This is to certify that the

thesis entitled

A COMPARISON OF COHORT GA WITH CANONICAL SERIAL AND ISLAND-MODEL DISTRIBUTED GA'S

presented by

Huafeng Pei

has been accepted towards fulfillment
of the requirements for

Master's degree in Electrical Eng

Major professor

Date Dec. 13, 2000

A COMPARISON OF COHORT GA WITH CANONICAL SERIAL AND ISLAND-MODEL DISTRIBUTED GA'S

By

Huafeng Pei

A THESIS

Submitted to
Michigan State University
in partial fulfillment of the requirements
for the degree of

MASTER OF SCIENCE

Department of Electrical and Computer Engineering

2000

ABSTRACT

A COMPARISON OF COHORT GA WITH CANONICAL SERIAL AND ISLAND-MODEL DISTRIBUTED GA'S

By

Huafeng Pei

This thesis considers the cohort genetic algorithm, a new type of genetic algorithm introduced by Holland. The cohort GA differs in several ways from the traditional canonical serial GA and island-model distributed GA, and was developed as a means of reducing "hitchhiking" – premature convergence of currently low-significance loci located near loci at which good building blocks are found early in the search process. The objective of this work is a comparison of one version of the cohort GA with canonical serial and island-model distributed GA's on the basis of their abilities to reduce hitchhiking. The comparison is done on two test functions: royal road function and hyperplane-defined function. It is experimentally shown that even though theoretically the cohort GA can reduce hitchhiking, the particular version of the cohort GA tested is prone to another form of premature convergence and performs worse than the other GA's. It is also shown that a small change in the placement of offspring among cohorts in the cohort GA may dramatically change its performance. This suggests that further work on the cohort GA may be fruitful.

ACKNOWLEDGEMENTS

I would like to express my sincere gratitude to my advisor Dr. Erik Goodman, for his guidance, support and patience during all of my graduate studies. Thanks to my committee members, Dr. Bill Punch and Dr. Robert Schlueter for their support and valuable suggestions.

Thanks also go to Prof. John Holland and his student Theodore C. Belding at University of Michigan for their valuable information and suggestions.

I am much obliged to Prof. Min Pei and Dr. Subbiah Kannappan for their help and advice during my stay at MSU.

I would like to thank my writing group members, Dr. Renate Snider, Dr. Eric Torng, Julide Celik, Andrew Chen, Yong Chen and Georgies Theocharous for their careful proof-readings of my thesis.

I am also grateful to the many good friends whose help I could always count on.

Finally, and most importantly, I would like to thank my parents and the rest of my family for their love and understanding.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

INTRODUCTION

Genetic algorithm (GA) is a family of optimization methods introduced by Holland in 1975 [1]. It has been successfully applied to many different types of problems since then. Much research has been done in order to understand how the GA works and how to improve its performance.

The cohort GA is a new type of GA designed by Holland [2]. It is aimed at reducing the "hitchhiking" effect that occurs in the process of a GA searching for an optimal solution. Hitchhiking is a form of premature convergence that will hinder the GA or even make it unlikely for the GA to find a good solution for a given problem. Hitchhiking occurs when the maximum reproduction rate is high. That is, the expected offspring of the best individual in the population is set to be two or more. If the maximal reproduction rate reduced to 1.1 or 1.2, the second offspring of the best individual would be produced with a probability determined by the fractional part of its fitness. Then the best individual's gain become uncertain. Cohort GA is designed to reduce reproduction rates without using probabilities. It is believed that cohort GA's mechanism will reduce hitchhiking, thus improve the performance of GA. In this work, I will use empirical study to verify that whether a cohort GA can reduce hitchhiking effect and therefore improve the performance of GA's by comparing it with canonical serial GA's and island-model distributed GA.

Chapter 1 will introduce what the hitchhiking effect is, how it is discovered and the distinctions between cohort GA, canonical serial GA and island-model distributed GA. Chapter 2 will present a detailed experimental design. Comparison results are given in Chapter 3, and conclusions are given in Chapter 4.

CHAPTER 1

BACKGROUND

GA is inspired by the evolutionary theory. In most implementations, GA encodes the potential solution of a target problem as a binary string, which is analogous to a chromosome. An initial population of binary strings (individuals) is generated randomly. Each individual is evaluated by a fitness function or an evaluation function. Fitness value indicates how well this potential solution solves the problem and determines the number of offspring this particular individual can produce. Genetic operators such as crossover and mutation are applied to the population in order to explore the solution space. Crossover is the exchange of parts of two strings and mutation is flipping individual bits. The population will evolve through this evolutionary process. The GA is considered to be successful if the evolution results in the optimal solution to the problem.

1.1 Building block hypothesis and Royal Road functions

Holland's building block hypothesis states that a GA's searching power for the optimal solution comes from its ability to form solutions by repeatedly combining short, highly-fit schemata (or building blocks) to longer schemata, then to a complete solution [3]. The term "building blocks" refers to specific bit patterns

that contribute to high fitness. In order to test this hypothesis, a set of test functions -- Royal Road functions (RR) -- were designed [4, 5,6,7].

Royal Road functions were constructed from a small set of pre-defined building blocks with known fitness. Higher level building blocks consist of some lower level building blocks. The fitness of an individual is determined by which building blocks it is an instance of. Figure 1 and Figure 2 showed R1 and R2 two RR functions from [5]. The difference between R1 and R2 is that R1 doesn't have intermediate-order schemata.

s^1 = 11111111**********; c1 = 8

s^2 = ********11111111**********; c2 = 8

s^3 = ****************11111111**********; c3 = 8

s^4 = ************************11111111**********; c4 = 8

s^5 = ********************************11111111**********; c5 = 8

s^6 = **11111111**********; c6 = 8

s^7 = **11111111********; c7 = 8

s^8 = **11111111; c8 = 8

s_{opt} =11111111111111111111111111111111111111111111111111111111111111111;

Figure 1: Royal Road Function R1. The fitness function R1(x) (where x is a bit string) is computed by summing the coefficients c_s corresponding to each of the given schemas (s_i) of which x is an instance. For example, R1(1111111100...0) = 8 and R1(1111111111111100...0) = 16.

s^1 = 11111111**********; c1 = 8

s^2 = ********11111111**********; c2 = 8

s^3 = ****************11111111**********; c3 = 8

s^4 = ************************11111111**********; c4 = 8

s^5 = ********************************11111111**********; c5 = 8

s^6 = **11111111**********; c6 = 8

s^7 = **11111111********; c7 = 8

s^8 = **11111111; c8 = 8

s^9 = 1111111111111111**********; c9 = 16

s^{10}= ****************1111111111111111**********; c10 = 16

s^{11} = ********************************1111111111111111**********; c11 = 16

s^{12} = **1111111111111111; c12 = 16

s^{13} = 11111111111111111111111111111111**********; c13 = 32

s^{14} = ********************************11111111111111111111111111111111; c14 = 32

s_{opt} =11111111111111111111111111111111111111111111111111111111111111111;

Figure 2: Royal Road Function R2. The fitness function R2(x) (where x is a bit string) is computed by summing the coefficients c_s corresponding to each of the given schemas (s_i) of which x is an instance. For example, R2(1111111100...0) = 8 and R2(1111111111111111100...0) = 32.

According to the building block hypothesis, the hierarchical structure of R2 function should enable a GA to find the optimum solution quicker than R1.

However, even though R2 was designed to lay out a "royal road" for the GA to follow to the global optimum, the GA actually did worse on R2 than R1 in which the intermediate-order schema is removed. This is due to the phenomenon of hitchhiking.

1.2 Hitchhiking

Hitchhiking is an effect that when a building block is discovered, the alleles at loci that are near the building block spread through the population almost as rapidly as the building block itself [2,4,5]. Because these alleles usually have little to do with good schemata, hitchhiking results in reducing exploration in the parts proximal to the building block. In the RR case, once an instance of a higher-order schema is discovered, its high fitness allows the schema to spread quickly in the population, with zeros in other positions in the string hitchhiking along with the 1s in the schema's defined positions. It thus slows finding of the ones in the other positions, or, for practical run length, prevents it entirely. For example, in R2, fitness values go up exponentially with the level of the schema, and strings that are instances of the order-16 schema receive much higher fitness than strings that are only instances of the order-8 schema (32 vs. 8). So the instances of 1111111111111111*...* will soon take over the entire population once they appear, often with many zeros in the right half of the string. Those zeros are hitchhikers. This hitchhiking effect canceled the progress that the population has ever made for generating the schema **********1111111111111111*...*.

The GA must start over to discover the second schema, thus seriously hurting the GA's performance.

Some attempts have been made to reduce the hitchhiking effect [6,8,9]. The Royal Road functions R3 and R4 [6] were developed to reduce the severity of hitchhiking that occurred in R2 by reducing the large fitness jump that occurred when an instance of a new level of RR was found. Figure 3 shows R4, a modified version of R2 with fitness jump equal to 0.2 when an instance of a new level of RR is found. R3 also contains non-coding regions (also called introns) in which the lowest-level order-8 schemata are each separated by bit positions that do not contribute to fitness. The most likely positions for hitchhikers are those close to the highly fit schema's defined positions, since they are less likely to be separated from the schema's defined positions by crossover [6,8]. These attempts did reduced hitchhiking to some degree.

Level 1: $S_1\ S_2\ \ S_3\ S_4\ \ S_5\ S_6\ \ S_7\ S_8\ \ S_9\ S_{10}\ \ S_{11}\ S_{12}\ \ S_{13}\ S_{14}\ \ S_{15}\ S_{16}$

Level 2: $(S_1\ S_2)\ (S_3\ S_4)\ (S_5\ S_6)\ (S_7\ S_8)\ (S_9\ S_{10})\ (S_{11}\ S_{12})\ (S_{13}\ S_{14})\ (S_{15}\ S_{16})$

Level 3: $(S_1\ S_2\ \ S_3\ S_4)\ \ (S_5\ S_6\ \ S_7\ S_8)\ \ (S_9\ S_{10}\ \ S_{11}\ S_{12})\ \ (S_{13}\ S_{14}\ \ S_{15}\ S_{16})$

Level 4: $(S_1\ S_2\ \ S_3\ S_4\ \ S_5\ S_6\ \ S_7\ S_8)\ \ (S_9\ S_{10}\ \ S_{11}\ S_{12}\ \ S_{13}\ S_{14}\ \ S_{15}\ S_{16})$

Figure 3: Royal Road Function R4. Each s_i is an order 8 schema. The desired schemas are $s_1$to s_8 and combinations of them. A String that is not an instance of any desired schema receives fitness 1.0. Every time a new level is reached, a small increment u = 0.2 is added to the fitness. For example, strings at level 1 (that are instances of at least one level-1 schema) have fitness 1+0.2.

