TALLER DE ALGORITMOS EVOLUTIVOS

Introducción

Durante la última década los métodos de optimización han cobrado más importancia debido, a que con ellos se pueden resolver ciertos problemas de ingeniería que sólo pueden abordarse mediante aproximación en los computadores actuales.

Los paradigmas evolutivos actuales están inspirados en la teoría de la evolución de Darwin e intentan emular en lo posible a la Naturaleza. De esta forma, los cromosomas y los genes se suelen asociar de alguna forma a cadenas de bits o a vectores de números reales. Por otra parte, diversas estrategias de selección imitan la selección natural.

Un buen tutorial sobre computación evolutiva se puede conseguir en
http://geneura.ugr.es/~jmerelo/ie/ags.htm

Anatomía de un algoritmo evolutivo

Los *algoritmos evolutivos* son métodos sistemáticos para la resolución de problemas de búsqueda y optimización que aplican a estos los mismos métodos de la evolución biológica: selección basada en la población, reproducción sexual y mutación.

Los algoritmos evolutivos son métodos de optimización, que tratan de hallar $(x_i,...,x_n)$ tales que $F(x_i,...,x_n)$ sea máximo. En un algoritmo evolutivo, tras parametrizar el problema en una serie de variables, $(x_i,...,x_n)$ se codifican en un cromosoma. Todas los operadores utilizados se aplicarán sobre estos cromosomas, o sobre poblaciones de ellos. Hay que tener en cuenta que un algoritmo evolutivo es independiente del problema, lo cual lo hace un algoritmo *robusto*, por ser útil para cualquier problema, pero a la vez *débil*, pues no está especializado en ninguno.

Las soluciones codificadas en un cromosoma *compiten* para ver cuál constituye la mejor solución (aunque no necesariamente la mejor de todas las soluciones posibles). Las otras soluciones, ejercerán presión selectiva de forma que sólo los mejor adaptados (aquellos que resuelvan mejor el problema) sobrevivan o leguen su material genético a las siguientes generaciones, igual que en la evolución de las especies. La diversidad genética se introduce mediante mutaciones y reproducción sexual.

En la Naturaleza lo único que hay que optimizar es la supervivencia, y eso significa a su vez maximizar diversos factores y minimizar otros. Un algoritmo evolutivo, sin embargo, se usará habitualmente para optimizar sólo una función, no diversas funciones relacionadas entre sí simultáneamente. La optimización que busca diferentes objetivos simultáneamente, denominada multimodal o multiobjetivo, también se suele abordar con un algoritmo evolutivo especializado.

