P1.2.- FUNDAMENTOS PRÁCTICOS

Librería GAlib de algoritmos genéticos: elementos básicos

Construcción de un algoritmo genético en GAlib para el problema de las n-reinas

Uso de la librería GAlib en Code::Blocks

Implementación del algoritmo genético para el problema de las n-reinas

- ¿Cómo se comporta internamente el algoritmo genético?

Terminando de construir el sistema basado en el algoritmo genético para resolver el problema de las n-reinas

- Ajuste de parámetros: Estudio de casos – valores de parámetros
- Manual de asignación

Búsqueda mediante Algoritmos Genéticos

Recordemos, un esquema de un Algoritmo Genético (AG):

función Algoritmo Genético
  t = 0
  inicializar P(t)
  evaluar P(t)
  mientras (no condición de parada) **hacer**
    t = t + 1
    seleccionar P(t) desde P(t-1)
    cruzar P(t)
    mutación P(t)
    evaluar P(t)

Construiremos AGs mediante la librería GAlib:

- Es una librería de objetos C++,
- e incluye distintas representaciones y operadores genéticos.
- Material complementario:

https://www.um.es/web/innovacion/plataformas/ocw/listado-de-cursos/sistemas-inteligentes

Librería GAlib de algoritmos genéticos: elementos básicos

Los elementos básicos para resolver un problema usando la librería GAlib, y siguiendo el esquema de un AG, son:

1. Tipo de AG
   - En GAlib se usa la clase base **`GAGeneticAlgorithm`** (pag. 22)
   - Usaremos una de las clases derivadas de **`GAGeneticAlgorithm`** (**"clase-tipoGA"**) (pag. 27, 29, 31, 32)
2. Representación de una solución del problema (individuo o cromosoma) usando un objeto
   - En GAlib se usa la clase base **`GAGenome`** (pag. 36)
   - Usaremos una de sus clases derivadas (**"clase-GAGenome"**) (pag. 39, 41, 42, 44, 45, 47, 48, 50, 52, 54, 57, 58)
3. Inicializar una solución con esa estructura de datos
   - **"clase-GAGenome":: "método-de-inicialización"**
4. Evaluar → lo define el usuario en función del problema
5. Selección
   - En GAlib se usa la clase base **`GASelectionScheme`** (pag. 77)
   - Usaremos una de las clases derivadas de **`GASelectionScheme`**(**"clase-selección"**)
6. Cruce →
   - **"clase-GAGenome":: "método-de-cruce"**
7. Mutación →
   - **"clase-GAGenome":: "método-de-mutación"**
8. Parada →
   - **"clase-tipoGA":: "método-de-terminación"**
9. Evolucionar la población → método **`evolve`** de **`GAGeneticAlgorithm`**

Librería GAlib de algoritmos genéticos: elementos básicos

Básicamente, los elementos dependen de dos clases: GAGenome, GAGeneticAlgorithm

Individuo

- En GAlib se utiliza la clase **`GAGenome`** → se usa una de sus clases derivadas(**`"clase-GAGenome"`**)

  Inicialización
  - **`"clase-GAGenome":: "método-de-inicialización"`**

  `Operadores`
  - Cruce → **`"clase-GAGenome":: "método-de-cruce"`**
  - Mutación → **`"clase-GAGenome":: "método-de-mutación"`**

Algoritmo Genético

- En GAlib se usa la clase **`GAGeneticAlgorithm`**
- Usaremos una de las clases derivadas de **`GAGeneticAlgorithm ("clase-tipoGA")`**

  Selección
  - En GAlib se utiliza la clase base **`GASelectionScheme`** → se usa una de sus clases derivadas (**`"clase-selección"`**)

  Parada
  - **`"clase-tipoGA":: "método-de-terminación"`**

  Evolucionar la población
  - método **`evolve`**

Librería GAlib de algoritmos genéticos: elementos básicos

INDIVIDUO

Una vez seleccionada la “clase-GAGenome” con la que se va a resolver el problema, se crea un individuo usando el constructor de la clase y especificamos la inicialización y los operadores de cruce y mutación mediante una de las siguientes tres opciones:

1. Dejando los operadores por defecto que asigna la librería a “clase-GAGenome”. En este caso el usuario no debe hacer nada.
2. Usando otros operadores definidos en la librería para esa clase. En este caso sobre el individuo creado invocamos los métodos siguientes:
   - initializer(“clase-GAGenome”::“método-de-inicialización”);
   - crossover(“clase-GAGenome”::“método-de-cruce”);
   - mutator(“clase-GAGenome”::“método-de-mutación”);

Librería GAlib de algoritmos genéticos: elementos básicos

INDIVIDUO

3. Implementando otros operadores. En este caso sobre el individuo invocamos los mismos métodos especificando como operadores métodos implementados por el usuario:

   - initializer(metodo_inicio);
   - crossover(metodo_cruce);
   - mutator(metodo_mutacion);

   Las signaturas de estos métodos implementados deben ser:

   void metodo_inicio(GAGenome &);
   El parámetro es el genoma a inicializar.

   int metodo_mutacion(GAGenome &, float);
   Los parámetros son el genoma y la probabilidad de mutación.
   Devuelve el número de mutaciones.

   int funcion_cruce(const GAGenome &, const GAGenome &, GAGenome*, GAGenome*);
   Los parámetros son los dos genomas que se van a cruzar (padres) y los dos genomas resultantes (hijos).
   Devuelve el número de genomas (individuos) generados.

Librería GAlib de algoritmos genéticos: elementos básicos

Operadores disponibles en la librería

Operadores primarios: operadores de inicio

- **UniformInitializer**: Inicializa el genoma siguiendo una distribución uniforme
- **OrderedInitializer**: Tiene en cuenta que el genoma es una lista ordenada

Operadores primarios: operadores de mutación

- **FlipMutator**: cambia el valor de un gen a otro de sus posibles valores
- **SwapMutator**: intercambia dos genes del individuo
- **GARealGaussianMutator**: añade un valor aleatorio según una distribución gaussiana

Operadores primarios: operadores de cruce

- **UniformCrossover**: compara los genes de los padres y los intercambia con cierta probabilidad si son distintos
- **EvenOddCrossover**: un hijo hereda los genes pares de los padres y el otro los impares
- **OnePointCrossover**: cruce por un punto.
- **TwoPointCrossover**: cruce por dos puntos
- **PartialMatchCrossover**: (genomas - listas ordenadas) dados dos puntos, se intercambia la parte intermedia de los dos padres y los valores repetidos se sustituyen por los que faltan siguiendo el mismo orden
- **OrderCrossover**: (genomas - listas ordenadas) una vez seleccionado el punto de cruce, la segunda parte del genoma se completa según el orden del otro genoma