In Holland's point of view [2], hitchhiking occurs because of the high maximum reproduction rate. In order to enable the building blocks that are presented in the best individuals to spread out in the population quickly, the best individual in the population is set to produce 2 or more offspring in each generation typically. This high maximum reproduction rate setting is the source of the hitchhiking.

Increasing the mutation rate while keeping the maximum reproduction rate high will reduce hitchhiking. But the high mutation rate will result in the population not being able to retain information about building blocks that have already been discovered.

If the maximal reproduction rate is set to a relative low value, such as 1.1 or 1.2, the mutation rate could be set low enough to retain building blocks that have already been discovered. But it will become uncertain that how many generations it will take for the second offspring of the best individual to occur because the second offspring is generated with probability much less than 1 in each generation. Some schemata may be found only after a long search process and this setting becomes the source of extreme variance. In order to control this variance, and reduce hitchhiking, Holland proposed the cohort genetic algorithm.

1.3 Cohort GA

The cohort GA is intended to reduce hitchhiking by lowering the reproduction rate without using probabilities [2]. In the cohort GA, the population is divided into an ordered set of sub-populations. These sub-populations are

called cohorts. The initial population is generated randomly. Individuals in cohort 1 will produce offspring first. Then the individuals in the successive cohorts will have chances to produce. When the last cohort is reached, the process begins again with cohort 1. In this given order, reproduction is carried out by cycling through the cohorts. When it is time for an individual in a cohort to reproduce, the individual crossovers with another individual in the same cohort, then the offspring undergo mutation probabilistically. The fitness of an offspring determines the cohort into which the offspring will be put, which in turn determines when its turn is to produce offspring. In this way, a string with high fitness will produce offspring quicker than a string with low fitness. Over an extended interval, the string with higher fitness will produce more offspring than the string with lower fitness. Figure 4 describes the reproduction process in the cohort GA, supposing the population is divided into N cohorts and each cohort has M individuals initially.

All strings in the population can produce a fixed number of offspring when it is their turn to reproduce, no matter what their fitness values are. This method lowers the reproduction rate. Theoretically, this method will reduce the hitchhiking effect. In order to verify this idea, I will describe, in the next two sections, how the cohort GA is distinct from the canonical serial GA and island-model distributed GA in term of their mechanisms. Then I will report an comparison study on these three GA's in chapter 2 and 3.

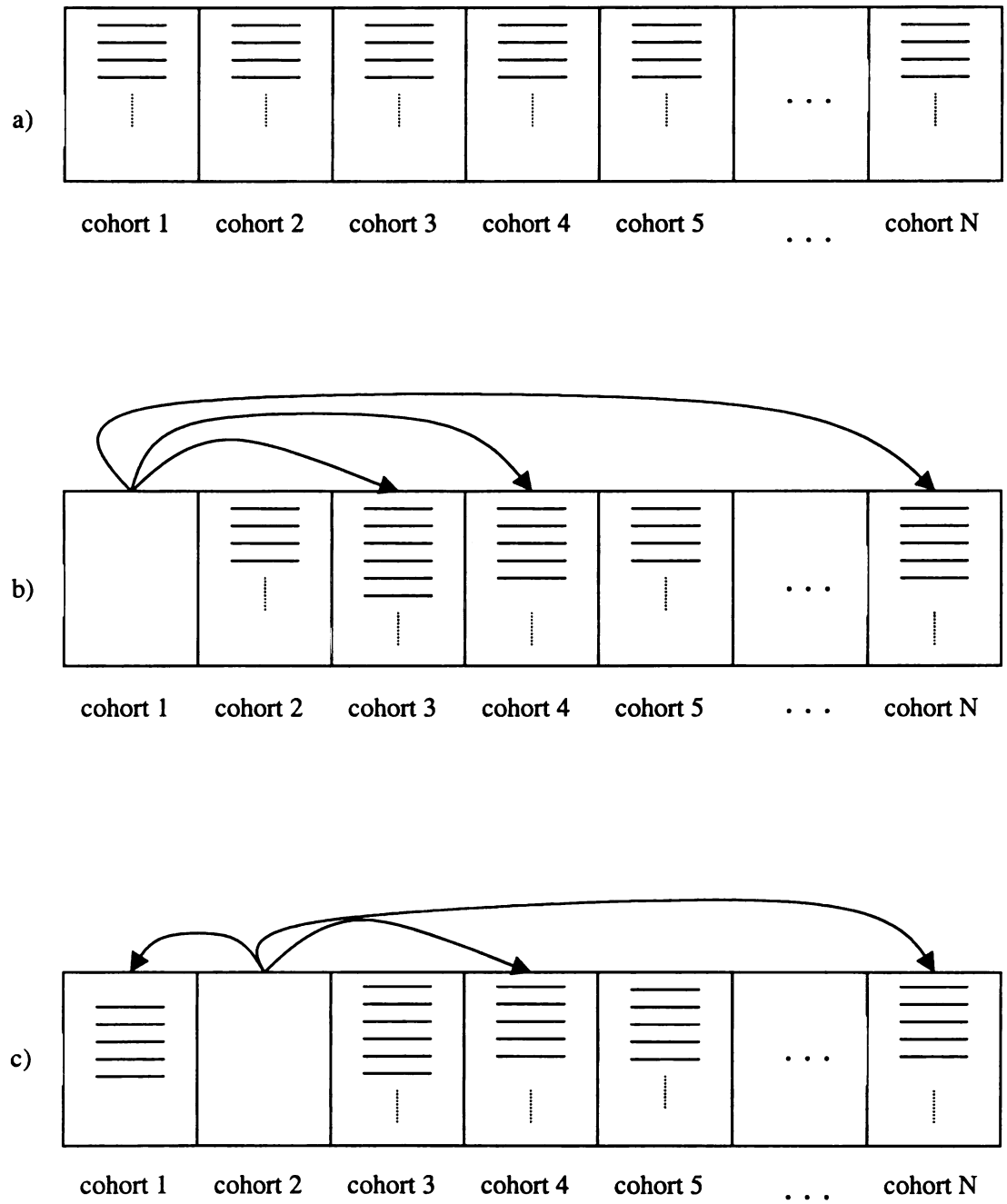

Figure 4: Reproduction process of cohort GA. a) Initial Cohorts. b) Offspring of the individuals in cohort 1 will go to cohort j, with 2<j<N, where j is larger as the fitness is smaller. c) The process is repeated for each cohort in turn, until it meets the termination condition.

1.4 C

tradit

popu

that

seria

1.4 Canonical Serial GA and its distinction from cohort GA

The canonical serial genetic algorithm is also called simple GA or traditional GA. It follows the basic structure of GA. Canonical GA works on a population of binary strings. The most charming feature of canonical serial GA is that it is simple and powerful. Figure 5 describes a typical process of canonical serial GA.

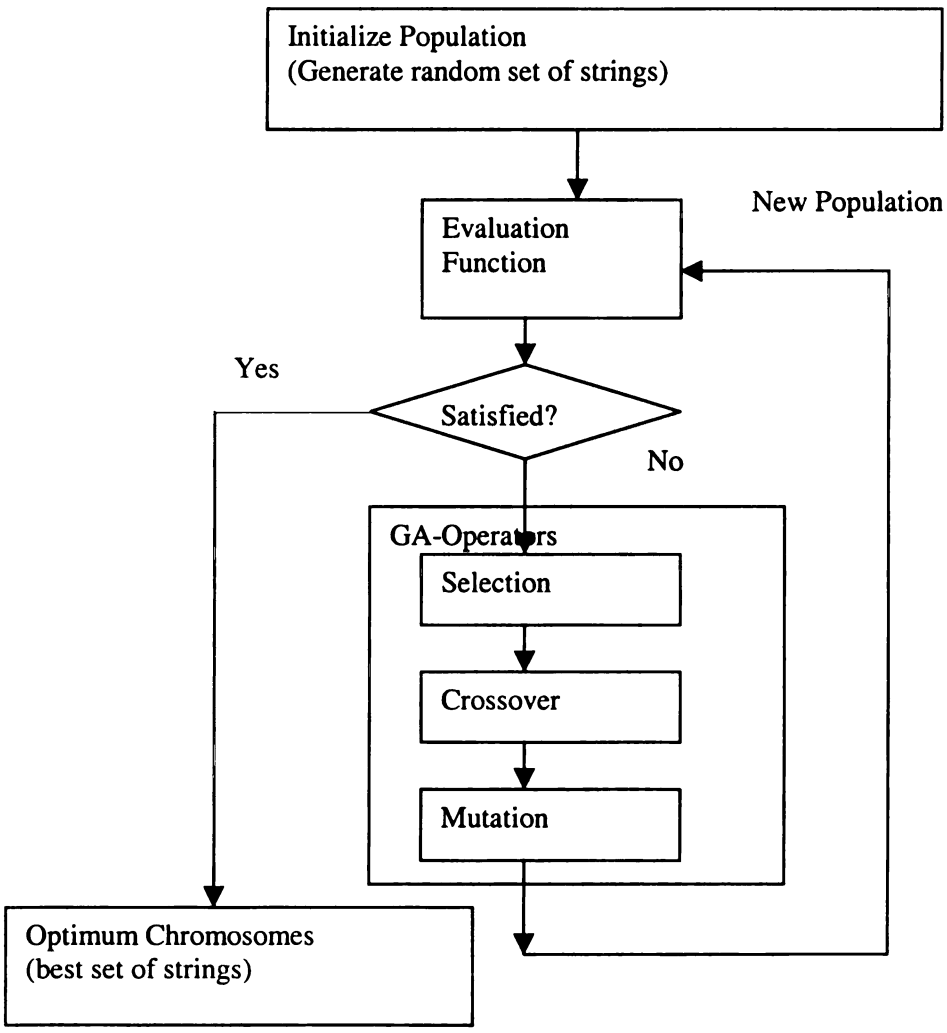

Figure 5: Canonical GA working process

There are two basic differences between the canonical serial GA and the cohort GA. First, a canonical serial GA is a generational GA and the cohort GA is a steady state GA. A generational GA selects parents from the old population to produce offsprings and the offsprings form a new population. This new population becomes the old population when the whole new population is created. Being a steady state GA [10,11,12], a cohort GA, on the other hand, selects parents from the population, and their offspring are placed back in the population (in other cohorts). The parents will be deleted to keep the size of the population constant. The individuals other than parents may be deleted, too, if the each parent individual produces more than one offspring. The second difference is that a canonical serial GA operates on a single large population while a cohort GA splits the big population into small cohorts and sequentially works on each cohort.