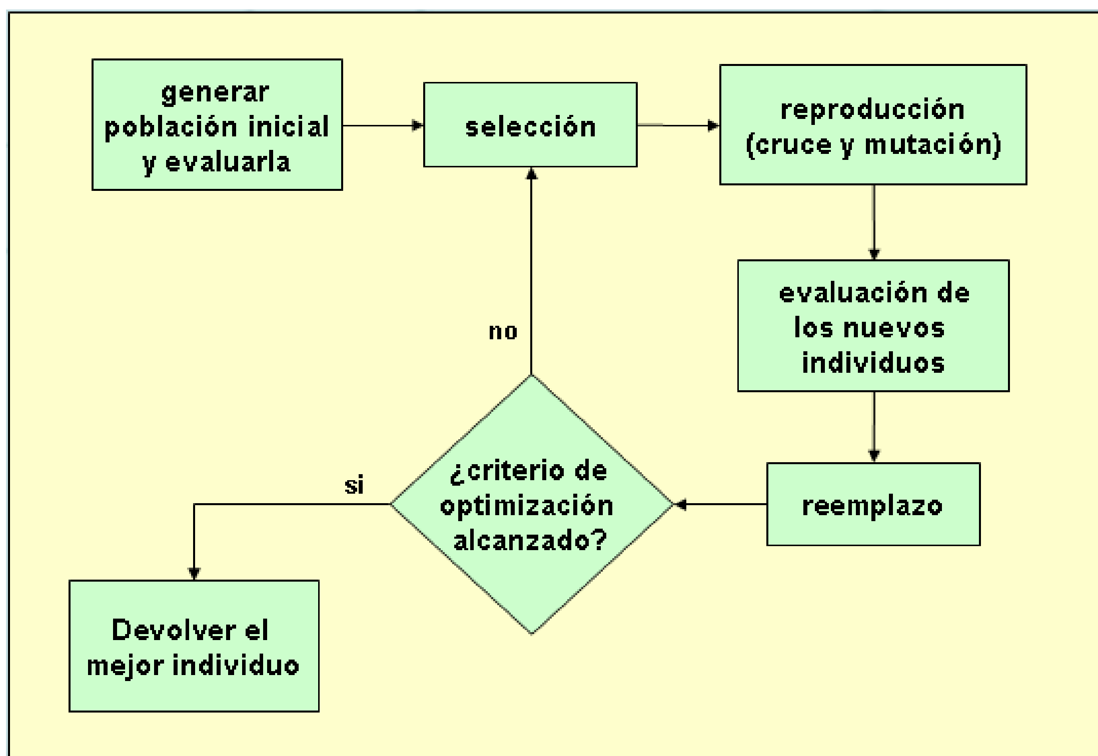
Por lo tanto, un algoritmo evolutivo consiste en lo siguiente: hallar de qué parámetros depende el problema, codificarlos en un cromosoma, y aplicar los métodos de la evolución: selección y reproducción sexual con intercambio de información y alteraciones que generan diversidad. En las siguientes secciones se verán cada uno de los aspectos de un algoritmo evolutivo.

Bucle general de un algoritmo evolutivo

Un algoritmo evolutivo sigue generalmente los siguientes pasos:

1. Generar una POBLACIÓN aleatoria de N individuos
2. Evaluar los individuos de la POBLACIÓN de acuerdo a la función de fitness
3. Repetir durante GENERACIONES iteraciones:
    a. Aplicar el operador de selección para elegir S individuos de la POBLACIÓN
    b. Aplicar los operadores genéticos a esos S individuos para generar la descendencia
    c. Evaluar los nuevos individuos de acuerdo a la función de fitness
    d. Reemplazar los peores individuos en POBLACIÓN por los individuos recién creados

Esquemáticamente, podríamos verlo según la siguiente figura:



Tipos de individuo

Mientras que un algoritmo genético simple utiliza individuos que codifican las variables (que se comentaban anteriormente) como cadenas binarias, otros tipos de algoritmos evolutivos pueden utilizar estructuras complejas como vectores de números reales, redes neuronales, grafos, árboles, etc.

En cada caso se debe elegir una representación que se adapte convenientemente al problema.

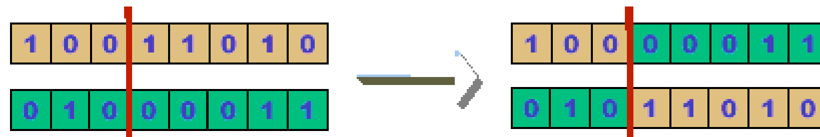
Operadores genéticos

En la Evolución, una ***mutación*** es un suceso bastante poco común (sucede aproximadamente una de cada mil replicaciones). En la mayoría de los casos las mutaciones son letales, pero en promedio, contribuyen a la diversidad genética de la especie. En un algoritmo genético tendrán el mismo papel, y la misma frecuencia (es decir, muy baja).

Una vez establecida la frecuencia de mutación, por ejemplo, uno por mil, se examina cada bit de cada cadena cuando se vaya a crear la nueva criatura a partir de sus padres. Si un número generado aleatoriamente está por debajo de esa probabilidad, se cambiará el bit (es decir, de 0 a 1 o de 1 a 0). Si no, se dejará como está.

