GAP-Lib Reference Document

TABLE OF CONTENTS

PURPOSE

The Genetic Algorithm Programming Library (GAP-Lib) is intended to make it easier to implement genetic algorithms for any purpose. The library itself implements most of the framework needed to handle one or several populations thus leaving the programmer free to concentrate on the purpose of her program.

Some work will still be needed, specifically setting up a genotype, creating a fitness function and in some cases functions for initializing, mutating, crossing and deleting individuals.

OVERVIEW

GAP-Lib: A Genetic Algorithm Programming Library.

A library for programming with genetic algorithms. The library features a simple yet flexible interface coupled with sensible default values and quite powerful built-in functionality to provide for both high and low-level programming.

The GAP-Lib is primarily bit-oriented and this is reflected in the existing default-functions for crossover and mutation. It is however possible to use almost any representation as long as the library itself only sees fixed-length individuals (Variable length individuals are possible though).

A typical program using the GAP-Lib will begin by initializing the GAP environment by calling EnterGAP() then create a population using CreatePopulation() followed by some number of calls to Evolve() and finally ending with calling DeletePopulation().

The current version as of 29-Jan-1999 is 0.72 (Version 0, Revision 72).

Current limitations include:

- Hard to implement dynamic-length individuals.
- Not thread-safe (Not dynamically linked).

STRUCTURES

GAP-Lib has three primary structures to keep track of data. One is the user-defined structure which describes an individual and though this structure is not always nessecary, it is highly recommended that you have one since it will help others to read your code.

The other structure is the population structure which in turn includes a statistics structure. A member-by-member explanation of these structures follow here:

```
struct Population {
   long   NumPolys;
   long   Generation;
   long   Flags;
   struct Popstat Stat;
   long   Bytes;
   void  *Polys;
   void  *Magic;
};
```

NumPolys - This is the number of individuals in the population.
Generation - This is how many times a generation shift has taken place.
Flags - Internal status, do not touch.
Stat - This is the statistics structure, see below.
Bytes - The byte-size of an individual (or its descriptor).
Polys - This is for the internal list of individuals, do not touch.
Magic - This is also internal, do not touch.

```
struct Popstat {
   double  AverageFitness;
   double  MedianFitness;
   double  TypeFitness;
   long    TypeCount;
   double  StdDeviation;
   double  MaxFitness;
   double  MinFitness;
   void   *Max;
   long    Generation;
};
```

AverageFitness - The average fitness of the population.
MedianFitness - The median fitness of the population.
TypeFitness - The type (most common) fitness value.
TypeCount - The number of individuals with the type fitness.
StdDeviation - The standard deviation of the fitness values.
MaxFitness - The fitness value of the fittest individual.
MinFitness - The fitness value of the least fit individual.
Max - A pointer to the fittest individual after the last call to Evolve().
Generation - The generation for which these statistics are valid.

The Popstat structure is read-only and it is important to remember that even if you copy the structure the Max pointer will become invalid the next time Evolve() is called.

TAGLISTS

A taglist is a list of pairs, the first member of the pair is the tag and determines what type of data the second member is. A typical member of a taglist could look like this:

```
{EVL_Elite,5}
 ^         ^
 |       |-> The data.
 |-> The type of data.
```

A complete taglist could look like this:

```
struct TagItem MyTaglist[]={
   {EVL_Evaluator, fitfunc}, /* Fitness function */
   {EVL_Elite,     5}, /* No. of elite individuals */
   {TAG_DONE,      0}  /* End-Tag */
};
```

The End-Tag, TAG_DONE, is a special tag common to all taglists. There are currently four such tags defined:

```
TAG_DONE    - Marks the end of a taglist.
TAG_END           - Equivalent to TAG_DONE.
TAG_IGNORE  - This tag is ignored.
TAG_MORE    - ti_Data is a pointer to a taglist with more tags,
              processing of the current taglist will be terminated.
```