1.5 Island-model distributed GA and its distinction from Cohort GA

One common problem in the canonical serial GA approach is premature convergence. Selection may drive all the bits in some positions to a singe value after several generations. If this happens without the population converging to a satisfactory solution, then the population has prematurely converged [10]. The island-model distributed GA (coarse-grain parallelism GA) [13] is intended to assist the user in avoiding premature convergence on difficult multimodal problems. Rather than a single large population, an island-model distributed GA maintains multiple, separate subpopulations that evolve independently. Each of

these subpopulations could run as a normal genetic algorithm. These GA's could be different types of GA or the same type of GA with the same or different parameter settings. Individuals in each subpopulation can be exchanged between sub-populations according to a migration strategy. Figure 6 shows a migration strategy between six subpopulations.

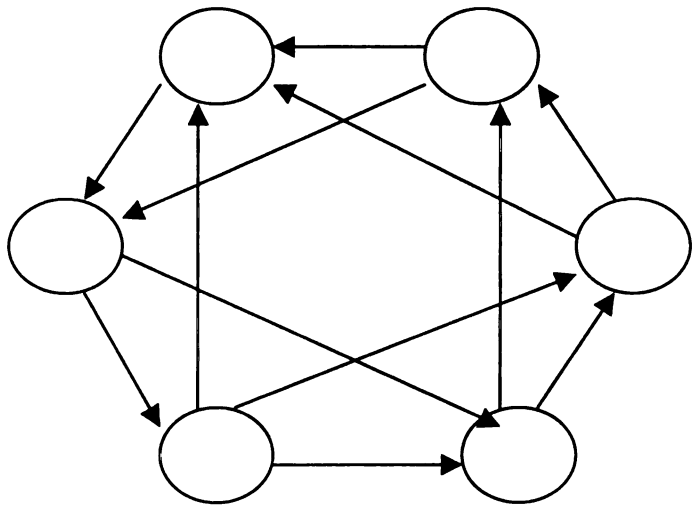

Figure 6: A migration strategy of an island-modal GA. Beside the direction of the migration, a migration strategy may also include other related parameters, such as the number of migrants.

Even though the cohort GA and island-model distributed GA both work on subpopulations, they are different in several ways. First, their placement of the offspring is different. In the cohort GA, the offspring of one cohort will be placed among all other cohorts except the current cohort; this will ensure that

information discovered in one cohort spreads through the whole population. In an island-model distributed GA, each subpopulation evolves itself just like a canonical serial GA. Offspring form a new generation within each subpopulation. Information exchange occurs in a migration process that happens only occasionally. Second, a cohort GA is steady state, while, an island-model distributed GA is generational; however, depending on parameter settings, an island model distributed GA can be made to have any behavior changing from pure generational ("generation gap" = 1.0) to steady state ("generation gap" = 0).

CHAPTER 2

EXPERIMENTAL DESIGN

2.1 Experiment objective

The main purpose of this work is to test whether a cohort GA can reduce the hitchhiking effect or not. By comparing a cohort GA with a serial canonical GA and island model distributed GA, I also tested whether a cohort GA is more efficient than other GA's. Another objective of this work is to test how some factors affect a cohort GA's performance.

2.2 Testbed – Royal Road function and HDF function

2.2.1 Royal Road function description

The Royal Road (RR) function used in this work is Holland's revised version of the RR function.[1] The RR function takes a binary string (chromosome) as input and produces a real number fitness value. A chromosome is composed of 2^k nonoverlapping contiguous regions, each of length b+g. My experiments take Holland's default settings, in which k is 4, b is 8, and g is 7. Therefore there

[1] Holland presented this version of RR at the Fifth International Conference on Genetic Algorithms in 1993 and then posted it to the Internet Genetic Algorithms mailing list. I take Jones T's " A Description of Holland's Royal Road Function" [14] as reference and use some different notations.

are 16 regions of length 15, giving an overall string of length 240. The part of length b is called a block (and is a basic building block) and the part of length g is called the intron part (which will not contribute to fitness). The calculation of the fitness of a chromosome is separated into two steps: PART calculation and BONUS calculation.

In PART calculation, each block is considered individually and in the same way. Each block receives a fitness value, and all these fitness values in a chromosome are added to produce the total PART. The fitness of each block is based on how many one bits it contains. If the number of ones in a block is less than or equal to a limit M, every one adds to the block's fitness by v. Thus with the setting v equal to 0.02 and M equal to 4, a block with three one's would have fitness $3 \times 0.02 = 0.06$. If a block contains more than M ones, but less than b ones, it receives –v for each one over the limit. Thus a block with six ones is assigned a fitness of $(6\text{-}4) \times \text{-}0.02 = \text{-}0.04$. If a block consists entirely of ones, it receives 0 from the PART calculation. However, it will receive BONUS, and it is said to be complete.

The BONUS calculation rewards completed blocks and some combinations of completed blocks with the concept of "levels". At the lowest level, 0, rewards are given for the complete blocks (i.e., blocks that consist entirely of ones). The first completed block receives u* as its fitness. Any additional complete blocks receive a fitness of u. Thus with the setting u* equal to 1, u equal to 0.3, a chromosome that contained three complete blocks would

receive

1. pairs

B_0. B_1.

$0 \le i < 2^k$

additio

$0 \le l <$

with 0

receive

total fit

BONU

reward

(level

2)

levels,

total B

receive a BONUS fitness of 1.0+(2×0.3) = 1.6 for level 0. At the next level, level 1, pairs of blocks are rewarded. Suppose we label the blocks from left to right as B_0, B_1,..., B_{2^k-1}. The function then rewards any completed pair (B_{2i},B_{2i+1}) for $0 \le i < 2^{k-1}$. As at level 0, the first completed pair of blocks receives u*, and additional completed pairs receive u. In general, there are k+1 levels, and at level $0 \le l < k$, contiguous sets of 2^l complete blocks, whose first block is labeled $B_{2i}{}^l$ with $0 \le i < 2^{k-1}$, are rewarded. At all levels, the first such set of complete blocks receives fitness u*, and additional sets of completed blocks receive fitness u. The total fitness for the level is the sum of these fitnesses. Thus with k equal to 4, the BONUS fitness calculation rewards completed single blocks (level 0) and rewards the completion of the sets of completed blocks:

$\{B_0, B_1\}, \{B_2, B_3\}, \{B_4, B_5\}, \{B_6, B_7\}, \{B_8, B_9\}, \{B_{10}, B_{11}\}, \{B_{12}, B_{13}\}, \{B_{14}, B_{15}\}$ (level 1)

$\{B_0, B_1, B_2, B_3\}, \{B_4, B_5, B_6, B_7\}, \{B_8, B_9, B_{10}, B_{11}\}, \{B_{12}, B_{13}, B_{14}, B_{15}\}$ (level 2)

$\{B_0, B_1, B_2, B_3, B_4, B_5, B_6, B_7\}, \{B_8, B_9, B_{10}, B_{11}, B_{12}, B_{13}, B_{14}, B_{15}\}$ (level 3)

$\{B_0, B_1, B_2, B_3, B_4, B_5, B_6, B_7, B_8, B_9, B_{10}, B_{11}, B_{12}, B_{13}, B_{14}, B_{15}\}$ (level 4)

The total BONUS is computed by adding the fitness at each of the k+1 levels, and the fitness of a chromosome is the sum of the total PART and the total BONUS.

2.2.2 The motivation of using the RR function

The hitchhiking phenomenon was noted by Holland while using the original RR functions to test the building block hypothesis. Thus it is natural to use the RR function as a testbed to compare cohort GA's performance with other GA's. The characteristics of the RR function, such as known building blocks, known fitness and a hierarchical structure of building blocks, should enable us to track the GA's performance over time. The revised RR function is especially good for testing because it incorporates the idea of "deception" or a fitness valley that the GA must cross to find a global optimal solution.

2.2.3 HDFs description

Hyperplane-defined functions (HDFs) are a set of randomly generated functions. They are extensions to the Royal Road functions and are designed to allow a large number of building blocks to be combined in a variety of ways. They also incorporate the idea of "pot holes", which refers to the fitness valleys that must be crossed in order to reach the global optimum. The interactions among the building blocks are more dramatic in HDFs than in RR.

Similar to RR, HDFs are composed of sets of pre-defined schemata with known fitness. The fitness of an individual chromosome is simply the sum of the fitness of the schemata that it contains.

$$u(x) = \max\left\{0, \sum_{s \in S | x \in s} u(s)\right\}$$

where S is the set of component schemata, s is one of the component schemata and x is the chromosome to be measured.

To generate an HDF function, one must specify the following parameters:

chrl: the length of a chromosome

nelt: the number of elementary schemata

maxl: maximum length of elementary schemata

minl: minimum length of elementary schemata

npr: the number of additional positions in refinements of elementary schemata

nocom: number of combinants to be produced.

Here I give a description on how to generate a HDF in general:

1). n = 0.

2). Randomly generate an elementary component schema

a) *Choose the new schema's order o, which is a random number between minl and maxl.*

b) *Choose o loci, each locus is a random number between 0 and chrl.*

c) Generate a new schema s by assigning a random allele from the set {0, 1} to each of the loci selected in b), and assign don't-care values (*) to the remaining loci.

d) Assign s a fitness u(s). Put the new schema in S and Se, where Se is the set of elementary schemata. Let n = n+1. if n == nelt, go to 3), If not, go to 2).

3). Add a refinement of a randomly selected elementary schema to S, produced by adding "nearby", randomly chosen defining bits to the schema.

a) Generate a set of eligible refinement positions that are distinct from any positions in the schemata of Se.

b) Form the combinations of positions generated by 3a.

c) Form a new schema s' from s in Se by assigning a random allele from the set {0, 1} to each of the undefined loci chosen in step 3b.

d) Assign the new schema a fitness u(s). Put the new schema in S and Sr, where Sr is the set of refined schemata.