No conviene abusar de la mutación. Es cierto que es un mecanismo generador de diversidad, y, por tanto, la solución cuando un algoritmo genético está estancado, pero también es cierto que reduce el algoritmo genético a una búsqueda aleatoria. Siempre es más conveniente usar otros mecanismos de generación de diversidad, como aumentar el tamaño de la población, o garantizar la aleatoriedad de la población inicial.

El ***cruce*** consiste en el intercambio de material genético entre dos cromosomas. Es el principal operador genético, hasta el punto que se puede decir que no es un algoritmo genético si no tiene *cruce*, y, sin embargo, puede serlo perfectamente sin mutación. Para aplicar el cruce se escogen aleatoriamente dos miembros de la población, los dos cromosomas se cortan por N puntos, y el material genético situado entre ellos se intercambia. Lo más habitual es un cruce de un punto o de dos puntos



El cruce es el encargado de mezclar bloques buenos que se encuentren en los diversos progenitores, y que serán los que den a los mismos una buena puntuación. La presión selectiva se encarga de que sólo los buenos bloques se perpetúen, y poco a poco vayan formando una buena solución.

Evaluación de las soluciones

Durante la evaluación, se decodifica el gen, convirtiéndose en una serie de parámetros de un problema, se halla la solución del problema a partir de esos parámetros, y se le da una puntuación a esa solución en función de lo cerca que esté de la mejor solución. A esta puntuación se le llama *fitness*.

El fitness determina siempre los cromosomas que se van a reproducir, y aquellos que se van a eliminar, pero hay varias formas de considerarlo para seleccionar la población de la siguiente generación.

El material para el taller

La documentación sobre algoritmos evolutivos y Perl, el intérprete del lenguaje, el paquete de la librería Algorithm::Evolutionary, y los ejemplos que veremos en este taller se encuentran en http://geneura.ugr.es/~pedro/cursos/baeza/ae/

¿Qué es Algorithm::Evolutionary?

Es una librería de programación escrita en Perl para hacer computación evolutiva, diseñada y desarrollada por Juan Julián Merelo (http://opeal.sourceforge.net).

Los principios del diseño son:

- Es fácil programar cualquier tipo de algoritmo evolutivo.
- Todas las representaciones para los cromosomas y todos los operadores son posibles.
- Un dialecto XML denominado EvoSpec se puede usar como lenguaje para descripción de algoritmos evolutivos y para la representación del estado del algoritmo.

Muchos investigadores opinan que los distintos paradigmas de computación evolutiva sólo difieren en cuanto a la representación y a los operadores de individuo y de población. La librería de computación evolutiva que utilizaremos abstrae las características de cada paradigma, permitiéndonos resolver problemas de optimización de forma rápida y casi sin tener que escribir código.

En Algorithm::Evolutionary tanto los individuos, como los operadores o los algoritmos son objetos, los cuales, se pueden combinar para crear algoritmos evolutivos a medida.

Instalación y configuración de Perl

En este taller utilizaremos Perl como lenguaje de programación.

Perl es un lenguaje de script, lo cual quiere decir que no hace falta generar un fichero binario para poder ejecutar las instrucciones que hemos codificado usando este lenguaje.

Perl tiene características de muchos lenguajes de programación (buscando tomar lo mejor de cada uno), pero al que más se parece en cuanto a sintáxis es al **C**.

El lenguaje se encuentra instalado en cualquier sistema Unix / Linux. Para instalarlo en Windows, podemos descargar la versión de ActiveState (http://www.activestate.com/store/languages/register.plex?id=ActivePerl) o una copia ya descargada de la página http://geneura.ugr.es/~pedro/cursos/baeza/ae/ActivePerl-5.8.7.815-MSWin32-x86-211909.msi

La instalación se realiza de forma automática, asociando los ficheros con extensión .PL al intérprete de Perl.

