My GA Code

Peter G. Anderson,

Computer Science Department

Rochester Institute of Technology,

Rochester, New York

anderson@cs.rit.edu http://www.cs.rit.edu/

February 2004
Abstract

Presenting my own genetic algorithm code (a steady-state algorithm).
Also presenting *params*, a method for parameterizing code.
And some methods for experimenting using shell-scripts, gnuplot, etc.
The Structure of My GA Code

Start-up

Main program loop

Tournament parent and loser selection

Initialization subroutine (somewhat problem dependent)

“fv” — the fitness evaluation code (is problem dependent)

Crossover

Mutation

Randomization
GA Main Program: Init

main( argc, argv ) int argc; char **argv; { int who;
  params( argc, argv ); init();

  for( who = 0; who < POP_SIZE; who++ )
    fitness[who] = fv(who);
  printf( "End of initial pop\n. . . Now evolve!\n" );

  . . . main loop goes here . . .

}
Notes on GA Main Program: Init

1. arg gives the name of a param file

2. params(argc, argv) processes that

3. POP_SIZE is an example of a run-time param
GA Main Program: Loop

for (trial = 0; trial < LOOPS; trial++) {
    if (hero >= MAX_HERO) {
        printf( "Goal reached: %d\n", hero ); break;
    }
tournament( &p1, &c1 ); tournament( &p2, &c2 );
make_children( p1, p2, c1, c2 );
if (MUT_RATE > 0.0) { mutate( c1 ); mutate( c2 ); }
    fitness[c1] = fv(c1);
    fitness[c2] = fv(c2);
}
printf( "%d evaluations", fitness_evals);
printf( "Hero = %d\n", hero );
Notes on GA Main Program: Loop

1. This is the steady-state algorithm. It puts the children right back into the population. There is no notion of “generation”

2. LOOPS and MUT_RATE are examples of run-time params
Tournament: Select a Parent and a Casualty

void tournament( winner, loser ) int *winner, *loser; {
    int size = tournament_size, i, winfit, losefit;
    for( i = 0; i < size; i++ ) {
        int j = random_int( POP_SIZE );;
        if( i==0 || fitness[j] > winfit ) {
            winfit = fitness[j];
            *winner = j;
        }
        if( i==0 || fitness[j] < losefit ) {
            losefit = fitness[j];
            *loser = j;
        }
    }
}
Initialization Subroutine

init()
{
    int PS, i, j;
    srand48( seed );
    hero = 0;
    PS = POP_SIZE;
    p = unchar_matrix( 0, PS-1, 0, N-1 );
    fitness = ivector( 0, PS-1 );
    MAX_HERO = N; /* for problem #1 */
    for( i = 0; i < POP_SIZE; i++ ) {
        for( j = 0; j < N; j++ ) { p[i][j] = random_int( 2 ); }
    }
}
Initialization Subroutine

We use matrix creation code from *Numerical Recipes in C*. Their subroutines perform the malloc for us.

1. `ivector( low, high )`

2. `imatrix( low1, high1, low2, high2 )`
Fitness Calculation

```c
int fv( who )
{
    int i, the_fitness = 0;
    fitness_evals++;
    for( i = 0; i < N; i++ )
        the_fitness += p[who][i];
    if( print_every_fitness )
        printf( "%4d fitness: ... ");

    if( the_fitness > hero )
    {
        hero = the_fitness;
        printf( "New hero %4d ... ");
    }
    return( the_fitness );
}
```
Uniform Crossover

make_children( p1, p2, c1, c2 ) int p1, p2, c1, c2;
{
    int i, j;

    for( i = 0; i < N; i++ ) {
        if( random_int(2) ) {
            p[c1][i] = p[p1][i];
            p[c2][i] = p[p2][i];
        } else {
            p[c1][i] = p[p2][i];
            p[c2][i] = p[p1][i];
        }
    }
}
Mutation

Randomize each bit with probability MUT_RATE

```c
mutate( who ) int who;
{ int j;

    for( j = 0; j < N; j++ ) {
        if( MUT_RATE > drand48() ) { p[who][j] = random_int(2); }
    }
}
```
Randomization Function

/* returns integer in range 0 <= r < n */
int random_int( n ) int n;
{
    return( (int) ( n * drand48() ) );
}
This is experimental programming so, we need to be able to experiment!

There are many parameters for a GA:
mutation rate, tournament size, population size, random seed, ...

I don’t want to write a specialized parameter-reading program, a GUI, lots of #defines, etc.

I want to perform and keep track of lots of experiments.
Params: Parameterizing Code

The following makefile and other files illustrate *params*.