4). Form new compound schemata by combining elementary and refined schemata.

a) *Generate designations of all combinants of number nocom. The designations refer to the sets of indices for elementary and refined schemata in S.*

b) *Generate defining positions and values for each combinant (new schema). The new schema is the intersection of schemata in its designed set that was generated in 4a). If the schemata are incompatible at one or more loci, resolve the conflicts by setting each locus to a random allele from the set {0, 1}. Assign each new schema a fitness u(s). Put the new schema in S.*

c) *Stop.*

In the experiments of this work, the fitness value of each schema is assigned as following: all the schemata in Se are assigned fitness 2, all the refined schemata are assigned fitness –1, and all the combinant schemata are assigned fitness 3. In this way, the refined schemata serve as "pot holes".

Figure 7 shows the HDF that is used in this work.

```
Loci:
{4  5  6  7  8}  {9  10  11  12  13  14}  {15  16  19  20  21  22}
{22  24  29}  {28  29  31}  {28  30  31  33}  {39  40  41  43  44}
 {55  56  57  58  59  61  62}  {4  5  6  7  8  15  16  19  20  21  22}
 {15  16  19  20  21  22  24  29}  {15  16  19  20  21  22  28  29  31}
 {22  24  29  28  31}  {28  29  31  39  40  41  43  44}  {4  5  6  7  8
11}  {9  10  11  12  13  14  19}  {15  16  19  20  21  22  29}  {22  24
29  31}  {28  29  31  33}  {28  30  31  33  40}  {39  40  41  43  44
59}  {8  9  10  11  12  13  14}  {9  15  16  19  20  21  22}  {21  22
24  29}  {22  28  29  31}  {29  28  30  31  33}  {31  39  40  41  43
44}  {41  55  56  57  58  59  61  62}

Allele:
{1  1  0  0  0}  {1  1  1  1  0  0}  {0  0  0  0  0  0}  {1  0  0}  {1
0  1}  {1  0  1  1}  {0  1  0  1  0}  {1  0  0  0  0  0  0}
 {1  1  0  0  0  0  0  0  0  0  0}  {0  0  0  0  0  0  0  0}
 {0  0  0  0  0  0  1  0  1}  {1  0  0  1  1}  {1  0  1  0  1  0  1  0}
 {1  1  0  0  0  1}  {1  1  1  1  0  0  0}  {0  0  0  0  0  0  0}
 {1  0  0  1}  {1  0  1  1}  {1  0  1  1  1}  {0  1  0  1  0  0}
 {0  1  1  1  1  0  0}  {1  0  0  0  0  0  0}  {0  1  0  0}  {1  1  0
1}  {0  1  0  1  1}  {1  0  1  0  1  0}  {0  1  0  0  0  0  0  0}

Values:
{2  2  2  2  2  2  2  2  3  3  3  3  3  -1  -1  -1  -1  -1  -1  -1  -1
-1  -1  -1  -1  -1  -1}
```

Figure 7: HDF generated with the following setting: chrl = 80, nelt = 8, minl = 6, maxl = 12, nrp = 2, and nocom = 5. Each parenthesized expression represents a schema. For example, the first schema's defining positions are 4, 5, 6, 7, 8, the defining values of each position are 1, 1, 0, 0, 0, respectively, and the fitness of this schema is 2.

2.2.4 The motivation of using the HDFs

The HDFs are easy to generate, hard to optimize, and easy to analyze. They also overcome some of the RR's drawbacks:

- They don't have a simple hill.
- There are many paths to any improvement.
- The interactions among the building blocks are more dramatic. A large number of building blocks can be combined in a variety of ways. Most combinations offer no improvements. Some combinations will produce

such improvement that will not produce further improvement. Some combinations will decrease the performance in the sense that the combined building block performs more poorly than the elementary building blocks that constitute it. Some combinations will produce further improvements and lead to higher order building blocks.

- They have more levels than RR and allow multiple improvements without the saturation effect. Saturation effect refers to that RR may not reach a new level even after a long period of search, it occurs when a large part of the chromosome must be fixed to get further improvements.

- Complexity is easy to control.

The experiments done by Holland [2] suggest that the cohort GA worked very efficiently on HDFs. It was shown that previously discovered schemata could be combined to yield longer schemata. At every stage, discovered schemata were retained, and the number of instances of the schemata increased as the cohort GA keeps running. To verify Holland's result and compare the efficiency of the cohort GA with other GA's, I tested the same HDF on three GA's in an attempt to see the cohort GA's ability to cross the "pot holes" and achieve higher fitness without severe hitchhiking.

2.3 Performance Criteria

Based on the different nature of the two test functions, I chose different criteria to measure the GA's performance. The RR function has explicit levels. Therefore, for the RR, I measured the number of function evaluations required to achieve a certain RR level. In order to measure the degree of the convergence in the intron part of the chromosome, I generated a relatively short chromosome in HDF (equal to 80). For each intron locus, I measured the difference between the total number of ones and the total number of zeros of all chromosomes in the population and used the absolute value of their difference as a criterion. I also measured the maximum fitness value that the various GA's achieved given a fixed number of function evaluations.

2.4 Cohort GA implementation detail

There are many ways to implement the cohort GA's central idea. I used Holland's version in [2] and made some modifications during the series of my tests. Figure 8 and 9 shows how Holland's cohort GA works, in some detail.

The following parameters need to be specified before running the cohort GA.

chrl: length of the chromosome

nocoh: the number of cohorts

lencoh: initial size of each cohort

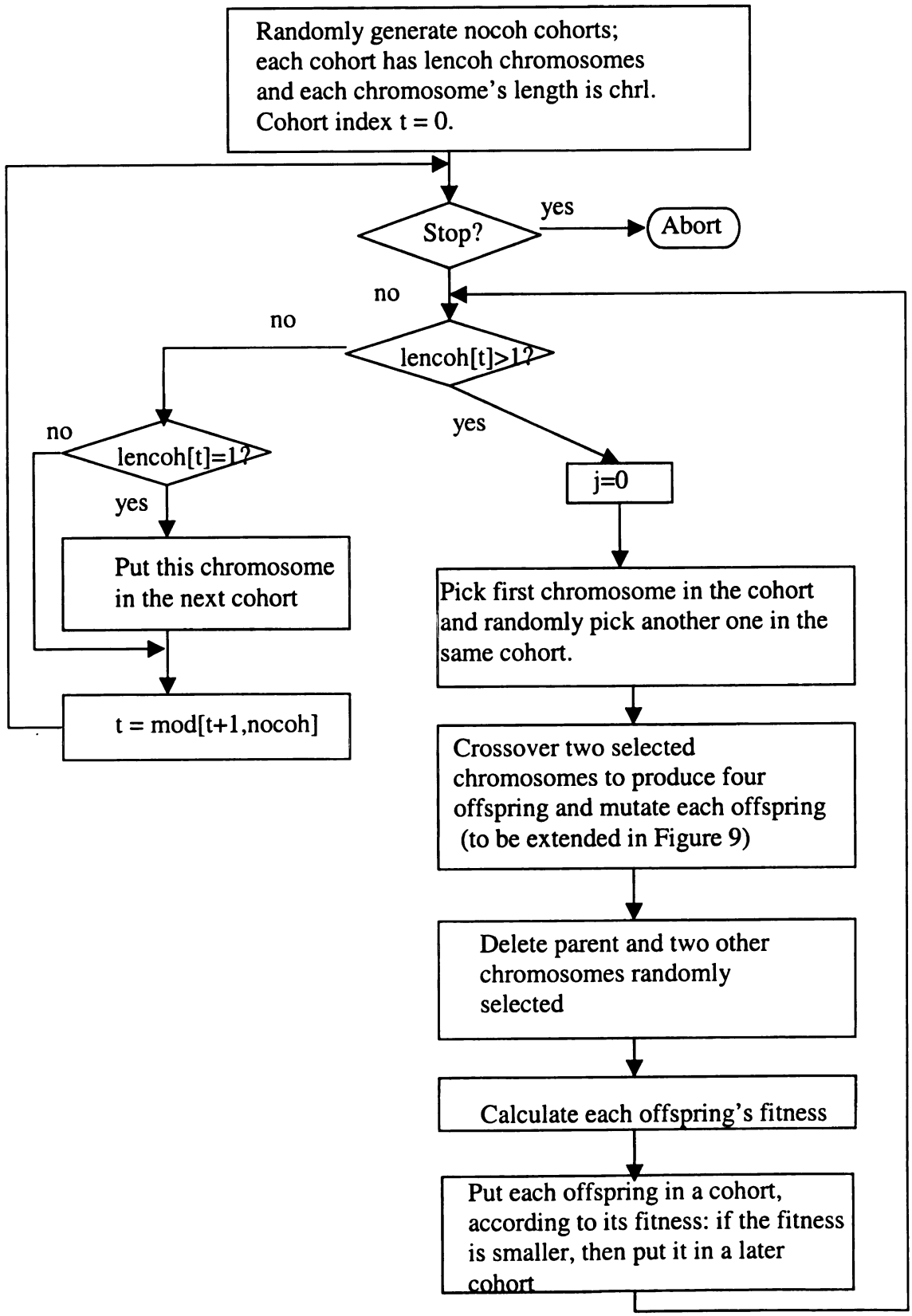

Figure 8: Cohort GA's implementation in detail.

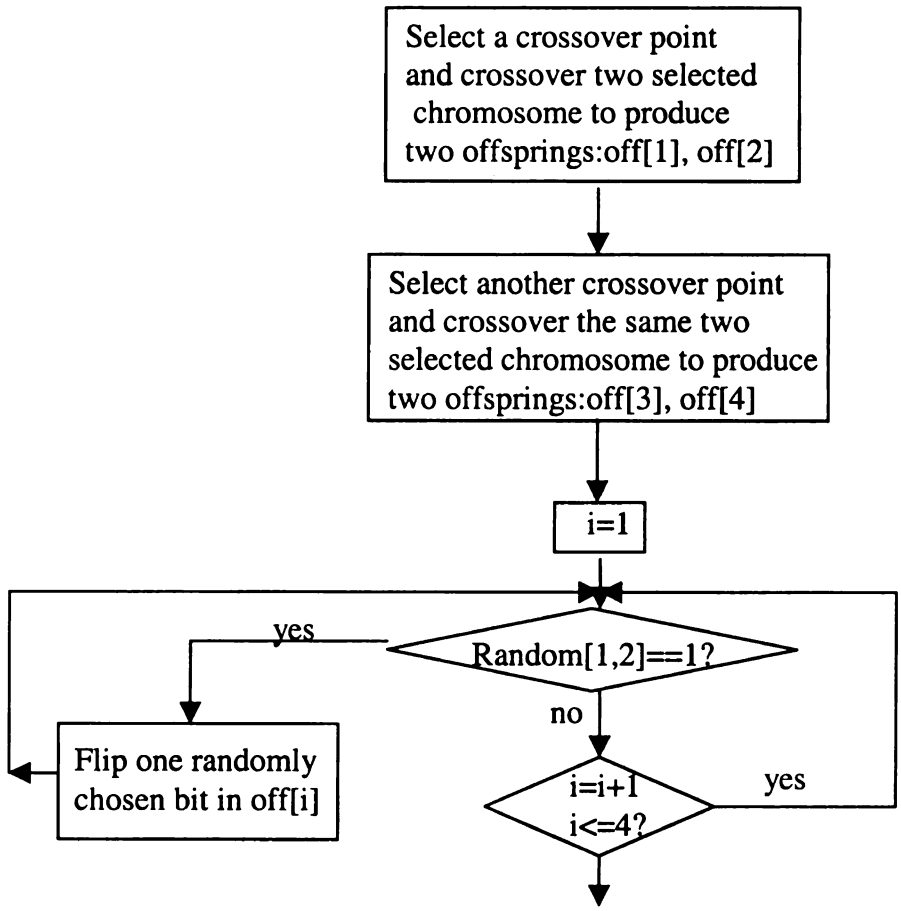

Figure 9: Crossover and mutation part of the cohort GA. Extended from Figure 8.