El aspecto del programa en Perl más sencillo es el siguiente (en Linux):

```
#!/bin/perl
print "hola \n";
```

si estamos trabajando bajo Windows, debemos indicarle en la primera línea el PATH absoluto hasta el programa PERL.EXE

```
#!c:/perl/bin/perl.exe
print "hola \n";
```

Como vemos en la segunda línea (donde se imprime información), hay que terminar cada sentencia de programa con un punto y coma (**;**).
Los comentarios en Perl se introducen poniendo al principio de la línea que será comentada un símbolo #

La ejecución del programa la haremos desde una ventana MSDOS (ejecutar el programa ***cmd***), de la siguiente forma:

```
C:\> perl.exe ejemplo.pl
```

Con Perl se puede programar casi todo (hay librerías -módulos- para casi cualquier cosa que se nos ocurra), pero hay aplicaciones que requieren mucha rapidez en las cuales es mejor utilizar programas compilados (la ejecución a través del intérprete introduce retardos).

Para ampliar conocimientos sobre el lenguaje, podemos visitar la siguiente página en la que se introduce el lenguaje http://geneura.ugr.es/~pedro/cursos/perl/

Instalación de la librería

Perl se instala con las librerías básicas, sin embargo, en muchas ocasiones necesitaremos instalar nuevas librerías. Si disponemos de conexión a internet, usaremos el asistente del lenguaje, de forma que baje los paquetes necesarios y las dependencias, y realice la instalación de forma automática.

Para ello, ejecutamos la siguiente orden:

```
C:\> perl.exe -MCPAN -e shell
```

El comando "***help***" nos muestra todas las opciones para manejar módulos.
Vemos la sintaxis de la opción de instalar: "***install PAQUETE***".

```
cmd - perl.exe -MCPAN -e shell
C:\Documents and Settings\pedro>perl.exe -MCPAN -e shell

cpan shell -- CPAN exploration and modules installation (v1.7601)
ReadLine support enabled

cpan> help

Display Information
 command   argument          description
 a,b,d,m   WORD or /REGEXP/  about authors, bundles, distributions, modules
 i         WORD or /REGEXP/  about anything of above
 r         NONE              reinstall recommendations
 ls        AUTHOR            about files in the author's directory

Download, Test, Make, Install...
 get                         download
 make                        make (implies get)
 test      MODULES,          make test (implies make)
 install   DISTS, BUNDLES    make install (implies test)
 clean                       make clean
 look                        open subshell in these dists' directories
 readme                      display these dists' README files

Other
 h,?           display this menu       ! perl-code   eval a perl command
 o conf [opt]  set and query options   q             quit the cpan shell
 reload cpan   load CPAN.pm again      reload index  load newer indices
 autobundle    Snapshot                force cmd     unconditionally do cmd
cpan>
cpan>
cpan>
cpan>
cpan>
cpan> force install Algorithm::Evolutionary
```

```
cmd - perl.exe -MCPAN -e shell

Display Information
 command   argument          description
 a,b,d,m   WORD or /REGEXP/  about authors, bundles, distributions, modules
 i         WORD or /REGEXP/  about anything of above
 r         NONE              reinstall recommendations
 ls        AUTHOR            about files in the author's directory

Download, Test, Make, Install...
 get                         download
 make                        make (implies get)
 test      MODULES,          make test (implies make)
 install   DISTS, BUNDLES    make install (implies test)
 clean                       make clean
 look                        open subshell in these dists' directories
 readme                      display these dists' README files

Other
 h,?           display this menu       ! perl-code   eval a perl command
 o conf [opt]  set and query options   q             quit the cpan shell
 reload cpan   load CPAN.pm again      reload index  load newer indices
 autobundle    Snapshot                force cmd     unconditionally do cmd
cpan>
cpan>
cpan>
cpan>
cpan>
cpan> force install Algorithm::Evolutionary
CPAN: Storable loaded ok
CPAN: LWP::UserAgent loaded ok
Fetching with LWP:
  ftp://ftp.rediris.es/mirror/CPAN/authors/01mailrc.txt.gz
Going to read \.cpan\sources\authors\01mailrc.txt.gz
CPAN: Compress::Zlib loaded ok
Fetching with LWP:
  ftp://ftp.rediris.es/mirror/CPAN/modules/02packages.details.txt.gz
```