Librería GAlib de algoritmos genéticos: elementos básicos

INDIVIDUO - *Métodos*

Para acceder a los genes de un genoma podemos usar los métodos siguientes:

- `T & gene(unsigned int x=0)`

  Devuelve el valor del gen en la posición x

- `int length() const`

  Devuelve la longitud del genoma

Librería GAlib de algoritmos genéticos: elementos básicos

INDIVIDUO - *Ejemplo*

Clase **`GA1DArrayAlleleGenome`** derivada de **`GA1DArrayGenome`**

(y ésta derivada de **`GAGenome`**)

Constructor de **`GA1DArrayAlleleGenome`**:

```
GA1DArrayAlleleGenome<T>(unsigned int length, const GAAlleleSet<T> &
alleleset, GAGenome::Evaluator objective=NULL, void * userData=NULL)
```

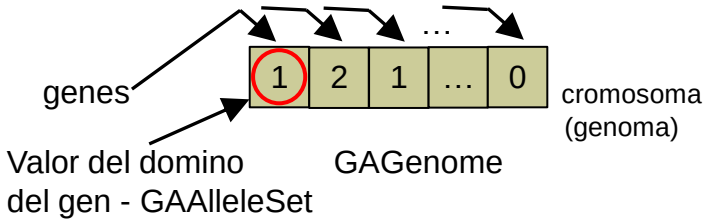
- Este constructor define un individuo o cromosoma (genoma) formado por un vector de objetos de tipo <T>, donde el valor de cada elemento (gen) está definido por un conjunto de alelos (dominio o conjunto de posibles valores para los genes).
- Como parámetros se indican:
  - La longitud del cromosoma (número de genes): **`length`**
  - El conjunto de alelos que definen el posible valor de cada gen (conjunto dominio): **`const GAAlleleSet<T> & alleleset.`** El dominio de los genes (conjunto de alelos) se define como un objeto de la clase **`GAAlleleSet<T>`** donde **`T`** indica el tipo de datos que representa los alelos.
  - La función objetivo que define la función de idoneidad o fitness de cada cromosoma, y que tiene que ser definida por el usuario: **`GAGenome::Evaluator objective=NULL`**. Su signatura es **`float Objective(GAGenome &);`**

Librería GAlib de algoritmos genéticos: elementos básicos

- Una estructura de datos con información del problema, si es necesaria: **`void * userData=NULL.`** Esa estructura puede ser recuperada desde los distintos operadores de la forma que se indica a continuación:

```
struct datos * Data=(struct datos *) genome.userData();
```

Gráficamente, un cromosoma lo podemos representar como:



INDIVIDUO - *Ejemplo*

Inicialización de la población

- Indicamos el método para inicializar los individuos como sigue:

```
GA1DArrayAlleleGenome<T>::metodo-de-inicializacion
```

Por ejemplo, asignamos al individuo la inicialización uniforme:

```
genome.initializer(GA1DArrayAlleleGenome<T>::UniformInitializer);
```

Operadores

- Indicamos los operadores a usar como sigue:

```
GA1DArrayAlleleGenome<T>::metodo-de-cruce
GA1DArrayAlleleGenome<T>::metodo-de-mutacion
```

Por ejemplo, asignamos cruce por un punto y mutación aleatoria:

```
genome.crossover(GA1DArrayAlleleGenome<T>::OnePointCrossover);
genome.mutator(GA1DArrayAlleleGenome<T>::FlipMutator);
```

Librería GAlib de algoritmos genéticos: elementos básicos

ALGORITMO GENÉTICO

Creamos un AG de la clase derivada GASimpleGA usando el constructor:

```
GASimpleGA(const GAGenome &)
```

Este constructor crea un AG clonando el genoma que se le pasa como parámetro. Cada generación produce una población nueva seleccionando individuos de la generación anterior para que se reproduzcan.

Para indicar si queremos que el mejor individuo de la población anterior pase a la actual si no hay en la actual ningún individuo mejor usamos el método:

```
GABoolean elitist(GABoolean flag)
```

`flag` puede tomar los valores **`gaTrue`** o **`gaFalse`**. Por defecto está a true y por lo tanto el AG se define con elitismo.

ALGORITMO GENÉTICO

Especificamos el método de selección a usar en el AG invocando sobre el mismo el método selector:

```
selector(const GASelectionSqueme & select);
```

donde select determina el esquema de selección y se define de la siguiente forma: `"clase-seleccion" select;`

Especificamos el operador de terminación de una forma equivalente a los operadores de cruce y mutación, pero invocando sobre el objeto algoritmo genético el método terminator:

```
terminator(GAGeneticAlgorithm::"metodo-de-terminacion");
```

si el método de terminación es de la librería ó

```
terminator(metodo_terminacion);
```

si el método lo implementa el usuario. En este caso, la signatura del método es:

```
GABoolean metodo_terminacion(GAGeneticAlgorithm &);
```

El parámetro de entrada es el AG y devuelve true cuando se cumple la condición de parada.

Librería GAlib de algoritmos genéticos: elementos básicos

Operadores disponibles en la librería

Operadores primarios: métodos de selección

- **GARouletteWheelSelector:** Se asigna una probabilidad de selección proporcional al valor del fitness del cromosoma. (Modelo clásico)
- **GATournamentSelector:** Escoge al individuo de mejor fitness de entre Nts individuos seleccionados aleatoriamente con reemplazamiento (Nts=2,3,…)
- **GAUniformSelector:** Todos los individuos tienen la misma probabilidad de ser selccionados.
- **GARankSelector:** La población se ordena en función de su fitness y se asocia una probabilidad de selección a cada individuo que depende de su orden

Operadores primarios: operadores de terminación

- **TerminateUponGeneration:** Condición de parada en base al número de generaciones.
- **TerminateUponConvergence:** Condición de parada en base a la no mejora de la solución durante un cierto número de generaciones.
- **TerminateUponPopConvergence:** Condición de parada en base a la cercanía en media de la población al mejor individuo.

Librería GAlib de algoritmos genéticos: elementos básicos

ALGORITMO GENÉTICO - *métodos*

Los principales métodos que podemos usar sobre un objeto algoritmo genético son los siguientes:

- **`evolve(unsigned int seed=0)`**

  Inicia la evolución del algoritmo genético hasta que se cumpla el criterio de parada.

- **`populationSize(unsigned int)`**

  Define el tamaño de la población.

- **`nGenerations(unsigned int)`**

  Define el número de generaciones.

- **`pCrossover(float)`**

  Define la probabilidad de cruce.