2.4 Modifications made to the original cohort GA

Several factors that may affect a cohort GA's performance, such as population size, offspring placement strategy, deletion strategy and crossover candidates chosen. These factors are considered during the tests, corresponding

changes are made to the original implementation, and repeated tests are done with different settings and implementations.

Population size

It also includes the relationship between nocoh and lencoh. Holland's default setting is nocoh = 20 and lencoh = 20; thus population size is 400. I chose different values for nocoh and lencoh and different combinations of these values to perform the tests in an attempt to see how these two parameters affect the cohort GA's performance.

Offspring placement strategy

Which specifies cohort an offspring with a certain fitness value will be placed in. It will affect selection pressure as well as interactions between cohorts. In Holland's original implementation, an offspring with fitness u is placed in cohort d, where d is determined by the equation:

$d = \mathrm{mod}(t + doub, nocoh)$, where t is the current cohort number and

$doub = \lceil 2 \times umax / u \rceil$, where umax is maximum fitness value found so far.

In this way, an individual with fitness umax is placed in the cohort next to the current cohort. And other individuals will be placed in a cohort based on the ratio of umax and its fitness value. An individual with higher fitness value will be put nearer the current cohort, and vice-verse.

During the experiments in RR, I found out that with this implementation, the individuals tend to accumulate in a small set of cohorts. In order to spread the

individuals among all the cohorts and keep the cohort GA working as intended, I tried a new placement strategy with

doub = $\lceil$(nocoh – 1) + (u – umin) × (2 – nocoh +1) / (umax – umin) $\rceil$ and umin is the minimum fitness found so far. In this way, a chromosome with the fitness umin will be placed in the cohort nocoh –1 from the current cohort (which is the farthest cohort from the current cohort) and a chromosome with the fitness umax will be placed in the cohort next to the current cohort. Other individuals will be placed in a cohort between 2 and nocoh – 1 from current cohort with the doub value proportional to the individual's fitness value.

The above strategy is deterministic. I also tried to put the offspring in a randomly selected cohort with some probability in order to produce more migration effect by providing the opportunity for inter-cohort mating.

Deletion strategy

Each paired individual will produce four offspring. To keep the population size constant, the parents are deleted as well as two other chromosomes which are randomly selected from two other cohorts. The cohorts that are the source of chromosomes for deletion are also randomly selected. The deletion of random chromosomes from random cohorts affects the cohort GA's ability to keep the good individuals found so far; it also affects selection pressure. In the original implementation, the source cohorts from which to delete chromosomes are randomly selected from cohort positions nocoh/2 to nocoh-1 relative to the current cohort; that is, the distant half of the cohorts. I also tried to delete

chromosomes randomly from 2 to nocoh –1 relative to the current cohort to lower the selection pressure.

Crossover candidate

In the original implementation, both parents are selected from the current cohort. I tested the one parent selected from the current cohort and with some probability another parent is randomly selected from another cohort. This strategy is also an attempt to provide inter-cohort mating.

2.5 Implementation details of canonical serial GA and island-model distributed GA

Though the procedure of a typical GA is well known, implementation details vary from system to system. And even very small differences in implementation may result in significant changes. Here I give the details of my implementation of a canonical serial GA and island-model distributed GA.

The canonical serial GA and island-model distributed GA software used was "GALOPPS" (The "Genetic Algorithm Optimized for Portability and Parallelism" System) [15]. The following are the descriptions of the steps of the GA that GALOPPS provided and which was used in my experiments.

2.5.1 Implementation of canonical serial GA

1) Create initial population (generation 0). The individuals of the initial population are generated randomly. Each bit in each individual is equally likely to be initialized to a 1 or a 0.

2) Evaluate the fitness of each individual in the current generation. Also calculate the maximum fitness, average fitness, standard deviation of the fitness and other statistics of the current population.

3) Terminate the program if the maximum generation number is reached. Otherwise, continue. The maximum generation number is specified in an input file.

4) Select the survivor or parents of the next generation, using "stochastic universal sampling" [16] selection method to select a list of chromosomes that will be the parents or the survivors of the next generation, sampling with the replacement; the size of the list is equal to the population size.

5) Reproduce. Here I take two choices. One is the straight canonical serial GA and another one allows niching of the population by using crowding and incest reduction. The latter technique is intended to help reduce premature convergence of the population. Employing this technique in my comparisons helped us to see where cohort GA stands among these techniques which reduce premature convergence.

In the straight canonical serial GA, both parents are uniform randomly selected from the list generated by step 4. One point crossover is performed on the parents and single bit mutation or multi-bit mutation is performed according to the crossover rate and mutation rate. The offspring then replace the parents.

The niching technique includes two mechanisms: one is crowding and another is incest reduction [15]. With incest reduction, pairs for crossover are picked by choosing the first parent at uniform random from the above list, then uniform randomly choose several possible candidates for the other parent; the number of candidates I used is 3. Among three candidates, the one with the greatest Hamming distance from the first parent is picked as the second parent. After the crossover (and any mutation) is done, for each child, "crowding-factor" (here, 3) members of the above list are selected (at uniform random). Among 3 candidates, the one with smallest Hamming distance from the child will be replaced.

The list generated by step 4 is not altered in this process. All individuals in this list are used in some crossover and/or mutation operation, or else survive unaltered into the next generation.

6) Go to step 2.

2.5.2 Implementation of island-model distributed GA

The island-model distributed GA divides the whole population into several subpopulations. It provides a chance for parallel execution by allowing use of several processors or computers. In my experiments, I use one workstation to serially simulate parallel execution. In that approach, I only use the distributed GA's ability of preventing premature convergence but not its ability of parallel execution.

In this case, each subpopulation is simulated for some number of generations called a cycle. Each population receives one turn per cycle. At the beginning of each population's turn, it reads one or more individuals from each of its declared neighboring subpopulations according to a master table. In addition to the neighboring subpopulations of each subpopulation, the master table also defines how many individuals are to migrate in from each neighbor each cycle, whether these migrants include the best individual and/or some number of randomly selected individuals, and which individuals are to be replaced by migrants. Immigration incest reduction and immigration crowding factor are used to direct the donating and receiving process. When a migrant is to be selected randomly and immigration incest reduction is used, the number "immigration incest reduction" of candidates are randomly chosen first. Then the one with the farthest Hamming distance from the best individual of the receiving subpopulation is selected. Immigration crowding factor refers to the random choice in the receiving subpopulation of migration_crowding_amount candidates for replacement, and picking the one that is closest in Hamming distance to the migrant to be replaced by the migrant.

The run of each subpopulation is same as for the canonical serial GA with crowding and incest reduction.

CHAPTER 3

RESULTS

The experiments described in this chapter investigate the argument for the cohort GA by comparing it with a canonical serial GA and island-model distributed GA and studying some of the implementation issues involved in cohort GA. All these experiments were performed on the RR function and the HDF. Each experiment was run 20 times and the results were the average of 20 runs.

3.1 Comparison of results using the RR function

3.1.1 Initial experiments

The first set of experiments investigates the performance of five different GA's on the RR function. The GA's include: original cohort GA, cohort GA with new placement implementation, the island-model distributed GA, the canonical serial GA and the canonical serial GA with niching.

The original cohort GA setting is as follows:

Number of cohorts: 20

Initial size of each cohort: 20

Population size: 400

Offspring placement strategy: doub = $\lceil 2 \times umax / u \rceil$ (note that d's calculations are always the same)

Deletion strategy: nocoh/2 to nocoh –1

Crossover: within the same cohort.

Stopping criterion: the total number of function evaluations exceeds 300,000

I also ran the cohort GA with the new offspring placement strategy:

$$doub = \lceil (nocoh - 1) + (u - umin) \times (2 - nocoh + 1) / (umax - umin) \rceil$$

The tests on canonical serial GA's and island-model distributed GA were done with the population size equal to 400 and following parameter settings:

Canonical serial GA:

Crossover rate: 0.15 (one point crossover)

Mutation rate: 0.0002 (per bit. Thus, 0.048 per chromosome)

Linear scaling with ratio of best fitness to mean fitness equal to 1.25

Stopping criterion: the generation that allows the total number of function evaluations to exceed 300,000

When niching was used:

Parameters are the same as canonical serial GA, plus:

Crowding factor: 3

Incest reduction: 3

Island-model distributed GA:

Number of subpopulations: 8

Population size of each subpopulation: 50

The settings of each subpopulation are the same as for the canonical GA with crowding and incest reduction. Migration neighbors are showed in Figure 10.