As some of you might be a bit lazy ;-), there is also an alternative way of specifying taglists. At least in GAP-Lib there is. As an example we have the function Evolve() which also has and interface named EvolveT(), EvolveT() takes a variable number of arguments, the last of which make up the taglist. To use the above taglist with EvolveT() one would write:

```
EvolveT(Pop,EVL_Evaluator,fitfunc,EVL_Elite,5,TAG_DONE);
      ^
      |->This is not part of the taglist.
```

PRIMITIVE TYPES

GAP-Lib defines one primitive data type; IPTR. The IPTR is a type large enough to hold both an integer and a pointer, it is used for the data-part of a tagitem. IPTR can be considered equivalent to intptr_t as defined in the C9X C-Language standard-to-be.

NAME

CreatePopulation -- Allocates and initializes a population.
CreatePopulationT -- Varargs interface to CreatePopulation.

SYNOPSIS

```
struct Population *CreatePopulation(long int,long int,struct TagItem *);
struct Population *CreatePopulationT(long int,long int,...);

Pop = CreatePopulation(Num,PSize,TagList);
Pop = CreatePopulationT(Num,PSize,...);
```

FUNCTION

This function will allocate and initialize a population. If no initialization function is given, CreatePopulation() will simply randomize all bits in the created individuals. There is also a predefined initialization function which initializes every individual to a string of zero bits.

INPUTS

Num - Number of individuals to be created in this population.
PSize - The byte-size of the individuals in this population.
TagList - A pointer to a taglist.

TAGS

POP_Init (void (*)(void *)) - A pointer to a function or one of the values RAND_INIT or ZERO_INIT. Currently NULL is equivalent to ZERO_INIT. A function to initialize an individual should take a pointer to an individual and return nothing (void).

POP_Destruct (void (*)(void *)) - A pointer to a function to be called when deleting an individual. If you are allocating resources with a custom initialization function, then you should supply this tag. The function should take a pointer to an individual and return nothing (void).

POP_Cache (BOOL) - Set this to false if you are modifying the individuals between calls to Evolve() or if you really need to save memory. Default is TRUE which enables some caching of data.

RESULT

Pop - An initialized population structure or NULL if something failed.

NOTE

CreatePopulation() will fail if EnterGAP() has not been called previously.

BUGS

None known.

SEE ALSO

EnterGAP(), DeletePopulation()

NAME

Crossover -- Perform crossover on two bitstrings.

SYNOPSIS

```
void Crossover(void *,void *,int,int);

Crossover(void *Ind1,void *Ind2,int At,int Size);
```

FUNCTION

Performs one-point crossover of two bitstrings. The bitstrings must have the same length.

INPUTS

```
Ind1  - Pointer to the first bitstring (Individual).
Ind2  - Pointer to the second bitstring (Individual).
At    - Bit to perform crossover at.
Size  - Size of bitstring in bytes. (OBS!: _BYTES_!!)
```

RESULT

None.

NOTE

Note well that all bitstrings must consist of a whole number of bytes.
This is for reasons of simplicity and efficiency.

BUGS

None known.

SEE ALSO

Flip

NAME

DeletePopulation -- Delete a previously allocated population.

SYNOPSIS

```
void DeletePopulation(struct Population *);

DeletePopulation(Pop);
```

FUNCTION

Deletes a previously allocated population and frees all resources associated with it. If no custom deallocation function was given only the resources allocated by CreatePopulation() will be freed (If CreatePopulation() was called without a custom initialization function, this is probably what you want).

INPUTS

Pop   - Pointer to the population to be deleted.

RESULT

None.

BUGS

None known.

SEE ALSO

CreatePopulation()

NAME

EnterGAP -- Initialize GAP environment.

SYNOPSIS

```
void EnterGAP(int);

EnterGAP(Level);
```

FUNCTION

Initializes the GAP environment.