- **`pMutation(float)`**

  Define la probabilidad de mutación

Construcción de un AG en GAlib para el problema de las n-reinas

Resolución del problema de las *n* reinas mediante AG usando la librería GAlib

1. Representar una solución del problema (individuo o cromosoma) usando un objeto:
   - Nosotros usaremos la clase **`GA1DArrayAlleleGenome<T>`** **(pag. 41)**
2. Seleccionamos el método de inicialización.
   - **`GA1DArrayAlleleGenome<T>::UniformInitializer`**
3. Evaluar → lo define el usuario en función del problema
4. **`GAGeneticAlgorithm`** → **`GASimpleGA`** **(pag. 31)**
5. Selección

   `GASelectionScheme` → **`GARouletteWheelSelector`** **(pag. 77)**] **Por defecto**
6. Cruce → método **`crossover : GA1DArrayAlleleGenome<T>::OnePointCrossover`**
7. Mutación → método **`mutator : GA1DArrayAlleleGenome<T>::FlipMutator`**
8. Parada → método **`terminator: GASimpleGA::TerminateUponGeneration`**
9. Evolucionar esa población para obtener la mejor solución.

   método **`evolve`** de la clase **`GAGeneticAlgorithm`**

Construcción de un AG en GAlib para el problema de las n-reinas

INDIVIDUO O CROMOSOMA:

Para representar a un individuo en el problema de las *n* reinas vamos a utilizar un array de *n* posiciones:

- Cada posición corresponde a una columna del tablero
- El contenido de la posición indica la fila en la que se encuentra situada la reina que corresponde a dicha columna
- En esta representación cada reina se mueve solamente en su columna

El conjunto de alelos (dominio de cada gen) está formado por el conjunto de posibles filas donde situar una reina (de 0 a *n*-1) con una longitud del cromosoma de *n* (para representar las *n* columnas)

- Utilizamos el método **`add(int i)`** de la clase **`GAAlleleSet<int>`**:

  Para las 8 reinas

```
GAAlleleSet<int> alelos;
for(int i=0;i<8;i++) alelos.add(i);
```

Construcción de un AG en GAlib para el problema de las n-reinas

Por tanto,

- Declaramos el tamaño del problema, definiendo la variable **`nreinas`**:

```
int nreinas=8;
```

- Definimos el dominio para los genes, creando un objeto **`alelos`** de la clase **`GAAlleleSet<T>`** que contiene el conjunto enumerado de alelos:

```
GAAlleleSet<int> alelos;
for(int i=0;i<nreinas;i++) alelos.add(i);
```

- Construimos un genoma (individuo), usando el constructor:

```
GA1DArrayAlleleGenome<int>
   genome(nreinas,alelos,Objective,NULL);
```

- La función idoneidad (o fitness), **`Objective`**, tendrá que ser definida por el usuario, y la explicaremos después

Construcción de un AG en GAlib para el problema de las n-reinas

INICIALIZACIÓN DE LA POBLACIÓN:

- La inicialización se lleva a cabo siguiendo una distribución uniforme: **`GA1DArrayAlleleGenome<int>::UniformInitializer`**

OPERADOR DE MUTACIÓN:

- La mutación se lleva a cabo cambiando el valor de un gen por otro valor del conjunto de alelos: **`GA1DArrayAlleleGenome<int>::FlipMutator`**

OPERADOR DE CRUCE:

- Usamos cruce por un punto:

`GA1DArrayAlleleGenome<int>::OnePointCrossover`

SELECCIÓN:

- Usamos la selección por ruleta: clase **`GARouletteWheelSelector`**

OPERADOR DE TERMINACIÓN:

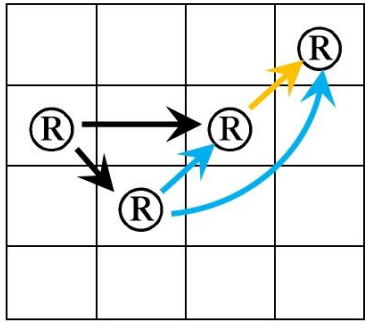
- La terminación se define en base al número de generaciones:

`GAGeneticAlgorithm::TerminateUponGeneration`

Construcción de un AG en GAlib para el problema de las n-reinas

FUNCIÓN FITNESS O IDONEIDAD

- Definimos la función fitness como el número de jaques que existen en el individuo
  - Un cromosoma será mejor cuanto menor sea el valor de esta función
  - El AG debe proporcionar la solución con menor valor (una solución con función fitness igual a 0 es una solución óptima)
- Los jaques se calcularán como se ilustra en la figura, contando una única vez los jaques mutuos entre dos reinas.
- La función fitness recibe un genoma como entrada y devuelve un valor para dicho genoma.



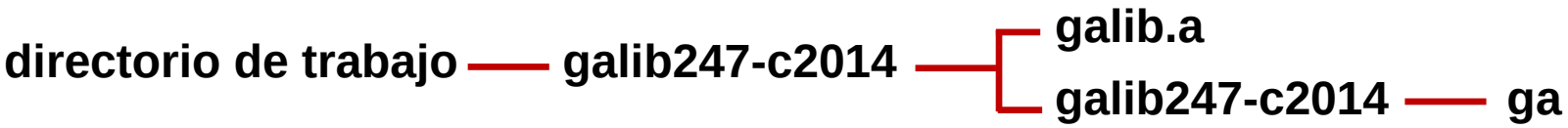
5 jaques

```
float Objective(GAGenome & g) {
  GA1DArrayAlleleGenome<int> & genome = (GA1DArrayAlleleGenome<int> &)g;
  float jaques=0;
   for(int i=0; i<genome.length(); i++)
         for(int j=i+1;j<genome.length();j++)
               if ((genome.gene(i)==genome.gene(j))||(abs(j-i)==
                    abs(genome.gene(j)-genome.gene(i)))) jaques++;
  return jaques;
}
```