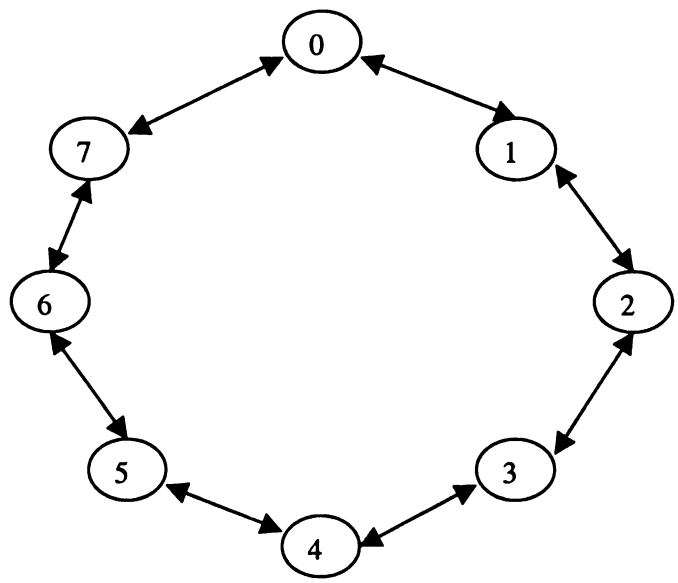

Figure 10: The neighboring subpopulation for each subpopulation, each subpopulation has two neighbors. For example, subpopulation 0 has subpopulation 1 and subpopulation 7 as its neighbors.

Number of cycles: 10

Number of neighbors of each subpopulation: 2

The number of migrant: 2 (one is the best, one is random)

Immigration incest reduction: 3

Immigration crowding factor: 4

The above parameter settings are to be taken as the defaults. In the following experiments I will only report the exceptions.

Table 1 lists the results on the RR function, of five GA's with population size equal to 400. Surprisingly, the cohort GA is the worst. In a total of 20 runs at this setting, the cohort GA achieves only level 1 and only in 5 runs. The results with the new placement implementation are much better. All the runs reached level 1 and seventy percent of the runs reached level 2. The other GA's achieved level 1 and level 2 with a smaller number of function evaluations, but also in a smaller percentage of the runs. In most runs, they did not reach level 2 within the number of function evaluations of the stopping criterion.

GA's used to work better on RR while the population size is larger. Because when the population size is larger, it is more possible for the initial population to contain all the basic building blocks GA's need to get to further level through crossover. Otherwise, it might take GA's a very long period of time to get

the basic building blocks through mutation. Therefore, the larger population sizes are used in the rest of the experiments on RR.

		level 1	level 2	level 3	level 4
original cohort GA	average	61326 (25%)			
	std. dev	60606			
cohort GA with new placement implementation	average	1786	28779 (7%)		
	std. dev	923	41898		
Island-model distributed GA	average	11344	59046 (65%)		
	std. dev	8975	24390		
Canonical Serial GA	average	1927	5552 (45%)		
	std. dev	853	1572		
Canonical Serial GA with niching	average	1885	5306 (35%)		
	std. dev	720	678		

Table 1: The average number of function evaluations until the optimum is found and the standard deviation of this average. The number in parentheses is the percentage of runs that the GA achieved that level. Average number shown without parentheses means hundred percent of runs achieved that level. Blank entries indicate that the level was never achieved in 300,000 function evaluations.

3.1.2 Varying the number of cohorts and initial size of each cohort

To test the effect of population size on the cohort GA, I ran the cohort GA's on different numbers of cohorts and different initial sizes of each cohort. The number of cohort equaled: 20, 35, and 50. The initial size of the each cohort

equaled: 20, 40 and 80. Thus, the population size varied from 400 to 4000. The results are listed in Table 2 to Table 5.

Table 2 shows that with the original cohort GA, the average number of function evaluations used to achieve level 1 is relatively small, with the number of cohorts equal to 20 and the cohort size equal to 40 and 80. With a cohort size of 80 and the number of cohorts at 35 or 50, most of the runs reached level 1, but also with many more function evaluations. This could be due to the population sizes being much bigger in these two cases. The initial population is more likely to contain instances of level 1 building blocks or schemata similar to them. In the latter case, after a long period of search, mutation led the RR to reach level 1. This confirmed that we need a bigger population size for RR. However, even though the number of cohorts, the cohort size and their ratio had some impact on the original cohort GA, it never achieved RR level 2 in all the runs I tried, due to the accumulation of individuals in only few cohorts.

With the new offspring placement implementation, most runs have achieved RR level 2, and a small number of runs achieved RR level 3, as shown in Table 4 and Table 5. This shows that the new offspring placement implementation improved the performance of the cohort GA. For the rest of the experiments on RR, I will only report the results of using the new offspring placement implementation. The cohort GA's performance didn't change linearly with changing of the number of cohorts, cohort size or population size. But it is shown that with the number of cohorts equal to 20, the cohort GA performs better overall, especially with cohort size equal to 80.

cohort size		# of cohorts = 20	# of cohorts = 35	# of cohorts = 50
20	average	61327 (25%)	38282 (40%)	60356 (45%)
	std. dev	60606	33704	49227
40	average	6093 (45%)	9124 (60%)	34577 (60%)
	std. dev	15062	19969	62927
80	average	16136 (50%)	139569 (80%)	151198
	std. dev	45328	157056	171798

Table 2: Results of the original cohort GA. The average number of function evaluations to achieve level 1 and the standard deviation of this average.

cohort size		# of cohorts = 20	# of cohorts = 35	# of cohorts = 50
20	average	1786	2854	3215
	std. dev	923	1063	1970
40	average	3352	3697	4392
	std. dev	1412	2295	2962
80	average	5151	4727	6541
	std. dev	2093	3288	4430

Table 3: Results of cohort GA with new placement implementation. The average number of function evaluations to achieve level 1 and the standard deviation of this average.

cohort size		# of cohorts = 20	# of cohorts = 35	# of cohorts= 50
20	average	28779 (70%)	37431 (80%)	44014 (55%)
	std. dev	41898	17337	27720
40	average	49627 (80%)	52819 (75%)	46973 (75%)
	std. dev	43897	26108	36142
80	average	35433 (95%)	59598 (80%)	64613 (85%)
	std. dev	18686	32518	41379

Table 4: Results of cohort GA with new placement implementation. The average number of function evaluations to achieve level 2 and the standard deviation of this average.

cohort size		# of cohorts = 20	# of cohorts = 35	# of cohorts = 50
20	average		65647 (15%)	73216 (10%)
	std. dev		14330	11677
40	average	148545 (10%)	165175 (15%)	
	std. dev	14332	61756	
80	average	107102 (30%)		316671 (5%)
	std. dev	7142		

Table 5: Results of cohort GA with new placement implementation. The average number of function evaluations to achieve level 3 and the standard deviation of this average.

Based on this observation, I proceeded with tests with the number of cohorts equal to 20 and 35 and the cohort size varying from 20 to 200, in an attempt to see whether the ratio of the cohort size and the number of cohorts really has some effect on the cohort GA's performance. The results are listed in Table 6 and Table 7.

While setting the number of cohorts at 20 still gives overall better performance, the performance doesn't improve linearly with an increase of the cohort size. The ratio of the cohort size and the number of cohorts doesn't seem to determine the cohort GA's performance.

cohort size		level 1	level 2	level 3	level 4
20	average	1786	28779 (70%)		
	std. dev	923	41898		
40	average	3352	49627 (80%)	148545 (10%)	
	std. dev	1412	43897	14332	
80	average	5151	35433 (95%)	107102 (30%)	
	std. dev	2093	18686	7142	
120	average	5586	43111 (85%)	165575 (40%)	
	std. dev	2981	20938	61669	
160	average	7800	56257 (90%)	172955 (35%)	
	std. dev	4351	26904	50264	
200	average	6577	49952 (80%)	193042 (20%)	420068 (5%)
	std. dev	4412	25705	71593	

Table 6: The average number of function evaluations until the optimum is found and the standard deviation of this average. The number of cohorts is 20. The stopping criterion for this test is the total number of function evaluations exceeding 500,000.

cohort size		level 1	level 2	level 3	level 4
20	average	2854	37431 (80%)	65647 (15%)	
	std. dev	1063	17337	14330	
40	average	3697	52819 (75%)	165175 (15%)	
	std. dev	2295	26108	61756	
80	average	4727	59598 (80%)		
	std. dev	3288	32518		
120	average	6587	65740 (75%)		
	std. dev	4030	37449		
160	average	6806	73607 (80%)		
	std. dev	4698	23745		
200	average	6806	73607 (80%)		
	std. dev	4698	23745		

Table 7: The average number of function evaluations until the optimum is found and the standard deviation of this average. The number of cohorts is 35. The stopping criterion for this test is the total number of function evaluations exceeding 500,000.

The island-model distributed GA and canonical serial GA's were tested on population sizes of 1600 and 4000. These population sizes are those that gives better performance on the cohort GA. Table 8 and Table 9 list the comparison results.

The results show that among four GA's, the island-model distributed GA and the two canonical serial GA's give significantly better results than the cohort GA. This may indicate some defects of the implementation of the cohort GA or defects that relate to the RR's shortcomings.

GA type		level 1	level 2	level 3	level 4
cohort GA with new placement implementation	average	5151	35433 (95%)	107102 (30%)	
	std. dev	2093	18686	7142	
Island-model distributed GA	average	2782	17860	36865 (90%)	
	std. dev	1529	4428	10538	
Canonical Serial GA	average	5658	18360	31394 (70%)	
	std. dev	2374	4258	5402	
Canonical Serial GA with niching	average	5248	21458	50321 (95%)	
	std. dev	2390	5041	16129	

Table 8: The average number of function evaluations until the optimum is found and the standard deviation of this average. The population size is 1600. The subpopulation size in the island-model distributed GA is 200 and the number of subpopulations is 8.

GA type		level 1	level 2	level 3	level 4
cohort GA with new placement implementation	average	6577	49952 (80%)	193042 (20%)	420068(5%)
	std. dev	4412	25705	71593	
Island-model distributed GA	average	3353	36117	76048	
	std. dev	185	5782	16013	
Canonical Serial GA	average	8210	38561	79776	
	std. dev	5458	8578	13681	
Canonical Serial GA with niching	average	8125	39345	84292	
	std. dev	5279	7854	17292	

Table 9: The average number of function evaluations until the optimum is found and the standard deviation of this average. The population size is 4000. The subpopulation size in island-model distributed GA is 200 and the number of subpopulations is 8.