It is _very_ easy to add new parameters to a program.
Params: Parameterizing Code

params.d defines and initializes some variables, the “parameters.”

params is read in at run-time to redefine the params.

params is like a list of assignments: the last one succeeds; and comments can track your progress.

Shell scripts can build and modify params files and invoke ga.

params.d and params are the only files the programmer needs to think about.
Makefile for ga

CFLAGS = -O -L/usr1/lib -I/usr1/include
LIBS = -lmu
INC = nr.h nrutil.h params.h headers.h
MYOBJ = ga.o nrutil.o params.o
$(MYOBJ): $(INC)

ga: $(MYOBJ) $(INC)
    cc $(CFLAGS) -o $@ $(MYOBJ) $(LIBS)

params.h params.c: params.d
    ./make-params

clean:
    rm -f $(MYOBJ) ga params.[ch]
params.d: Names & Values

%int
print_the_params 1
POP_SIZE 100
tournament_size 2
print_every_fitness 1
print_every_hero 1
LOOPS 1000000
debug 0
N 10

%float
MUT_RATE 0.001
seed 0.0 # random seed
int print_the_params = 1;
int POP_SIZE = 100;
int tournament_size = 2;
int print_every_fitness = 1;
int print_every_hero = 1;
int LOOPS = 1000000;
int debug = 0;
int N = 10;
float MUT_RATE = 0.001;
float seed = 0.0;

params( argc, argv )
{ ..
}
File params.h Produced by make-params

extern int print_the_params;
extern int POP_SIZE;
extern int tournament_size;
extern int print_every_fitness;
extern int print_every_hero;
extern int LOOPS;
extern int debug;
extern int N;
extern float MUT_RATE;
extern float seed;
params Overrides Default Values

print_the_params 1
POP_SIZE 100
print_every_fitness 0
print_every_hero 1
LOOPS 1000000
N 100
MUT_RATE 0.001
seed 0.0 # random seed
## everything worked up to here
MUT_RATE 0.000001 this is very small!!!!
## this failed:
POP_SIZE 10000
## now it worked!
Setting the Parameters

How do we choose the “params”?  

There is a statistical discipline: experimental design

For the present, we will treat this issue informally.

Try different setting and get a problem to work.

Then systematically try various settings.
Params for Prob. #1: Test Seed

%int
print_the_params  1
POP_SIZE     20
tournament_size  2
print_every_hero  1
uniform    1
FITNESSES   20000
N       200
problem_number  1
%float
MUT_RATE  0.001000
seed      0.000000

We then ran this with many seed values.
bash Script to Test seed Settings

Run with the previous parameters, and seed = 0.1000 (0.001) 0.2000

(( SEED = 1000 ))
while (( SEED < 2000 ))
do
  ( cat params; echo seed 0.$SEED ) > ,params11
  ./ga ,params11 | grep Stop | awk '{ print $3 }'
(( SEED = SEED + 1 ))
done
Seed Test Results

We ran 1,000 tests with different seed values

The number of fitness evaluations to solve the problem was:

<table>
<thead>
<tr>
<th>count</th>
<th>fitnesses</th>
</tr>
</thead>
<tbody>
<tr>
<td>124</td>
<td>2620</td>
</tr>
<tr>
<td>126</td>
<td>2860</td>
</tr>
<tr>
<td>125</td>
<td>3064</td>
</tr>
<tr>
<td>124</td>
<td>3096</td>
</tr>
<tr>
<td>125</td>
<td>3236</td>
</tr>
<tr>
<td>124</td>
<td>3260</td>
</tr>
<tr>
<td>125</td>
<td>3262</td>
</tr>
<tr>
<td>127</td>
<td>4256</td>
</tr>
</tbody>
</table>
Seed Variation Results

We ran the system 1,000 times with different seed values
Params Settings for Problem #1

print_the_params  1
POP_SIZE   200
tournament_size  2
print_every_hero  1
uniform       1
FITNESSES     10000
N            400
problem_number 1
MUT_RATE     0.002000
seed         0.000000

These settings worked!
Result: Stopped after 8824 fitness evals. Hero = 400
Shell Script to Test POP_SIZE

Run with the previous parameters, and POP_SIZE = 5 (5) 495

```bash
(( P = 5 ))
while (( P < 500 ))
do
  ( cat params; echo POP_SIZE $P ) > ,params
  ./ga ,params | grep Stop | awk '{print pop, $3}' pop=$P -
  (( P = P + 5 ))
done
```
Population Test Results: Tiny & Huge Pops Failed

We only allowed 10,000 fitness evaluations.
Many Fitnesses. Use param `print_every_fitness 1`