INPUTS

Level - Level of verbosity at startup, supported values range from 0 to 2 with 0=quiet, 1=normal, 2=verbose.

RESULT

0 for failure, non-zero for success.

EXAMPLE

```
int main(void)
{
      /* Do some stuff here */
...
if(EnterGAP(1)) {
...
...   /* Do everything else here. */
...
} else {
   fprintf(stderr,"Initialization failed.\n");
}
return(0); /* Finished, exit. */
}
```

BUGS

None known.

SEE ALSO

NAME

Evolve -- Performs generation shift on a population.
EvolveT -- Varargs interface to Evolve().

SYNOPSIS

```
struct Population *Evolve(struct Population *,struct TagItem *);
struct Population *EvolveT(struct Population *,Tag,...);

Pop = Evolve(Pop,TagList);
Pop = EvolveT(Pop,Tag0Type,...);
```

FUNCTION

This is the big one. Evolve performs a generation shift, taking a population and returning a new one.

INPUTS

Pop - Pointer to an initialized population structure.
TagList - Pointer to a taglist.

TAGS

EVL_Evaluator (double (*)(void *)) - Pointer to a function taking a pointer to an individual and returning its fitness value as a double. Note well that this tag is _required_. Also read the NOTE label further down.

EVL_Mutator (void (*)(void *,int)) - Pointer to a mutation function taking a pointer to an individual and its byte-size as an integer. This function should also decide if a mutation is to take place as it will be called exactly once for every individual in the population. NULL is a permitted value for this tag meaning that no mutation will take place. The default is to use a built-in function designed to mutate bitstrings.

EVL_Crosser (void (*)(void *,void *,int)) - Pointer to a function which performs crossover on two individuals. It should take two pointers to individuals and a byte-size parameter and return nothing (void). It will be called exactly once for every individual generated by crossover. NULL is not a permitted value for this tag. The default is to use a built-in function designed to perform crossover on two bitstrings.

EVL_Thermostat (double (*)(long,long)) - Pointer to a heat-regulating function for Boltzmann selection (TEMPERATURE). The default function is PopSize*(2.722-pow(1.0+1.0/Generation,Generation)) but this might change in later versions. The function taked the size of the population as first argument and the generation as second.

EVL_Elite (int) - Sets the number of top individuals to copy from one generation to the next without crossover (with the risk for mutation though). Setting this value high will result in a steady-state type of GA. The default value is 0.
Note!: Setting the Crowding flag currently alters the semantics of this tag! If Crowding is in effect the Elite number is the number of individuals not to generate. That is, in a population of eg. 20 individuals an Elite value of 15 would mean generate 5 new individuals.

```
EVL_Flags (unsigned long int) - Bulk initialization of flag tags
     available flags are:

      FLG_InitDumped - Same as EVL_InitDumped
      FLG_EraseBest  - Same as EVL_EraseBest
      FLG_Crowding   - Same as EVL_Crowding
      FLG_Statistics - Same as EVL_Statistics

     Example usage:  {EVL_Flags,FLG_Crowding|FLG_Statistics}
     **NOTE** If using EVL_Flags, you must explicitly set
     FLG_Statistics to generate statistics.

EVL_Dump (int) - Sets the number of bottom (worst) individuals to dump
     by replacing them with copies of the top (best) individuals.
     Default is 0.

EVL_Select (int) - Sets the select method used to determine parents
     when generating new individuals. Available methods are:

      DRANDOM    :      Double-random selection. A random individual
                        and one of those fitter than the first one
                        selected are chosen. (Default)

      FITPROP    :      Fitness proportionate selection.

      SIGMA      :      Sigma scaled fitness proportionate selection.

      TOURNAMENT :      Tournament selection (fast).

      INORDER    :      Inorder selection. The fittest individual is selected
                        together with the rest in descending order of fitness.

      TEMPERATURE:      Boltzmann scaled selection. The selection
                        pressure varies over time as determined by
                        a ’heat’ function. See also the EVL_Thermostat
                        tag.

EVL_Stats (BOOL) - Generate statistics. Generating statistics will
     increase processing time significantly compared to not doing it.
     If statistics are enabled, the fitness function might be called
     twice as many times. Once for every old individual for evaluation
     and once for every new individual for generating statistics.
     This is dependant on caching and previous state. When caching is
     disabled, then the fitness function will always be called exactly
     twice if generating statistics. Default is TRUE.

EVL_PreMutate (BOOL) - Mutate old generation instead of new. This
     will mutate the parent population before generating new
     individuals. Note that this is done after evaluation so that
     setting this tag to TRUE will mean that there is no nessecary
     connection between good genes and a high fitness - only a
     probability thereof (depending on the mutator function). This
     emulates mutation occuring in mature individuals in nature.

EVL_Newbies (int) - Number of new individuals to generate. The
     individuals to replace will be randomly selected from the old
     population. This could for example be used to keep the fitness
     of a population down when co-evolving populations.
```