The new offspring placement implementation gives the cohort GA obviously better performance, but during the experiments we still can see that the individuals tend to be accumulated in a small number of cohorts instead of spreading among all the cohorts after a certain number of cycles. Table 10 and 11 illustrate the degree of accumulations of the original and new offspring placement implementations respectively. The number of cohorts here is 20 and initial cohort sizes are 20. With the original offspring placement, after only 20 cycles and about 1000 function evaluations, the population has prematurely converged with the maximum fitness 1.86, RR level 0. The individuals have accumulated in 10 cohorts instead of being spread among 20 cohorts. With the

new offspring placement, after 160 cycles and 8000 function evaluations, the population has converged with the maximum fitness 4.3, RR level 1.

number of circles	number of evaluations	maximum fitness value	cohort sizes
Initial sizes			{20, 20, 20, 20, 20, 20, 20, 20, 20, 20, 20, 20, 20, 20, 20, 20, 20, 20, 20, 20}
5	228	1.84	{0, 0, 0, 0, 0, 43, 35, 42, 53, 45, 44, 30, 22, 16, 22, 10, 5, 8, 6, 19}
10	564	1.84	{1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 70, 61, 57, 59, 58, 51, 22, 15, 5, 1}
20	1316	1.86	{80, 56, 60, 56, 51, 41, 28, 19, 7, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0}
30	2020	1.86	{0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 61, 58, 62, 57, 51, 44, 44, 17, 3, 3}
40	2688	1.86	{65, 50, 65, 57, 64, 43, 34, 14, 8, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0}
80	5284	1.86	{48, 67, 66, 61, 48, 40, 28, 20, 15, 6, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0}
160	10524	1.86	{45, 53, 63, 56, 64, 44, 39, 29, 6, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0}

Table 10: Maximum fitness values and cohort sizes recorded in a type cohort GA run at some number of circles. Original offspring placement implementation.

number of circles	number of evaluations	maximum fitness value	cohort sizes
Initial sizes			{20, 20, 20, 20, 20, 20, 20, 20, 20, 20, 20, 20, 20, 20, 20, 20, 20, 20, 20, 20}
5	204	1.84	{6, 2, 5, 1, 0, 19, 17, 17, 20, 15, 11, 13, 18, 28, 31, 29, 39, 41, 52, 35}
10	376	1.96	{23, 17, 27, 15, 16, 12, 3, 3, 2, 0, 8, 12, 15, 27, 30, 26, 35, 41, 51, 36}
20	940	2.02	{14, 9, 23, 6, 35, 10, 18, 15, 8, 15, 32, 33, 49, 44, 33, 25, 15, 10, 5, 0}
30	1308	2.26	{17, 13, 7, 5, 2, 5, 3, 0, 0, 0, 57, 35, 59, 56, 44, 34, 21, 17, 14, 10}
40	2008	2.34	{30, 44, 24, 30, 20, 12, 46, 27, 43, 39, 42, 8, 10, 16, 6, 1, 1, 0, 0, 0}
80	4320	3.52	{1, 39, 52, 57, 30, 54, 69, 33, 37, 16, 9, 0, 1, 0, 0, 1, 0, 0, 0, 0}
160	8456	4.3	{6, 62, 82, 74, 77, 47, 43, 7, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0}
180	9728	4.3	{21, 0, 0, 0, 0, 0, 0, 0, 0, 119, 111, 78, 48, 20, 2, 0, 0, 0, 0, 0}

Table 11: Maximum fitness values and cohort sizes recorded in a type cohort GA run at some number of circles. New offspring placement implementation.

The result of this accumulating is a kind of premature convergence. A large amount of individuals tend to be put within a few cohorts, indicating that their fitness values are similar to a degree that they could not be separated by the current offspring placement strategy. The similarity of the fitness might indicate the similarity of the structure of the chromosome. This might be due to

one of RR's shortcoming: saturation effect. Saturation effect refers to the fact that a large part of the chromosome must be fixed to achieve a further RR level. Therefore, before RR achieves a new level, a long period of search needs to occur to get substantial improvement in fitness. This might result in most of the chromosomes in the whole populations being similar, or there might exist some groups of chromosomes, with each group having similar chromosomes with similar fitness values. When crossover happens within the same cohort among the similar chromosomes, there is little chance to get any improvement.

3.1.3 Varying crossover candidate and offspring placement

To give a better chance for the individuals with more different structures to mate, I tried two strategies that will enable inter-cohort crossover: choose different crossover candidates and non-deterministic placement of the offspring.

To change the crossover candidate, one candidate is still selected from the current cohort, and with the probability 0.1, another candidate is randomly chosen from another cohort. In another words, one-tenth of the second candidates do not come from the current cohort.

In non-deterministic offspring placement, the offspring may be placed in a randomly selected cohort, rather than using the calculation of d. The probability of this random placement was also set to 0.1.

Tables 12, 13 and 14 list the results of employing the two strategies.

cohort size		# of cohort = 20	# of cohort = 35	# of cohort = 50
20	average	2729	2780	3886
	std. dev	1046	1737	2767
40	average	4177	5533	4924
	std. dev	3227	2556	3358
80	average	5428	8596	8770
	std. dev	2573	4161	4542

Table 12: The average number of function evaluations to achieve level 1 and the standard deviation of this average.

cohort size		# of cohort = 20	# of cohort = 35	# of cohort = 50
20	average	59376 (65%)	23229 (40%)	40770 (55%)
	std. dev	54162	24682	49555
40	average	72828 (30%)	29839 (65%)	33986 (50%)
	std. dev	104787	18161	19129
80	average	37266 (60%)	52150 (70%)	43366 (40%)
	std. dev	15101	23541	11095

Table 13: The average number of function evaluations to achieve level 2 and the standard deviation of this average.

cohort size		# of cohort = 20	# of cohort = 35	# of cohort = 50
20	average	36949 (5%)		
	std. dev			
40	average	138433 (15%)		40184 (5%)
	std. dev	91292		
80	average	152437 (10%)	118787 (15%)	74651 (5%)
	std. dev	40165	62362	

Table 14: The average number of function evaluations to achieve level 3 and the standard deviation of this average.

Comparing Tables10, 11 and 12 with Tables 3, 4, and 5, we can see that, with this setting, the cohort GA needs more function evaluations to achieve a certain level. These results show that employing the inter-cohort crossover didn't improve the cohort GA's performance. This might be due to a potential defect of the implementation of inter-cohort crossover. This might also be due to the fact that the RR's saturation effect is too strong to be overcome by the cohort GA.

In an attempt to alleviate the premature convergence, I also tried to reduce the selection pressure by changing the implementation of deletion from deleting from second half of the cohorts to deleting from all cohorts except the current cohort. With this change, the performance of the cohort GA went down.

This shows that the selection pressure alone is not a big factor in causing premature convergence of the cohort GA on RR.

3.1.4 Summary

The experiments described in this section investigate the cohort GA's performance on the RR function, Several parameters and implementations are changed in order to see what their effects are on the cohort GA's performance.

The results of these experiments indicate that the number of cohorts and the offspring placement have the most effect on cohort GA performance on RR. The number of cohorts equal to 20 gives the best overall performance, especially when the initial cohort size is 80. But this ratio of cohort size and the number of cohorts can not be generalized. A new implementation of offspring placement in an attempt to spread all the individuals among all the cohorts gives a great improvement in the cohort GA. But compared with an island-mode distributed GA and two canonical serial GA's, the best cohort GA's performance is still worse than the other two. The RR function's drawbacks might cause this version of the cohort GA to fail. But there might be some practical problems that also have the characteristics of the RR function. So there is a need to find another way to implement the cohort GA's central idea and avoid the problems found here.

3.2 Comparison results using HDF

To use HDF as a direct tool to verify the cohort GA's effect on hitchhiking, I generated a small HDF with chromosome size equal to 80. It is easier to look directly at the intron loci when the chromosome size is small. The measurement I

use is to calculate the total number of zeros and the total number of ones at each intron locus in the whole population. Then the sum of their absolute differences should indicate the degree of convergence of intron loci.

Because HDF don't have an explicit concept of level, I measured maximum fitness values that the different GA's achieved given the number of function evaluations equal to 2500 and 6000. I chose the maximum number of function evaluations as 6000 because the GA's likely either have already found the optimum solution or have prematurely converged at that time. In these experiments, I only compared cohort GA, canonical serial GA and canonical serial GA with niching techniques.

3.2.1 Varying population size

In the initial experiments I used Holland's original implementation for the cohort GA. The population size equals to 200, 400 and 800. The number of cohorts was 20 and the cohort sizes were 10, 20 and 40. Tables 15 and 16 list the comparison results for population size 200. Since the results are quite similar, I did a t-test on the data from all 20 runs in order to test whether the difference is significant.

	Canonical Serial GA (1)		Canonical Serial GA with niching (2)		cohort GA(3)	
	maximum fitness after 2500 evaluations	sum of the differences in intron part	maximum fitness after 2500 evaluations	sum of the differences in intron part	maximum fitness after 2500 evaluations	sum of the differences in intron part
average	13.90	3896.60	14.30	2404.80	8.90	1982.50
std. dev	3.34	759.90	2.74	486.19	1.71	327.59

	t-test on maximum fitness	t-test on sum of the differences
(1,2)	0.597465326	**2.02091E-06**
(1,3)	**1.59564E-05**	**4.27796E-08**
(2,3)	**3.67275E-07**	**0.003991985**

Table 15: Comparison of the maximum fitnesses reached by three GA's after 2500 function evaluations and the sum of the differences in intron parts. Population size is 200. The small table shows the t-test results. The numbers in parentheses represent the source of the data. For example, (1, 2) means the data from the 20 runs on canonical serial GA and canonical serial GA with niching. The cell with bold font indicates the difference is significant at 0.05 level.

	Canonical Serial GA (1)		Canonical Serial GA with niching (2)		cohort GA(3)	
	maximum fitness after 6000 evaluations	sum of the differences in intron part	maximum fitness after 6000 evaluations	sum of the differences in intron part	maximum fitness after 6000 evaluations	sum of the differences in intron part
average	14.35	4688.90	15.60	2039.90	9.65	2368.20
std. dev	3.66	2214.26	3.05	472.10	1.53	386.05

	t-test on maximum fitness	t-test on sum of the difference
(1,2)	0.19023366	**8.24062E-05**
(1,3)	**2.67546E-05**	**0.000161509**
(2,3)	**1.98192E-08**	**0.034474718**

Table 16: Comparison of maximum fitness reached by three GA's after 6000 function evaluations and the sum of the difference in intron part. The small table shows the t-test results.

The results showed that with both numbers of function evaluations (2500 and 6000), the canonical serial GA with niching gets better results. Especially for the sum of differences of the intron parts, the canonical serial GA with niching does significantly better than without niching. This indicates that by employing a niching technique, the convergence of the intron part is greatly reduced. Even though the convergence of the intron part of the cohort GA is at about the same level as that of the canonical serial GA with niching, its maximum fitness level is

significantly lower than those of the other two GA's, so its the lower convergence should not be given much weight.