Si es necesario, bajará los paquetes necesarios para que el que estamos instalando funcione correctamente.

```
21.
main::createAndTest() called too early to check prototype at t\individuals.t lin
e 28.
Name "main::op" used only once: possible typo at t\individuals.t line 29.
t\individuals.........ok
t\NoChangeTerm........ok
t\ops.................main::createAndTest() called too early to check prototype
at t\ops.t line 31.
t\ops.................ok
t\run.................Can't locate XML/LibXML.pm in @INC (@INC contains: ../.. .
./../.. . .. C:\.cpan\build\Algorithm-Evolutionary-0.53\blib\lib C:\.cpan\build\
Algorithm-Evolutionary-0.53\blib\arch C:/perl/lib C:/perl/site/lib) at t\run.t l
ine 6.
BEGIN failed--compilation aborted at t\run.t line 6.
t\run.................dubious
        Test returned status 2 (wstat 512, 0x200)
t\validate............Can't locate XML/LibXML.pm in @INC (@INC contains: C:\.cpa
n\build\Algorithm-Evolutionary-0.53\blib\lib C:\.cpan\build\Algorithm-Evolutiona
ry-0.53\blib\arch C:/perl/lib C:/perl/site/lib .) at t\validate.t line 2.
BEGIN failed--compilation aborted at t\validate.t line 2.
t\validate............dubious
        Test returned status 2 (wstat 512, 0x200)
Failed Test  Stat Wstat Total Fail  Failed  List of Failed
-------------------------------------------------------------------------------
t\run.t         2   512    ??   ??       %  ??
t\validate.t    2   512    ??   ??       %  ??
1 subtest skipped.
Failed 2/9 test scripts, 77.78% okay. 0/102 subtests failed, 100.00% okay.
dmake.exe:  Error code 2, while making 'test_dynamic'
  dmake test -- NOT OK
Running make install
Appending installation info to C:\Perl\lib/perllocal.pod
  dmake install  -- OK

cpan>
cpan>
cpan>
```

Tras la instalación del paquete, todo está listo para empezar a programar usando dicha librería.

¿Cómo usar la librería?

Para construir un Algoritmo Evolutivo, tendremos que programar nuestra función de fitness (función de evaluación) en Perl y elegir los tipos de datos (individuos), operadores y algoritmo evolutivo a aplicar.

En los siguientes apartados describiremos cómo usar Algorithm::Evolutionary a partir de sencillos ejemplos. En el caso de que los operadores, algoritmos y/o individuos existentes no se adecuen a la necesidad de nuestro problema, siempre podemos añadir nuevos elementos a la librería, contribuyendo a su mejora.

Definición y uso de individuos binarios

Un algoritmo evolutivo básico utiliza individuos binarios (cadenas binarias). Para usar este tipo de individuos en un programa, debemos incluir el módulo BitString. A continuación ya podemos crear un objeto individuo (indicando la longitud de la cadena binaria).

```
use Algorithm::Evolutionary::Individual::BitString;

my $indiv = Algorithm::Evolutionary::
                  Individual::BitString->new (10);

print  $indiv->Atom( 7 ) ;
print  $indiv->asString() ;
```

Vemos que la definición de datos y la forma de acceder al individuo y a sus componentes (bits) es muy sencilla.