EVL_Mensurator (double (*)(void *,void *,int)) - Pointer to a function taking two pointers to individuals in addition to their (equal) size and returning the absolute value of the distance (dissimilarity measure) between them. Default is to measure the Hamming distance between individuals.

EVL_Crowding (BOOL) - Use crowding replacement where each new individual generated replaces the individual most like itself. Note! This tag currently alters the meaning of the EVL_Elite tag!

EVL_InitDumped (BOOL) - If dumping individuals (see EVL_Dump above) initialize them instead of replacing them with copies of the fittest indiviuals.

EVL_EraseBest (BOOL) - If generating new random individuals (see EVL_Newbies tag above) replace the fittest individuals instead of random ones.

RESULT

Pop - A pointer to the population structure or NULL is something went horribly wrong.

NOTE

Note that Evolve always treats higher fitness values as better, this means that you must take care to transform your fitness values accordingly if needed before returning them.

BUGS

None known, but if there are any major bugs this is probably where they are.

SEE ALSO

CreatePopulation()

NAME

Flip -- Flip a bit in a bitstring.

SYNOPSIS

```
void Flip(void *,int);

Flip(Ind,At);
```

FUNCTION

Flips a bit in a bitstring. Bits are counted from lower addresses to higher.

INPUTS

```
Ind   - A pointer to the bitstring (individual).
At    - The bit-position to be flipped.
```

RESULT

None.

BUGS

None known.

SEE ALSO

Testbit()

NAME

GaussRand -- Generate a gaussian pseudo-random number.

SYNOPSIS

```
double GaussRand(double,double);

Val = GaussRand(My,Sigma);
```

FUNCTION

Generates a pseudo random number around My with a Gaussian distribution.

INPUTS

My   - Value around which to generate a random number.

Sigma - Standard deviation of the generated numbers.

RESULT

Val   - A random number.

BUGS

None known.

SEE ALSO

Rnd(), InitRand(), InRand(), InGaussRand()

NAME

HammingDist -- Measure the Hamming distance between two bitstrings.

SYNOPSIS

```
unsigned long int HammingDist(void *,void *,int);

distance = HammingDist(Ind1,Ind2,Size);
```

FUNCTION

Counts the number of differings bits in two bitstrings.

INPUTS

```
Ind1  - Pointer to the first bitstring (Individual)
Ind2  - Pointer to the second bitstring (Individual)
Size  - Number of _BYTES_ in each bitstring.
```

RESULT

The number of differing bits.

BUGS

None known.

SEE ALSO

NAME

InGaussRand -- Generate a bounded gaussian random number.

SYNOPSIS

```
double InGaussRand(double,double,double);

Val = InGaussRand(My,Sigma,Delta);
```

FUNCTION

Generates a pseudo random number in the range [My-Delta,My+Delta] with a Gaussian distribution around My.

INPUTS

My   - Value around which to generate a random number.

Sigma - Standard deviation of the generated numbers.