Uso de la librería GAlib en Code::Blocks

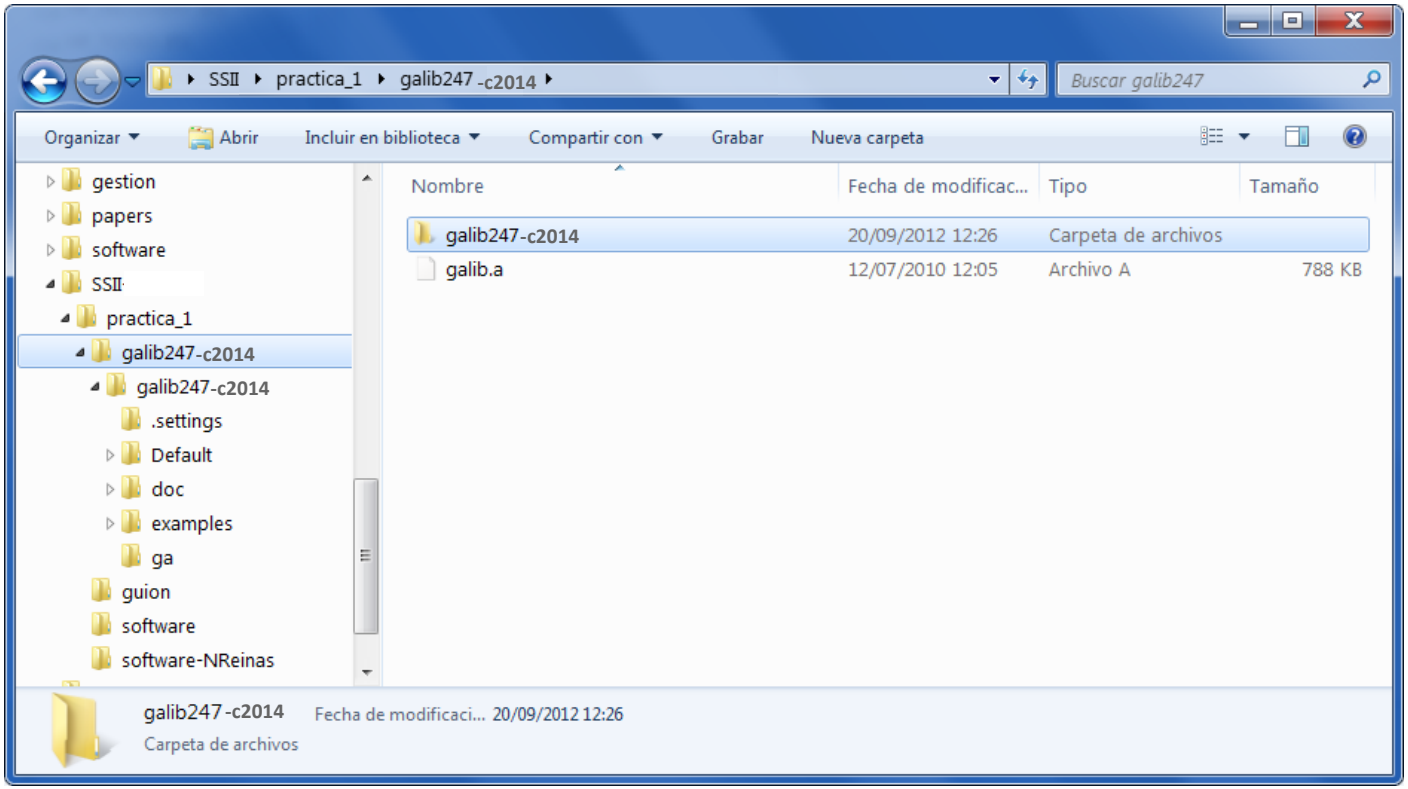
Para poder crear un proyecto en Code::Blocks que utilice la librería GAlib hay que seguir los siguientes pasos (ver también las transparencias 41 y 42)**:**

- Copiar la estructura de directorios, con su contenido, proporcionada en **"galib247-c2014.rar"**
  - La estructura consiste el subdirectorio **`"galib247-c2014"`** en el directorio de trabajo
  - Este subdirectorio contendrá la librería estática **`galib.a`** y el subdirectorio **`"galib247-c2014"`**
  - Este nuevo subdirectorio contendrá el subdirectorio **`"ga"`** con los .c, .h, .o



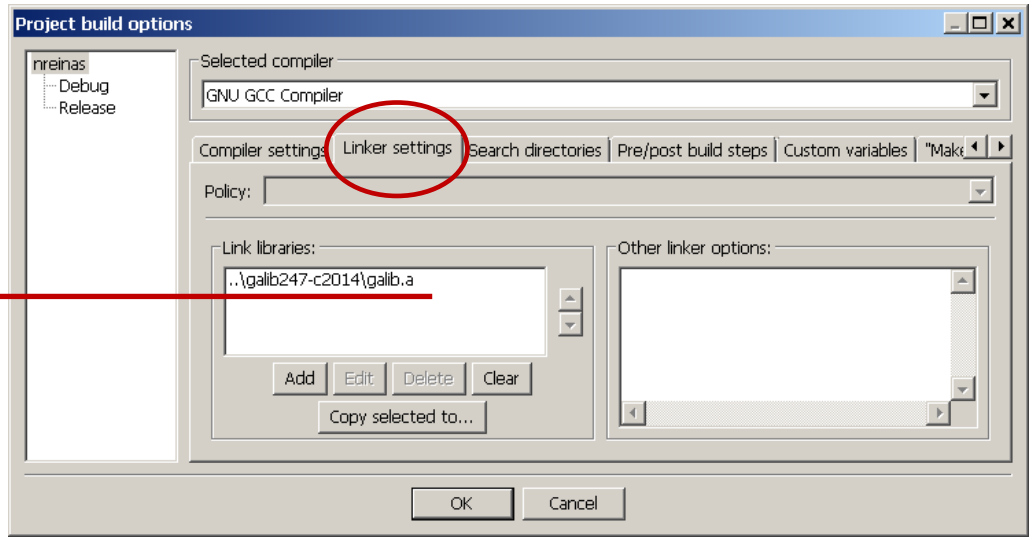
Uso de la librería GAlib en Code::Blocks

Por ejemplo, si llamamos `practica_1` al directorio de trabajo dentro de una carpeta creada para la asignatura `SSII`, la estructura de directorios sería la siguiente:

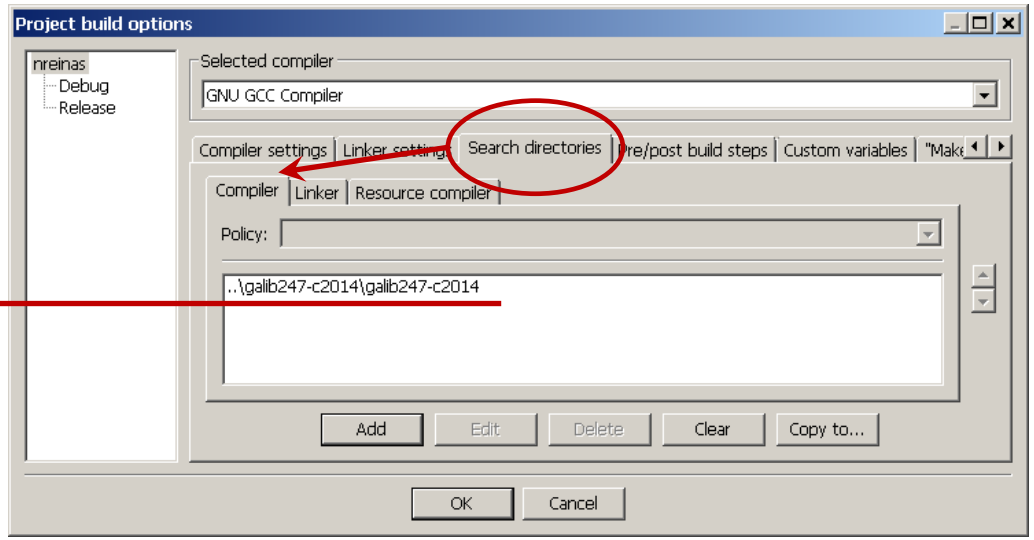


Uso de la librería GAlib en Code::Blocks

- Añadir la librería **`galib.a`** a las librerías estáticas en las opciones de proyecto



- Añadir el path para los includes



Implementación del AG para el problema de las n-reinas

El sistema implementado (basado en AG) tiene los parámetros (en orden):

número de reinas (nº entero)
tamaño de la población (nº entero)
número de generaciones (nº entero)
operador de selección (texto)
operador de cruce (texto)
operador de mutación (texto)
probabilidad de cruce (nº$\in$[0,1])
probabilidad de mutación (nº$\in$[0,1])

Veamos a continuación algunos detalles que hay que tener en cuenta en dicha implementación y el análisis de la salida del software.