Definición y uso de un operador de mutación

Cualquier algoritmo evolutivo necesita uno o varios operadores de variación para crear nuevos individuos a partir de los que hay en la población. Para usar un operador de tipo mutación en un programa, debemos incluir el módulo Mutation. A continuación ya podemos crear un operador (indicando el porcentaje de mutación).

```
use Algorithm::Evolutionary::Individual::BitString;
use Algorithm::Evolutionary::Op::Mutation;

my $mutar = Algorithm::Evolutionary::Op::Mutation->new( 0.1 );

my $indiv = Algorithm::Evolutionary::
                 Individual::BitString->new( 8 );

my $mutado = $mutar->apply( $indiv );

print  $indiv->asString() ;
print  $mutado->asString() ;
```

Vemos que la aplicación del operador de mutación resulta muy sencilla.

Definición y uso de un operador de cruce

Para usar un operador de cruce, debemos incluir el módulo Crossover. Después creamos un operador indicando el número de puntos de cruce. El cruce genera dos descendientes.

```
use Algorithm::Evolutionary::Individual::BitString;
use Algorithm::Evolutionary::Op::Crossover;

my $cruce = Algorithm::Evolutionary::Op::Crossover->new( 2 );

my $i1 = Algorithm::Evolutionary::
                 Individual::BitString->new (8);
my $i2 = Algorithm::Evolutionary::
                 Individual::BitString->new (8);

my $hijo1 = $cruce->apply( $i1, $i2 );
my $hijo2 = $cruce->apply( $i2, $i1);

print  $i1->asString() ;
print  $i2->asString() ;

print  $hijo1->asString() ;
print  $hijo2->asString() ;
```

Ejemplo de un algoritmo genético con individuos binarios: problema de la función marea

Queremos resolver un problema de optimización usando algoritmos evolutivos. En este caso tenemos la función marea cuya expresión matemática es la siguiente:

$$f(x,y) = \frac{sin(\sqrt{x^2+y^2})}{\sqrt{x^2+y^2}}$$

y cuya gráfica se muestra en la Figura 1. El problema al que nos enfretamos es encontrar las coordenadas *(x,y)*, que maximicen la función *f(x,y)*.

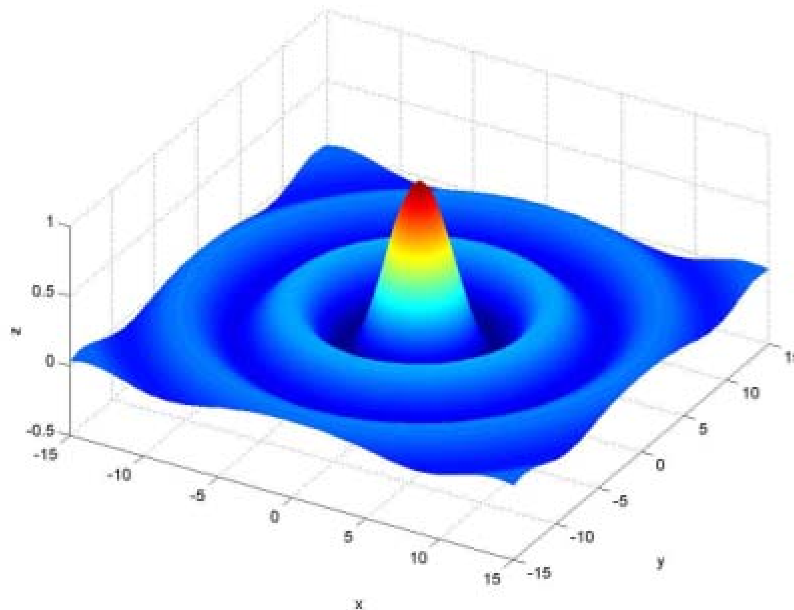


Figura 1:Gráfica de la función "marea".

Si observamos la gráfica de la función (ver Figura 1) la forma de la función va haciendo ondulaciones conforme se acerca a la coordenado *(0,0)*, alcanzando el máximo en dicha coordenada, de forma que *f(0,0)=1*

Elección del tipo de individuo

Una vez conocido el problema a resolver, deberemos decidir el tipo de datos que usaremos para codificar la solución, en este ejemplo usaremos un vector de 128 bits, donde los primeros 64bits, serán el valor binario de la *x* y los restantes 64bits, serán la codificación en binario del valor *y*. Por tanto, de entre los tipos de individuo escogemos BitString que sirve pare representar un vector de valores binarios.