The results for population sizes 400 and 800 are similar to those for population size 200, except that the results didn't have much difference when the number of function evaluations reached 2500, because when the population sizes are larger, it took more evaluations for initialization of the initial populations. But as the search went on, the same pattern as with population size 200 appeared.

In these HDF tests, the cohort GA always showed earlier convergence, and the phenomenon of individuals accumulating in a small number of cohorts existed, as with the RR function. Thus I applied the new implementation of offspring placement from the RR runs to the HDF in the next experiments.

3.2.2 Varying the offspring placement implementation

The same offspring placement implementation as in RR was tried for HDF. Thus, in the calculation of the cohort into which the offspring would be placed, I used

$$doub = \lceil (nocoh - 1) + (u - umin) \times (2 - nocoh + 1) / (umax - umin) \rceil$$

instead of

$$doub = \lceil 2 \times umax / u \rceil.$$

Tables 17 and 18 list the comparison results between the original cohort GA and the cohort GA with the new implementation. The population size is 400. The t-tests on the two paired data sets indicate no significant difference between the two implementations. From the observations of the experiments, I found out that the new implementation of offspring placement in HDF didn't alleviate the degree of accumulation of individuals in some small number of cohorts. With population size 800, I got the similar results. These results indicate that we cannot generalize the effect of the new implementation on RR to other test functions or practical problems while using the cohort GA.

	original cohort GA		cohort GA with new placement	
	maximum fitness after 2500 evaluations	sum of the differences in intron part	maximum fitness after 2500 evaluations	sum of the differences in intron part
average	9.05	2974.4	8.45	2847.4
std. dev	1.57	286.94	1.36	831.20

Table 17: Comparison of maximum fitness reached by two cohort GA's after 2500 function evaluations and the sum of the differences in intron part.

	original cohort GA		cohort GA with new placement	
	maximum fitness after 6000 evaluations	sum of the differences in intron part	maximum fitness after 6000 evaluations	sum of the differences in intron part
average	9.35	3419	8.9	3966.15
std. dev	1.87	794.18	1.48	477.89

Table 18: Comparison of maximum fitness reached by two cohort GA's after 6000 function evaluations and the sum of the differences in intron part.

Besides the above experiments, I also tested the relationship between the number of cohorts and the cohort size by conducting paired experiments. The pairs included 10/20 versus 20/10, 10/40 versus 20/20, and 10/80 versus 20/40 (x/y, x representing the number of cohorts and y representing the cohort size). Each pair had the same population size. The results of these experiments showed that the ratios didn't have a significant impact on the performance of the cohort GA.

3.2.3 Summary and discussion

The results of the experiments on HDF showed that the canonical serial GA with niching could reduce the convergence of the intron part very much. This means that crowding and incest reduction did maintain the population diversity and reduce premature convergence. The results also confirmed that the

implementation of the cohort GA I used might have some defects in comparison to the implementation used by Holland.

One particular parameter setting (mutation rate) may have had a negative effect on the HDF runs reported here for the non-cohort GA runs. In contrast with the cohort GA, which did one or more mutation each individual in the current cohort with probability ½, the non-cohort GA rates were set lower. I used the same mutation rate per bit (0.0002) as on RR. It gives a relatively low mutation rate (0.016) per chromosome. Another way of viewing the difference is that in the cohort GA, nearly all new individuals are generated by crossover, and half are also subject to mutation, whereas in these non-cohort GA runs, of RR, about ¾ of the new individuals resulted from crossover and ¼ from mutation. However, for HDF was about 90% new individuals resulted from crossover and only about 10% from mutation. With an increased rate, the non-cohort GA's might give even better performance, and this could be explored further.

CHAPTER 4

CONCLUSIONS

The cohort genetic algorithm is designed as a means of reducing premature convergence, specifically, hitchhiking. In this thesis, I investigated one version of the cohort GA's performance on the RR function and the HDF and compared the cohort GA with canonical serial GA and island-model distributed GA in order to see how well the cohort GA works in comparison with other techniques for preventing premature convergence. The experiments showed that even though theoretically the cohort GA should work well in dealing with hitchhiking and be more efficient, the implementation affects its performance very much. This version of the cohort GA didn't perform better in any of the comparison tests due to another form of premature convergence, in which the individuals tend to accumulate in a few cohorts instead of spreading among all the cohorts.

Besides using the original implementation, I also tested different settings and implementations in order to see how some factors affect a cohort GA's performance. The factors included population size, offspring placement strategy, deletion strategy, and inter-cohort crossover. Among these factors, population size, which also includes the relationship between the number of cohorts and

cohort size and the offspring placement strategy, had the most significant effect on its performance. In particular, a new implementation of offspring placement in an attempt to spread all the individuals among all the cohorts gives a great improvement in the cohort GA on RR.

The experiments also showed that crowding and incest reduction performed very well in preventing premature convergence. The degree of intron convergence was greatly reduced after using these niching techniques.

The comparison results indicate that the potential defects exist in this version of the cohort GA, and the fact that a small change of placement of offspring among cohorts greatly improved the cohort GA's performance also suggests that further work on the cohort GA may be fruitful. Here are two suggestions for future work:

1) Set an upper limit on the cohort size during the run according to the initial cohort size, For example, if the initial cohort size is 20, the maximum cohort size during the run could be set to 35. In this way, the individuals are forced to spread among the cohorts. The calculation of which cohort an offspring is to be placed in could be done as usual, but if its cohort size has already reach the upper limit, the offspring could be placed in another cohort that has fewer individuals. The new receiving cohort could be calculated deterministically or probabilistically.

2) Use the mean fitness value (umean) in the offspring placement strategy. Place the individuals with fitness values between umin and umean into the first half of the cohorts and place the individuals with fitness value between umean and umax into the second half of the cohorts. Also follow the principle that the individual with higher fitness value should be put nearer the current cohort.

The detailed implementation issues regarding those changes need to be considered carefully. However, even a trial-and-error process would help us to better understand the cohort GA and to advance it.

APPENDIX

GLOSSARY

allele The value at a particular locus in a chromosome.

basic building blocks The smallest groups of loci that may affect the fitness of an individual of a GA population.

building block A collection of particular alleles at particular loci that contribute to the fitness of an individual of a GA population.

building block hypothesis The hypothesis that the GA finds solutions by finding low order, highly-fit building blocks and recursively combining these building blocks to form higher-order, even-more-fit building blocks until a solution is found.

crossover rate The probability that two individuals that have been chosen to be parents will crossover.

early convergence When the entire population of the GA converges to a sub-optimal solution. This state may be temporary (the evolution of solutions continues) or permanent (no additional building blocks can be found).

function evaluation The evaluation of a single individual by the GA.

Hamming distance The number of locations that differ in two binary individuals of equal length. For example, the strings 01011 and 11001 have a Hamming distance of two since bits zero and three are different.

loci (pl of **locus**) The positions of bits (or other values) in a chromosome.

mutation rate The probability that a single bit (locus) of a GA individual will be complemented (changed). Sometimes, instead, the probability that any mutation will occur on a chromosome at a given time.

intron part (also called non-coding area) Bits (or loci) between two building blocks of a GA function which are completely ignored by the fitness function.

REFERENCES

[1] J. H. Holland, "Adaptation in Natural and Artificial Systems," in *University of Michigan Press*, 1975

[2] J. H. Holland, "Cohort Genetic Algorithms (CGA) and Hyperplane-Defined Functions (HDFs)," Version 1.0, Mathematica 3.0 package, Edited and packaged by Theodore C. Belding, [Online] Available http://www-personal.umich.edu/~streak/software/jhh-hdf-1.0.tar.gz, 1998.

[3] D. E. Goldberg, *Genetic Algorithms in Search, Optimization & Machine Learning*. Addison-Wesley, 1989.

[4] M. Mitchell, S. Forrest, and J. H. Holland, "The Royal Road for Genetic Algorithms: Fitness Landscapes and GA Performance," In *Towards a Practice of Autonomous Systems: Proceedings of the First European Conference on Artificial Life*, pp. 245-254, 1992.

[5] S. Forrest and M. Mitchell, "Relative Building-Block Fitness and the Building-Block Hypothesis" In *Whitley, L. D. (Ed.) Foundations of Genetic Algorithms 2* , pp. 109-126, San Mateo, 1992.

[6] M. Mitchell, J. H. Holland, and S. Forrest, "When Will a Genetic Algorithm Outperform Hill Climbing?" In *Cowan, J. D., Tesauro, G. & Alspector, J. (Eds.), Advances in Neural Information Processing Systems 6*, San Mateo, CA: Morgan Kaufmann, Santa Fe Institute working paper 93-06-037, 1994.

[7] S. Forrest, and M. Mitchell, "What Makes a Problem Hard for a Genetic Algorithm? Some Anomalous Results and Their Explanation," *Machine Learning*, vol 13, pp. 285-319, 1993.

[8] A. S. Wu and R. K. Lindsay, "Empirical Studies of the Genetic Algorithm With Noncoding Segments," *Evolutionary Computation*, 3(2), pp. 121-147, 1995.

[9] A. S. Wu, "Non-Coding DNA and Floating Building Blocks for the Genetic Algorithm," PhD thesis, University of Michigan, 1996.

[10] D. Whitley, "A Genetic Algorithm Tutorial," *Statistics and Computing*, vol 4, pp. 65-85, 1994.

[11] D. Whitley and J. Kauth, "Genitor: a Different Genetic Algorithm," in *Proc. of the Rocky Mountain Conf. on Artificial Intelligence*, Denver, pp.118-130, 1988.

[12] F. Vavak and T. C. Fogarty, "Comparison of Steady State and Generational Genetic Algorithms for Use in Nonstationary Environments," in *Fogarty, T. C. (Ed.) Evolutionary Computing, AISB Workshop Selected Papers, Lecture Notes in Computer Science 1143 (Spring-Verlay)*, pp. 297-304, 1996.

[13] S. C. Lin, W. F. Punch III and E. D. Goodman, "Coarse-Grain Parallel Genetic Algorithms: Categorization and New Approach," *Parallel & Distributed Processing*, Dallas TX, 1994.

[14] T. Jones, "A description of Holland's Royal Road function," *Evolutionary Computation*, 2(4), pp. 411-417, 1995.

[15] E. D. Goodman, "An Introduction to GALOPPS, The 'Genetic Algorithm Optimized for Portability and Parallelism' System", [Online] Available http://GARAGe.cps.msu.edu/software, 1996.

[16] E. B. James, "Reducing Bias and Inefficiency in the Selection Algorithm," in *Proceedings of the second international conference on genetic algorithms*, 1987.