Para los parámetros “operador de selección”, “operador de cruce” y “operador de mutación” vamos a utilizar únicamente dos posibles valores:

- Op. selección: ROUL o TOUR (GARouletteWheel o GATournament)
- Op. cruce: ONEP o TWOP (OnePointCrossover o TwoPointCrossover)
- Op. mutación: FLIP o SWAP (FlipMutator o SwapMutator)

Métodos de búsqueda local y problemas de optimización

Implementación del AG para el problema de las n-reinas


```
/* ---------------------- PROBLEMA DE LAS N REINAS ---------------------- */
#include <ga/GASimpleGA.h>
#include <ga/GA1DArrayGenome.h>
#include <iostream>
#include <fstream>
using namespace std;
……..
float Objective(GAGenome &);
……..
int main(int argc, char **argv)
{  ……..
   int nreinas=atoi(argv[1]);
   ……..
   int popsize = atoi(argv[2]);
   int ngen = atoi(argv[3]);
   string sel = mayuscula(string(argv[4]));
   string cross = mayuscula(string(argv[5]));
   string muta = mayuscula(string(argv[6]));
   float pcross = atof(argv[7]);
   float pmuta= atof(argv[8]);
   ……..
```

Métodos de búsqueda local y problemas de optimización

Implementación del AG para el problema de las n-reinas


```
// Creamos el genoma y definimos operadores de inicio, cruce y mutación
GA1DArrayAlleleGenome<int> genome(nreinas,alelos,Objective,NULL);
if (cross.compare("ONEP")==0) genome.crossover(GA1DArrayAlleleGenome<int>::OnePointCrossover);
else
   if (cross.compare("TWOP")==0) genome.crossover(GA1DArrayAlleleGenome<int>::TwoPointCrossover);
if (muta.compare("FLIP")==0) genome.mutator(GA1DArrayAlleleGenome<int>::FlipMutator);
else
    if (muta,compare("SWAP")==0) genome.mutator(GA1DArrayAlleleGenome<int>::SwapMutator);

// Creamos el algoritmo genético
GASimpleGA ga(genome);
……..
ga.minimaxi(-1);
ga.populationSize(popsize);
ga.nGenerations(ngen);
ga.pCrossover(pcross);
ga.pMutation(pmuta);
if (sel.compare("ROUL")==0) {  GARouletteWheelSelector selector;        ga.selector(selector);   }
else
      if (sel.compare("TOUR")==0) {  GATournamentSelector selector;          ga.selector(selector);   }
ga.evolve(1);

// Imprimimos el mejor individuo que encuentra el GA y su valor fitness
……..
}
……..
```

Siempre antes de crear el AG

Utilizamos en el parámetro de entrada el valor 1

Implementación del AG para el problema de las n-reinas


```
/* ---------------------- PROBLEMA DE LAS N REINAS ---------------------- */
#include <ga/GASimpleGA.h>
……..

int main(int argc, char **argv)
{
   ……..
   ga.nGenerations(ngen);
   ……..

   ga.evolve(1);
   ……..
}
……..
```

ESPECIFICANDO EL OPERADOR DE TERMINACIÓN

Por defecto

```
GABoolean Termina(GAGeneticAlgorithm & ga)
 {
   if (ga.statistics().generation()==ga.nGenerations()) return gaTrue;
   else return gaFalse;
 }
```

Implementación del AG para el problema de las n-reinas

ESPECIFICANDO EL OPERADOR DE TERMINACIÓN

```
/* ---------------------- PROBLEMA DE LAS N REINAS ---------------------- */
#include <ga/GASimpleGA.h>
……..
GABoolean Termina(GAGeneticAlgorithm &);

int main(int argc, char **argv)
{
   ……..
   ga.nGenerations(ngen);
   ……..
   ga.terminator(Termina);
   ga.evolve(1);
   ……..
}
……..
// Funcion de terminacion

GABoolean Termina(GAGeneticAlgorithm & ga)
{
   if ( ga.statistics().minEver()==0 ) return gaTrue;
   else return gaFalse;
}
```

Implementación del AG para el problema de las n-reinas

ESPECIFICANDO EL OPERADOR DE TERMINACIÓN

```
/* ---------------------- PROBLEMA DE LAS N REINAS ---------------------- */
#include <ga/GASimpleGA.h>
……..
GABoolean Termina(GAGeneticAlgorithm &);

int main(int argc, char **argv)
{
   ……..
   ga.nGenerations(ngen);
   ……..
   ga.terminator(Termina);
   ga.evolve(1);
   ……..
}
……..
// Funcion de terminacion

GABoolean Termina(GAGeneticAlgorithm & ga)
{
   if ( (ga.statistics().minEver()==0) || (ga.statistics().generation()==ga.nGenerations()) ) return gaTrue;
   else return gaFalse;
}
```

Implementación del AG para el problema de las n-reinas

- Ejemplo de uso (salida):

Obsérvese que, en el código, se numeran las filas y columnas de 0 a 7.

si mostramos el genoma →

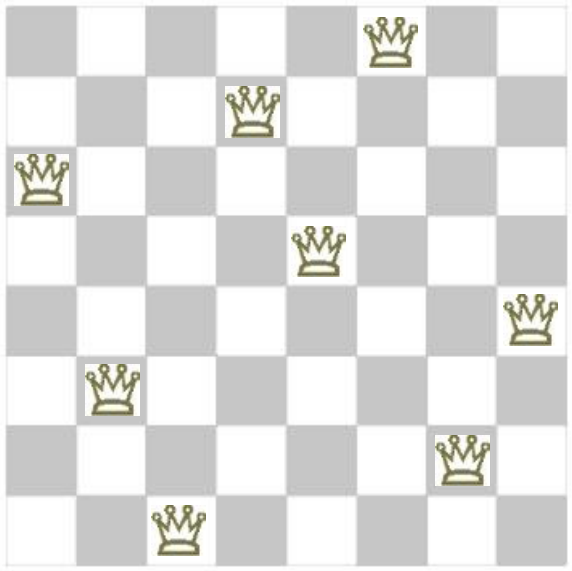
Problema de las 8 reinas

Parámetros: (100, 1000, ROUL, ONEP, FLIP, 0.9, 0.1)

El sistema encuentra la solución

(2 5 7 1 3 0 6 4)

con 0 jaques.