Creación de la función de evaluación o fitness

Ya tenemos el tipo de individuo a usar y el problema a resolver, por lo que ahora debemos construir nuestra función de fitness (función de adecuamiento) que mide lo buena que es el individuo, en nuestro ejemplo calculará *f(x,y)* a partir del individuo de 128bits, cuyos primeros 64bits son para la *x* y lo restantes 64bits para la *y*. Vamos a

construir nuestra función de fitness en Perl, para nuestro problema y nuestra representación:

```
my $funcionMarea = sub {
  #Cogemos el individuo a evaluar
  my $chrom = shift;

  my $str = $chrom->Chrom();
  my $fitness = 1;

  #extraemos los dos números reales de la cadena binaria
  my $l2=length($str)/2;
  my $x=eval("0b".substr ($str, 0, $l2));
  my $y=eval("0b".substr ($str,    $l2));

  #Los normalizamos y los pasamos al rango [0,1]
  my $max=(2 ** ($l2) )-1;
  $x=$x/$max;
  $y=$y/$max;

  my $sqrt=sqrt( ($x*$x) + ($y*$y) );
  $fitness= sin( $sqrt ) / $sqrt  if ($sqrt !=0 );

  return $fitness;
};
```

Con las primeras instrucciones tomamos el individuo a evaluar (lo pasan a la función como parámetro), y extraemos los dos valores numéricos que codifica el individuo.

Creación de la población inicial

Una vez elegidos el tipo de individuo y construida la función de fitness, construimos la población inicial de individuos que evolucionarán:

```
my @pop;
for ( 0..$popSize ) {
  my $indi = Algorithm::Evolutionary::
         Individual::BitString->new( $numBits ) ;
  push( @pop, $indi );
}
```

Definición de los operadores genéticos

Para llevar a cabo la evolución, debemos definir unos operadores de variación (operadores genéticos) que reciban ciertos individuos y generen nuevos individuos (mediante mutaciones y cruces).

```
my $mutacion = Algorithm::Evolutionary::Op::Mutation->new(0.1);
my $cruce = Algorithm::Evolutionary::Op::Crossover->new(2);
```

Elección del algoritmo

Con la función de evaluación definida, y los operadores de variación creados vamos a definir la generación básica del algoritmo evolutivo. Dicho objeto-generación necesita la función de evaluación, una tasa de selección, y los operadores de variación

(operadores genéticos) para la fase de reproducción. Como ya describimos, la generación consistirá en:

- Selección de los mejores individuos de la población
- Aplicación de los operadores de variación para crear la descendencia
- Evaluación de los nuevos individuos
- Reemplazo de los peores individuos de la población por los nuevos

```
my $generation = Algorithm::Evolutionary::
         Op::Easy->new( $funcionMarea , 0.2 , [$m, $c] ) ;
```

El bucle general del algoritmo evolutivo se limitará a hacer un número especificado de generaciones (condición de terminación), aplicando en cada ciclo del bucle una generación a la población.

```
do {
  $generation->apply( \@pop );
  $numGens -- ;
} while( $numGens > 0 );
```

Terminando

Una vez terminado el algoritmo, podemos mostrar el mejor individuo encontrado:

```
print "La mejor solución encontrada es: ";
print $pop[0]->asString() ;
```

Para terminar el programa, podemos mostrar los valores numéricos que codifica ese individuo:

```
my $l2=length($str)/2;
my $x=eval("0b".substr ($str, 0, $l2));
my $y=eval("0b".substr ($str,    $l2));

my $max=(2 ** ($l2) )-1;
$x=$x/$max;
$y=$y/$max;
```

Para completar el programa Perl, viendo qué módulos hemos necesitado, al principio del código debemos incluir los siguientes módulos:

```
use Algorithm::Evolutionary::Individual::BitString;
use Algorithm::Evolutionary::Op::Easy;
use Algorithm::Evolutionary::Op::Mutation;
use Algorithm::Evolutionary::Op::Crossover;
```

Con esto tenemos listo el algoritmo evolutivo que encuentra el máximo de una función matemática. Sólo hemos tenido que programar la función de fitness y elegir el tipo de datos, ya que los operadores y el algoritmo a usar siempre acaban siendo los mismos.