Delta - Delta from My in which to generate numbers.

RESULT

Val   - A random number.

BUGS

Setting Delta too small (<Sigma) will result in _very_ inefficient random number generation. Setting Delta < 0.0 makes the function enter an infinite loop.

SEE ALSO

Rnd(), InitRand(), InRand(), GaussRand()

NAME

InitRand -- Initialize pseudo-random number generator.

SYNOPSIS

```
void InitRand(long);

InitRand(seed);
```

FUNCTION

Initializes the internal pseudo-random number generator. This function should be called with an appropriate seed before any of the random number functions are called. Note that Evolve() also uses the random number functions. A default seed is supplied but it is not recommended to leave this as it is since every run will then be identical.

INPUTS

seed - A seed value for the pseudo random number generator.

RESULT

None.

EXAMPLE

```
#include <stdlib.h> /* For definition of NULL */
#include <time.h>
#include <GAP.h>
...
int main(void)
{
...
InitRand(time(NULL));
...
return(0);
}
```

BUGS

A seed value of 0 will not work properly.

SEE ALSO

Rnd(), InRand()

NAME

InRand -- Generate a bounded floating point pseudo-random number.

SYNOPSIS

```
double InRand(double,double);

Val = InRand(Lo,Hi);
```

FUNCTION

Generates a pseudo random number between Lo and Hi. The resolution of the generated number is in steps of (Hi-Lo)/2147483646.

INPUTS

```
Lo    - Lower bound of the range in which to generate a random
        number.
Hi    - Upper bound of the range in which to generate a random
        number.
```

RESULT

```
Val   - A random number in the range [Lo,Hi] (inclusive).
```

BUGS

None known.

SEE ALSO

Rnd(), InitRand()

NAME

IRange -- Map an integer onto a range.

SYNOPSIS

```
double IRange(unsigned long int,double,double);

v = IRange(Val,Lo,Hi);
```

FUNCTION

Maps an unsigned long integer onto the range [Lo,Hi] (inclusive).

INPUTS

```
Val - The value to map.
Lo  - Lower bound of the range.
Hi  - Higher bound of the range.
```

RESULT

```
v   - A value in the range [Lo,Hi].
```

BUGS

None known.

SEE ALSO

InRand()

NAME

PopMember -- Get the n:th member of a population.

SYNOPSIS

```
void *PopMember(struct Population *,int);

Ind = PopMember(Pop,n);
```

FUNCTION

Returns a pointer to the n:th individual in a population (counted from zero). For a population with say 50 individuals, valid numbers would range from 0 to 49.

INPUTS

Pop - A pointer to an population structure.
n - The number of the individual to retrieve.

RESULT

Ind - A pointer to the n:th individual in the population.

BUGS

None known.

SEE ALSO

CreatePopulation()

NAME

Rnd -- Generate a pseudo-random integer.

SYNOPSIS

```
long int Rnd(long int);

Val = Rnd(Hi);
```

FUNCTION

Generates a pseudo-random integer between zero and one less than Hi. The random number generator is cyclic and repeats after 2147483645 generated numbers.

INPUTS

Hi - Upper bound of random number.

RESULT

Val - An integer in the range [0,Hi[.

BUGS

Hi can not be greater than 2147483646 (0x7ffffffe = 2^31-1). This means that Rnd() only gives 31 random bits, not 32.

SEE ALSO

InRand(), InitRand()

NAME

Testbit -- Test the status of an arbitrary bit in a bitstring.

SYNOPSIS

```
int Testbit(void *,int);

status = Testbit(Ind,At);
```

FUNCTION

Tests the status of a bit in a bitstring. Bits are counted from lower addresses to higher.

INPUTS

```
Ind   - A pointer to the bitstring.
At    - The number of the bit to be tested.
```

RESULT

0 if the bit is clear, non-zero otherwise.

BUGS

None known.

SEE ALSO

Flip()