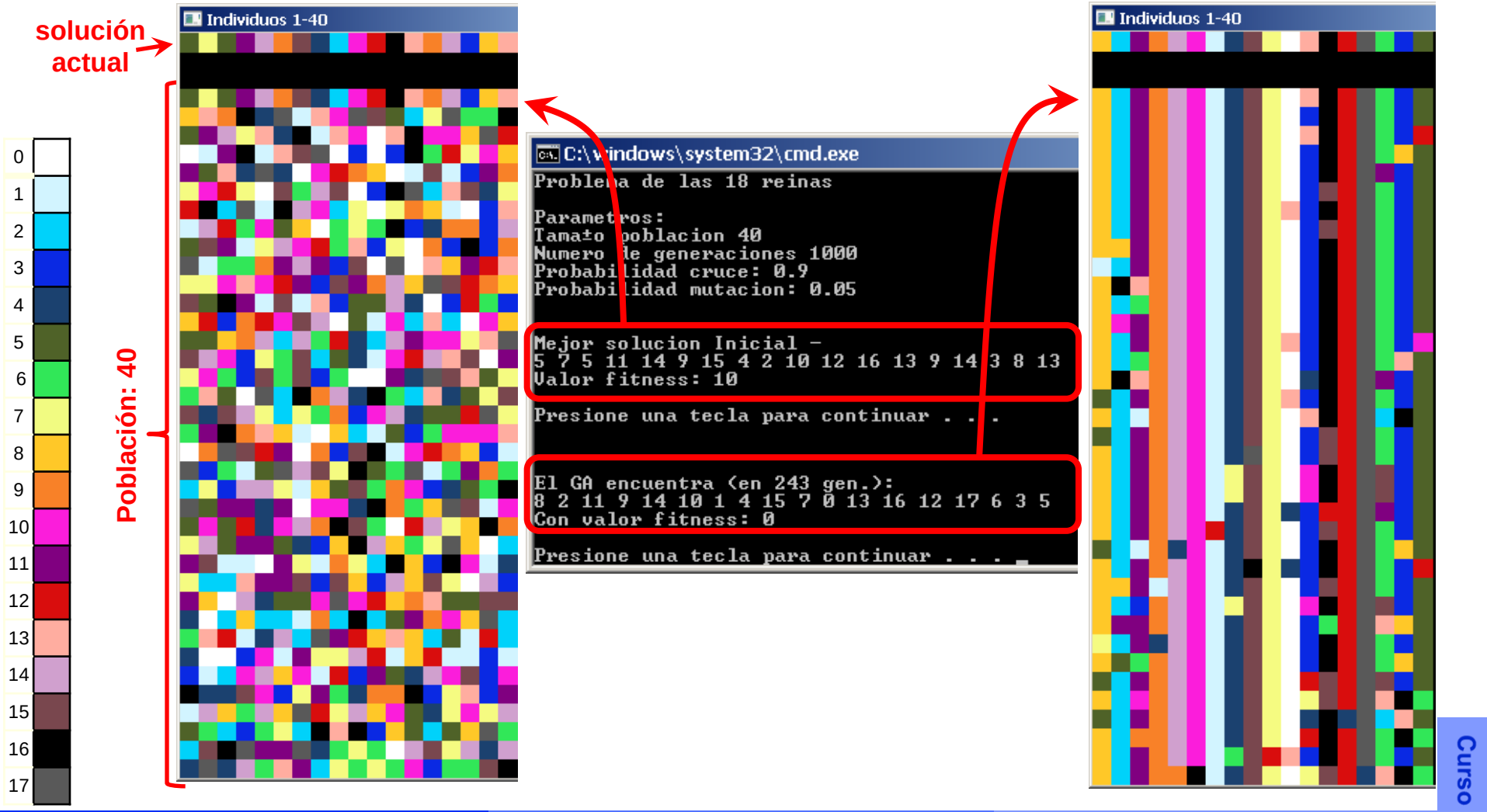
Es solución óptima, pues fitness es 0

2	5	7	1	3	0	6	4

Ojo, ¿es interpretable la solución para un usuario final? ¿Por qué?

Implementación del AG para el problema de las n-reinas

¿CÓMO SE COMPORTA INTERNAMENTE EL ALGORITMO GENÉTICO?



Terminando de construir el sistema basado en un AG

Para terminar el sistema basado en el AG diseñado y construido debemos:

- Encontrar las mejores combinaciones de operadores genéticos-valores de parámetros que hagan que el AG tenga un comportamiento óptimo, "bueno" o "aceptable". Este será el objetivo del ajuste de parámetros.
- A partir de este ajuste, debemos construir un manual que le sirva al usuario final a decidir, para un problema dado, los valores concretos de parámetros y operadores que debe utilizar (Manual de Asignación)

Veamos como ajustar el AG que hemos construido para el problema de las n-reinas.

AJUSTE DE PARÁMETROS: ESTUDIO DE CASOS – VALORES DE PARÁMETROS

Como ya hemos comentado, una vez diseñado/construido el AG para resolver el problema de las n-reinas, debemos realizar el ajuste de parámetros. A continuación describimos los pasos para este ajuste:

a) Parámetros del AG.

Parámetros con valores fijos: nº generaciones = 10000
operador de cruce = ONEP
operador de mutación = FLIP

Parámetros ajustables con valores:

→ tamaño de la población (TP)= {100,150}
→ método de selección = {ROUL, TOUR}
→ p_c = {0.8, 0.85}
→ p_m = {0.025, 0.05, 0.075}