Ejemplo de un algoritmo evolutivo con individuos reales: problema de la función marea

A continuación veremos el mismo problema anterior, pero en este caso usaremos vectores de números reales para codificar las soluciones.

De esta forma, de entre los tipos de individuo escogemos Vector, que sirve para representar un vector de valores reales.

Ya que cambiamos la representación, debemos cambiar también el tipo de operadores de variación (adaptados al tipo de representación).
En este caso, usaremos una mutación de tipo Gausiano, y un cruce multipunto adaptado a vectores de números reales en lugar de cadenas binarias.

El resto del algoritmo evolutivo queda igual que el descrito anteriormente, salvo que no hay que decodificar las cadenas binarias en números reales, sino acceder directamente a cada componente del individuo.

```
#!/usr/bin/perl
use warnings;
use strict;

#Incluimos los elementos de la librería necesarios
use Algorithm::Evolutionary::Individual::Vector;
use Algorithm::Evolutionary::Op::Easy;
use Algorithm::Evolutionary::Op::GaussianMutation;
use Algorithm::Evolutionary::Op::VectorCrossover;

#----------------------------------------------------------#
#leemos los parámetros de la línea de comandos
my $popSize = shift || 100;
my $numGens = shift || 100 ;

#----------------------------------------------------------#
#Definimos la función de fitnes, que es la función Marea
my $funcionMarea = sub {
  my $indi = shift;
  my ( $x, $y ) = @{$indi->{_array}};
  my $sqrt = sqrt( $x*$x+$y*$y);

  if( !$sqrt ){ return 1; }
  return sin( $sqrt )/$sqrt;
};

#----------------------------------------------------------#
#Creamos la población inicial con $popSize individuos
my @pop;
for ( 0..$popSize ) {
  my $indi = Algorithm::Evolutionary::Individual::Vector->new( 2 );
  push( @pop, $indi );
}

#----------------------------------------------------------#
#Definimos los operadores de variación
my $m = Algorithm::Evolutionary::Op::GaussianMutation->new( 0, 0.1 );
my $c = Algorithm::Evolutionary::Op::VectorCrossover->new(2);

#----------------------------------------------------------#
#  Usamos estos operadores para definir una generación del algoritmo. Lo cual
#  no es realmente necesario ya que este algoritmo define ambos operadores por
#  defecto. Los parámetros son la función de fitness, la tasa de selección y
#  los operadores de variación.
my $generation = Algorithm::Evolutionary::Op::Easy
                                ->new( $funcionMarea , 0.2 , [$m, $c] ) ;
```

```
#----------------------------------------------------------#
#Evaluamos la población inicial
for ( @pop ) {
  if ( !defined $_->Fitness() ) {
    my $fitness = $funcionMarea->($_);
    $_->Fitness( $fitness );
  }
}

#bucle del algoritmo evolutivo
my $contador=0;
do {
  $generation->apply( \@pop );

  print "$contador : ", $pop[0]->asString(), "\n" ;

  $contador++;
} while( $contador < $numGens );

#----------------------------------------------------------#
#tomamos la mejor solución encontrada y la mostramos
my ( $x, $y ) = @{$pop[0]->{_array}};

print "El mejor es:\n\t ";
print $pop[0]->asString() ;
print "\n\t x=$x \n\t y=$y \n\t Fitness: ";
print $pop[0]->Fitness() ;
```