GAP

The Genetic Algorithm Programming Library

The Official Tutorial

Welcome to the GAP tutorial!

This tutorial assumes that GAP is already correctly installed on your system and that you, the reader, have at least a rudimentary understanding of C.

Contents:

GAP is an abbreviation for Genetic Algorithm Programming. It is a code library for writing programs which implement some kind of genetic algorithm.

GAP-Lib, henceforth called GAP, was written because to provide a simple yet flexible and powerful base to build such programs upon. The focus has been on providing the programmer with as much as possible while still allowing her to change almost any aspect of how the library works.
As an example one can point to that While GAP is primarily bit-oriented, it is still written in such a way that one is not restricted to bit representations.

GAP is not blazingly fast, but it is blazingly good (Advertisement of the day).

1.1 What can I do with GAP?

GAP enables you to manage populations, generate statistics and evolve populations with a minimum of fuss. It can be used for everything from simple function optimization to empirical studies of GAs themselves. It is however limited to GA.

Some examples of where GAP has been successfully used include:

* Function optimization
* Evolving rule-based input/action systems.
* Comparing different representations for problems solved with GA.

2.0 Setting up the problem.

The objective of this tutorial will be to implement a genetic algorithm based program to optimize the function f(x)=x+sin(32x) in the range x=[0,¶] for the highest value of f(x). The actual source code presented in this tutorial is also included as a separate file.

To accomplish our objective, we will need to set up a fitness function, design a template for individuals and implement this.

2.1 Designing the individuals.

In this case we will simply model the individuals as integers and let this integer represent a fraction of the range from 0 to PI with 0 meaning 0 and the maximum value of the integer representing PI.

In code this would look something like:

```
struct Polyphant {
   unsigned long value;
};
```

If one wants to, one could omit the struct declaration in this case. But I personally find it neater to keep it as to show that the individuals are distinct entities and not just a primitive datatype.

For this program, only two functions are actually needed. A fitness function and a main function. For those of you not familiar with C, main is the starting point of any program - and the program does need somewhere to start. It is however desirable to have one more function, namely one to initialize the individuals when creating the population (In reality we could use the built-in random-initialization function for this.).

The main function of this program would look something like this :

```
int main(void)
{
int i;
struct Population *Pop;
struct Polyphant *Individual;
struct TagItem EvolveTags[]={
   {EVL_Evaluator,fitfunc},
   {TAG_DONE,0L}
};
struct TagItem CreateTags[]={
   {POP_Init,init},
   {TAG_DONE,0L}
};

EnterGAP(1);

Pop = CreatePopulation(20,sizeof(struct Polyphant),CreateTags);

for(i=0;i!=25;i++) {
   Pop = Evolve(Pop,EvolveTags);
   printf("Generation %d: Max = %lf\n",i+1,Pop->Stat.MaxFitness);
}

Individual = Pop->Stat.Max;

printf("After %d generations:\n",i);
printf("Best value = %lf.\n",Pop->Stat.MaxFitness);
printf("For f(%lf).\n",IRange(Individual->value,0,PI));

DeletePopulation(Pop);

return(0);
}
```

this:

```
void init(struct Polyphant *Polly)
{
/* Rnd() only returns values between 0 and max 2147483646 (30 bits) */
Polly->value = Rnd(0x7ffffffe)^(Rnd(0x7ffffffe)<<2);
}

double fitfunc(struct Polyphant *Polly)
{
double x;
x = IRange(Polly->value,0,PI);
return(x+sin(32*x));
}
```

2.3 Choosing parameters.

Choosing the right parameters for your GA can be very important for its performance. In GAP, most of the parameters can either be specified to the Evolve() function or should be built-in into the fitness, crossover and mutation functions. One other parameter however which is not specified in either of those places is the size of the population.

In the above code, the size of the population was arbitrarily set to 20 individuals. You might wish to experiment with this value to see if the behaviour of the GA changes.

Regarding parameters built into the fitness function and the representation of the individuals themselves, we have an example of this in the fact that we have let an integer represent a real number between 0 and ¶.

Lastly, we have the parameters one can specify to the Evolve() function. Evolve() takes two formal parameters, one which is the population and one which is a list of parameters on the form {{Parameter,Value},{},...}. Of the parameters in this list only two are necessary, EVL_Evaluator - which is used to specify the fitness function to use, and TAG_DONE which denotes the end of the parameter list (Taglist).

Other tags (parameters) in the list which you might want to try for this problem are EVL_Select and EVL_Elite.

If you have now written your main, fitness and initialization functions, you need only write a little more code and you will be ready to test your program.

The prototypes for Evolve(), CreatePopulation() etc. are in the file GAP.h so in the beginning of your program you should include <GAP.h> for these.

You should also have prototypes for your fitness and initialization functions in addition to having defined the type for the individuals. So the beginning of your program could look something like this:

```
#include <stdio.h>
#include <GAP.h>
#include <math.h>

struct Polyphant {
   unsigned long value;
};

#ifndef   PI
#define   PI 3.14159265359
#endif

void init(struct Polyphant *);
double fitfunc(struct Polyphant *);
...
...
```

Now if you have finished writing the program, all you need to do is to compile it before you can test it. Assuming you are using an ISO-C compliant C compiler under some UNIX flavour and have named your source file 'tutorial.c' :

```
cc tutorial.c -o tutorial -lgap -lm
```

The compiler will probably complain about the line containing {EVL_Evaluator,fitfunc} with a message like "initialization makes integer from pointer without a cast". If you want to you can perform the cast and rewrite it like {EVL_Evaluator,(IPTR)fitfunc}.

Assuming no errors occurred all you have to do now is to test your program, otherwise you now have the task of correcting those errors at hand. If you are unfamiliar with C, you might want to have a more experienced C-programmer to have a look at the code.

Now, finally, you have written your first program using GAP. A the prompt (assuming you work in a shell environment) type :

```
tutorial
```

And press enter (assuming you named the program 'tutorial').

The program should generate some output like this:

```
Generation 1: Max = 3.642083
Generation 2: Max = 3.664751
Generation 3: Max = 3.664751
Generation 4: Max = 3.667555
Generation 5: Max = 3.690620
Generation 6: Max = 3.690894
Generation 7: Max = 3.690893
Generation 8: Max = 3.690894
Generation 9: Max = 3.690893
Generation 10: Max = 3.690893
Generation 11: Max = 3.690894
Generation 12: Max = 3.690915
Generation 13: Max = 3.690916
Generation 14: Max = 3.690915
Generation 15: Max = 3.690916
Generation 16: Max = 3.690916
Generation 17: Max = 3.690916
Generation 18: Max = 3.690916
Generation 19: Max = 3.690916
Generation 20: Max = 3.690916
Generation 21: Max = 3.690916
Generation 22: Max = 3.690916
Generation 23: Max = 3.690916
Generation 24: Max = 3.690916
Generation 25: Max = 3.690916
After 25 generations:
Best value = 3.690916.
For f(2.784364).
```

One can readily see that this is far from an optimal value (The exact best value is 1+61*¶/64 (approximately 3.9943) for 61*¶/64) and I leave it as an exercise to improve the GA by choosing better parameters for this problem - which I will again take the opportunity to stress is very important.

(Hint: Add more items to the Evolve TagList.)

And this is the end of the tutorial, happy hacking!

-Peter Bengtsson (29/1-1999)