b) Conjunto de casos del problema: de 8 a 45 reinas

c) Resolver todos los casos utilizando todas las combinaciones de valores

Métodos de búsqueda local y problemas de optimización

1

Selector = ROUL

nreinas		TP=100						TP=150					
		p_c=0.8			p_c=0.85			p_c=0.8			p_c=0.85		
	p_m	0.075	0.05	0.025	0.075	0.05	0.025	0.075	0.05	0.025	0.075	0.05	0.025
8		0	0	0	0	0	0	0	0	0	0	0	0
9		0	0	0	0	0	0	0	0	0	0	0	0
10		0	0	0	0	0	0	0	0	0	0	0	0
11		0	0	0	0	0	0	0	0	0	0	0	0
12		0	0	0	0	0	0	1	0	0	0	1	0
13		0	0	0	0	0	1	0	0	0	1	0	0
14		0	0	0	1	0	0	0	0	0	1	0	1
15		0	0	0	0	0	0	1	0	0	1	1	0
16		1	0	0	1	0	0	1	1	0	1	0	0
17		1	0	0	1	0	0	1	1	0	1	0	0
18		2	1	0	1	1	0	0	1	0	1	0	0
19		1	0	0	1	0	0	2	1	0	1	1	0
20		2	0	1	1	0	0	3	0	1	1	0	0
21		1	1	1	2	2	0	2	2	0	2	2	0
22		2	2	1	1	1	0	1	1	0	2	2	1
23		3	3	0	0	0	0	3	3	1	2	2	1
24		2	2	1	2	2	0	2	2	2	2	2	1
25		2	2	0	2	2	0	1	1	1	3	3	0
26		3	3	0	3	3	0	1	1	0	3	3	1
27		5	3	2	4	2	0	4	3	1	3	3	2
28		5	3	1	4	2	0	4	2	1	5	3	1
29		4	3	1	2	2	0	2	2	2	4	1	2
30		5	2	2	6	3	0	6	3	1	6	3	1
31		7	3	0	5	2	1	4	3	2	6	3	0
32		5	3	1	4	2	2	4	3	2	5	2	1
33		5	2	2	7	3	1	5	3	1	4	4	1
34		5	4	1	6	2	1	5	3	2	4	4	2
35		6	3	2	8	4	1	4	4	2	5	4	1
36		6	4	2	6	4	3	8	5	2	6	3	1
37		7	2	2	7	4	1	8	4	4	6	6	1
38		6	3	3	6	4	4	6	3	1	6	4	4
39		7	5	2	7	5	2	7	5	2	7	7	3
40		8	3	1	6	4	2	8	5	1	8	6	2
41		5	8	4	9	8	5	10	7	3	10	7	4
42		7	7	4	8	9	6	8	9	5	11	7	6
43		8	8	4	11	7	4	10	7	5	11	9	5
44		12	9	5	10	8	5	10	7	4	9	9	5
45		11	6	5	11	9	5	8	7	5	9	9	6

Métodos de búsqueda local y problemas de optimización

2

Selector = TOUR

nreinas	p_m	TP=100						TP=150					
		p_c=0.8			p_c=0.85			p_c=0.8			p_c=0.85		
		0.075	0.05	0.025	0.075	0.05	0.025	0.075	0.05	0.025	0.075	0.05	0.025
8		0	0	0	0	0	1	0	0	0	0	0	0
9		0	0	0	0	0	0	0	0	0	0	0	0
10		0	0	0	0	0	0	0	0	0	0	0	0
11		0	0	0	0	0	0	0	0	1	0	0	0
12		0	0	0	0	0	0	0	0	0	0	0	0
13		0	0	0	0	0	0	0	0	0	0	0	0
14		0	0	0	0	0	0	0	0	0	0	0	0
15		0	0	0	0	0	0	1	0	0	0	0	0
16		0	0	0	0	0	0	1	0	0	0	0	0
17		1	0	0	1	0	0	1	0	0	1	0	0
18		1	0	0	1	0	0	0	0	0	1	0	0
19		1	0	0	1	0	0	1	0	0	0	0	0
20		0	0	0	0	0	0	1	0	0	0	0	0
21		1	1	0	0	0	0	1	1	0	1	1	0
22		0	0	0	0	0	0	0	0	0	0	0	0
23		2	2	0	1	1	0	1	1	0	1	1	0
24		2	2	0	1	1	0	1	1	0	1	1	0
25		1	1	0	1	1	0	0	0	0	1	1	0
26		1	1	0	2	2	0	2	2	0	1	1	0
27		1	1	0	3	1	0	3	1	0	4	1	0
28		3	1	0	3	1	0	3	2	0	3	2	0
29		3	2	0	3	1	0	4	1	0	3	1	0
30		2	0	0	5	2	0	4	1	0	4	1	0
31		4	2	0	3	2	0	3	1	0	4	2	0
32		4	3	0	4	1	0	3	1	0	4	1	0
33		4	2	0	4	2	0	5	2	0	2	2	0
34		5	2	0	6	2	0	5	2	0	5	2	0
35		3	2	0	5	3	0	5	2	0	3	2	0
36		4	3	0	5	2	0	5	3	0	4	1	0
37		3	2	0	5	3	0	6	2	0	6	1	0
38		5	2	0	6	2	0	6	3	0	4	2	0
39		5	3	0	4	2	0	5	3	0	4	2	0
40		6	4	0	5	3	0	5	3	0	3	3	0
41		9	6	2	7	6	3	8	5	4	7	5	3
42		8	6	3	9	6	3	10	6	3	7	6	3
43		8	5	3	9	5	3	10	4	4	7	5	3
44		8	5	4	6	6	2	9	7	3	8	6	3
45		10	5	5	8	6	3	10	8	4	7	8	3

Terminando de construir el sistema basado en un AG

d) se analizan los resultandos obtenidos (Análisis de Resultados)

Las decisiones tienen que estar guiadas por los resultados y fundamentadas por los cuatro criterios “Completitud”, “Optimalidad”, “Complejidad en Tiempo” y “Complejidad en Espacio”.

Con el comportamiento que observemos, debemos realizar una generalización del proceso de asignación de valores a parámetros y casos a resolver.

- Primero marcamos las combinaciones de valores que obtienen un "0" para todos los casos resueltos (buscando la optimalidad).
- En la Tabla 2 se observan más soluciones 0 que en la Tabla 1. Y además, para los casos que no obtienen “0” tienen un número de “jaques” sensiblemente menor que los de la Tabla 1. Por tanto, el operador de selección “TOUR” (**GATournament**) tiene un mejor comportamiento que el operador de selección “ROUL” (GARouletteWheel).
- Centrándonos ahora en los resultados de la Tabla 2 (con operador de selección GATournament). No hay diferencias significativas (comportamiento similar) entre el tamaño de la población. Podríamos utilizar cualquiera de ellos, pero como la complejidad en espacio debería estar caracterizada por el tamaño de la población (entre otros), seleccionamos el valor **TP=100**.
- Entonces, nos queda analizar el comportamiento para los parámetros p_c y p_m en las columnas de la Tabla 2 referidas a TP=100.

Podríamos obtener otros comportamientos diferentes

Terminando de construir el sistema basado en un AG

Valores de p_c y p_m para tamaños de problemas en los que se obtiene solución óptima

	p_m	nreinas
p_c=0.8	0.025, 0.05, 0.075 0.025, 0.05 0.025	8 – 16, 20, 22 8 – 20, 22, 30 8 – 40
p_c=0.85	0.05, 0.075 0.025, 0.05, 0.075 0.025, 0.05 0.025	8 9 – 16, 20 – 22 9 – 22 9 – 40
p_c=0.8 **p_c=0.85**	No hay valor para el parámetro que obtenga 0 jaques	41 – 45

Tanto para p_c=0.8 y p_c=0.85, el sistema tiene un comportamiento similar obteniendo valores 0 en prácticamente los mismos casos. Según la complejidad en tiempo, seleccionamos **p_c=0.8** por requerir menos recursos.

