 347-359, August 2007.

[7] O. Kitaura, H. Asada, and M. Matsuzaki, "A custom computing machine for genetic algorithm without pipeline stalls," IEEE International Conference on Systems, Man, and Cybernetics, vol. 5, pp. 577-584, Tokyo, Japan, August 2002.

[8] H. G. Cobb, and J. J. Gerefenstette, "Genetic algorithms for tracking changing environments," Fifth International Genetic Algorithm Conference, 1993.

[9] F. Vavak, and T. C. Fogarty, "Comparison of steady state and generational genetic algorithms for use in nonstationary environments," IEEE International Conference on Evolutionary Computation, pp. 192-195, Nagoya, August 2002.