Podríamos obtener otros comportamientos diferentes

Terminando de construir el sistema basado en un AG

Resumiendo: El sistema para 10000 generaciones, con operador de cruce "OnePointCrossover" y operador de mutación "Flipmutator" y utilizando los casos para ajuste de tamaños 8-45, tiene su mejor comportamiento (según los criterios indicados en el análisis) con:

- Operador de selección = TOUR (GATournamentSelector).
- Tamaño de la población de =100
- $p_c = 0.8$

  $p_m = \{0.025, 0.05, 0.75\}$ para 8-16, 20 y 22 reinas

  $p_m = \{0.025, 0.05\}$ para 8-20, 22, 30 reinas

  $p_m = \{0.025\}$ para 8-40 reinas

Para este análisis/ajuste inicial, podemos concluir que los distintos parámetros están definidos excepto para "p_m" cuyo comportamiento nos induce que conforme crece el problema (mayor número de reinas) el valor de "p_m" debería decrecer. Para terminar de ajustar este parámetros p_m debemos ejecutar nuevos casos y con nuevos valores de p_m (disminuyendo su valor).

Podríamos seleccionar, por requerir menos recursos y ser más estable (recoge a mayor número de casos), los valores de p_m=0.025 para valores de nreinas $\leq$ 40.

HACIA EL MANUAL DE ASIGNACIÓN

El manual de usuario nos indica cómo debemos ejecutar el software y qué tipo de información se nos pedirá asociada a los distintos parámetros. Lo que NO nos indica es qué valor concreto debemos asignar a cada parámetro en función del caso que se resolverá y para que tenga un comportamiento óptimo o aceptable. La función del “Manual de Asignación" debe ser esta.

El Manual de Asignación se debe obtener a partir de un estudio previo de casos y valores de parámetros. A partir del estudio realizado, el Manual de Asignación debe indicar claramente al usuario qué valores/métodos debe utilizar en los parámetros del sistema para un caso concreto a resolver.

Nota: Tanto un Manual de Usuario como un Manual de Asignación van dirigidos al usuario y por tanto, no deben incluir detalles técnicos.

A modo de ejemplo, un sencillo Manual de Asignación podría ser el siguiente:

Terminando de construir el sistema basado en un AG

Recomendaciones para la asignación de valores a parámetros para que el sistema tenga un comportamiento aceptable:

Al resolver un caso concreto del problema de las n-reinas le recomendamos los siguientes valores para los parámetros que le pedirá el sistema con el fin de obtener buenos resultados. Algunos valores recomendados van en función del valor de “n” (nº de reinas: tamaño del caso) correspondiente al parámetro 1º:

Valor para el parámetro 2º: 100 (valor recomendado)
Valor para el parámetro 3º: 10000 (valor recomendado)
Valor para el parámetro 4º: TOUR (valor recomendado)
Valor para el parámetro 5º: ONEP (valor recomendado)
Valor para el parámetro 6º: FLIP (valor recomendado)
Valor para el parámetro 7º: 0.8 (valor recomendado)
Valor para el parámetro 8º:

- Si --- ≤ n ≤ ---: xx (valor recomendado)
- Si --- ≤ n ≤ ---: yy (valor recomendado)
- .....
- Si --- ≤ n ≤ ---, zz (valor recomendado)

Sobre la compilación con GAlib

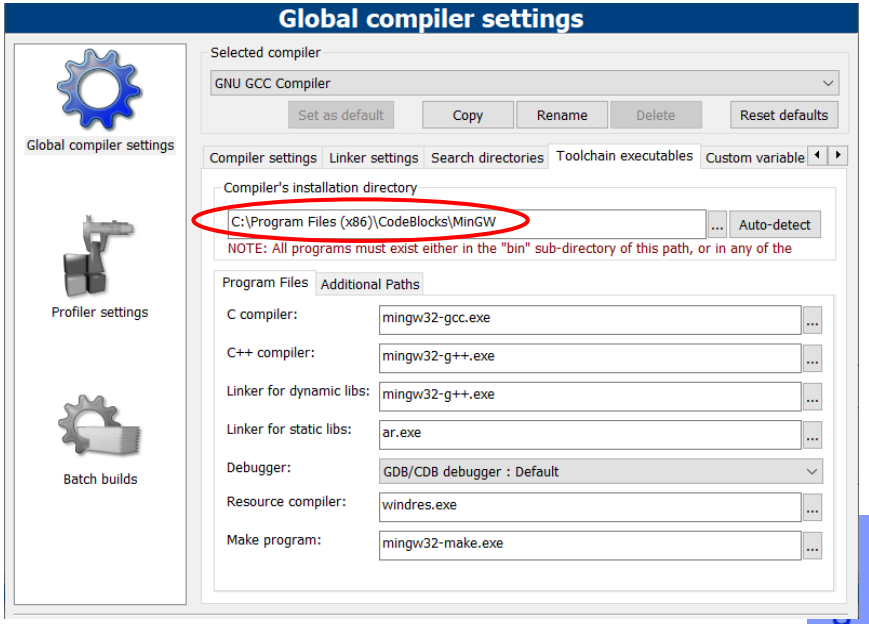
Su uso con Code::Blocks para Windows

En las transparencias anteriores se indica cómo se utiliza la librería GAlib en un proyecto desarrollado en Code::Blocks para Windows. Para que no haya ningún problema en el uso de la librería se debe tener en cuenta lo siguiente:

- Primero, debe descargarse la versión de Code::Blocks con el compilador "MinGW" incluido (codeblocks-17.12mingw-setup.exe) e instalarlo, la cual se puede encontrar disponible de manera pública en:

  https://sourceforge.net/projects/codeblocks/files/Binaries/17.12/Windows/

- Segundo, hay que asegurarse que no tenemos ningún otro compilador C/C++ en el ordenador, o que, en la compilación de nuestro proyecto, Code::Blocks está utilizando el compilador MinGW instalado con Code::Blocks (lo podemos comprobar dentro de Code::Blocks en setting-compiler--Toolchain Executables) (ver figura).



Sobre la compilación con GAlib

Su uso con Code::Blocks para Windows

En caso de no querer realizar lo anteriormente expuesto, se recomienda la opción de utilizar el entorno virtual EVA (“INFORMATICA_WINDOWS”).

En el entorno virtual se puede utilizar Code::Blocks, teniendo en cuenta que se utilizará únicamente para compilar y crear el .exe de nuestro proyecto.

Con dicho .exe podremos ya, en nuestro ordenador, realizar los experimentos indicados en el guion de la práctica.