Diseño e implementación del juego de cifras empleando algoritmos genéticos

Grupo 5

Sandra Saéz Raspeño

Lissete Moscoso León

Índice

Introducción

Introducción (1)

- El juego de las cifras consiste en generar una expresión aritmética que al evaluarse se aproxime lo máximo posible a un número objetivo.
- La expresión aritmética se obtiene de combinar seis números usando operaciones básicas (suma, resta y multiplicación).
- Es importante que la expresión aritmética mantenga el orden de los números dados.
- El objetivo consiste en minimizar el valor absoluto de la diferencia del número objetivo y el valor de la expresión aritmética obtenida.

Introducción (2)

Por ejemplo, dados los números: 25, 6, 9, 75, 50 y 3, las siguiente secuencia de operadores se interpretan como:

Secuencia	1ª operación	2ª operación	3ª operación	4ª operación	5ª operación
+x+-+	25+6=31	31x9=279	279+75=354	354-50=304	304+3=307
--++x	25-6=19	19-9=10	10+75=85	85+50=135	135x3=405

Software

Galib

- Biblioteca de algoritmos genéticos desarrollada en C++.
- Potente y flexible sistema de clases
- Permite personalizar con facilidad
- Implementa varios algoritmos genéticos y multitud de operadores
- Sólo es necesaria la codificación de la función objetivo

Implementación

Diseño usando algoritmos genéticos

Pasos previos (1)

Implementación

- Representación de la población (1):

  Para la representación de la población hemos utilizado el genotipo **GAStringGenome**, se ha escogido esta representación debido a que permite la representación con alelos no binarios. En nuestro caso los alelos son "**+, -, ***".

```
55      GAStringAlleleSet alleles;
56      alleles.add('+');
57      alleles.add('-');
58      alleles.add('*');
```

Pasos previos (2)

Implementación

- Representación de la población (2):

  Para la creación de nuestro genotipo tendremos que especificar los siguientes valores:
  - Longitud del genotipo.
  - Conjunto de alelos de los cuales estará formado.
  - Función objetivo

```
59
60      GAStringGenome genome(length, alleles, objective);
61
```

Pasos previos (3)

Implementación

- Selección de los parámetros:

  Estos parámetros son muy importantes para que el algoritmo genético nos de buenos resultados. Los parámetros a especificar son:

  - Tamaño de población.
  - Número de generaciones.
  - Probabilidad de cruce.
  - Probabilidad de mutación.

```
34   int popsize  = 50;
35   int ngen     = 250;
36   float pmut   = 0.05;
37   float pcross = 0.9;
```

Inicialización de la población

Implementación

En nuestra implementación hemos optado por una inicialización totalmente aleatoria de la población.

```
63      genome.initializer(OperatorsInitializer);
```

```
132  void OperatorsInitializer(GAGenome & c)
133  {
134    GAStringGenome &genome=(GAStringGenome &)c;
135    int i;
136    char operators [] = {'+', '-', '*'};
137    for (i = 0; i < genome.size(); i ++){
138      genome.gene(i, operators[GARandomInt(0, 2)]);
139    }
140    for(i=0; i<genome.size(); i++)
141      if(GARandomBit()) genome.swap(i, GARandomInt(0, genome.size()-1));
142  }
```

Algoritmo genético (1)

Implementación

El algoritmo genético controla el proceso de evolución, determina qué individuos se reproducirán, qué individuos serán reemplazados y cuales sobrevivirán, así como la terminación de la evolución.

En nuestra implementación nos hemos centrado en dos de los algoritmos proporcionados por Galib.

- GASimpleGA.
- GASteadyStateGA.

Algoritmo genético (2)

Implementación

```
69      GASimpleGA ga(genome);
70      ga.populationSize(popsize);
71      ga.nGenerations(ngen);
72      ga.pMutation(pmut);
73      ga.pCrossover(pcross);
74      ......
```

Función objetivo (1)

Implementación

Hemos diseñado la función objetivo para que minimice la diferencia entre el valor objetivo y el valor obtenido tras evaluar la expresión matemática obtenida que representa el individuo.

- Por defecto la biblioteca que utilizamos escoge como mejores individuos aquellos que maximicen la función objetivo. Como en nuestro caso queremos minimizar el valor obtenido, tendremos que modificar este comportamiento.

```
74      ga.minimize();
```

Función objetivo (2)

Implementación

```
115  float
116  objective(GAGenome & c)
117  {
118    GAStringGenome & genome = (GAStringGenome &)c;
119    int result = values [0];
120    for(int i=0; i<genome.size(); i++){
121      result = Operations (result, values[i+1], genome.gene(i));
122    }
123    return abs(result - optimal);
124  }
```

Función objetivo (3)

Implementación

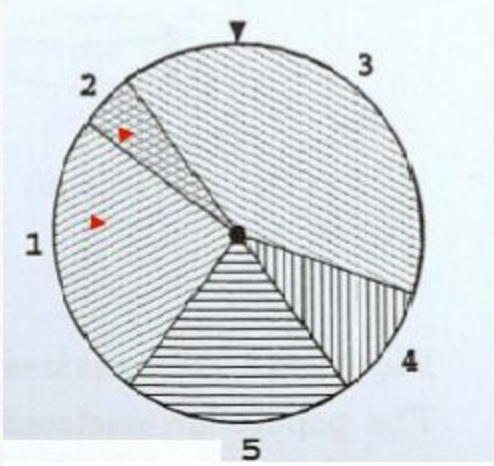
```
 98  int Operations(int firstOperator, int secondOperator, char operation){
 99    if (operation == '+'){
100      return firstOperator + secondOperator;
101    }
102    else if (operation == '-'){
103      return firstOperator - secondOperator;
104    }
105    else {
106      return firstOperator * secondOperator;
107    }
108  }
```

Operador de selección

Implementación

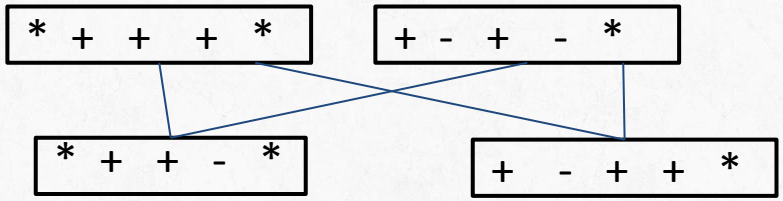
Este operador selecciona, entre los distintos individuos de la población, aquellos que van a tener descendencia, en nuestro caso hemos utilizado el operador de selección por defecto:

GARouletteWheelSelector (método de la ruleta).

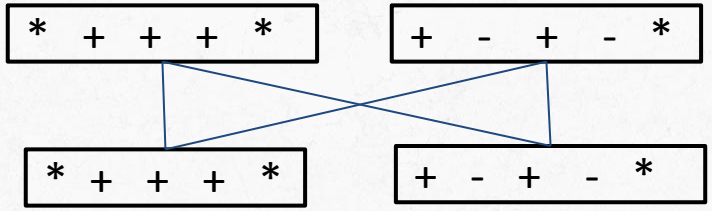


Operadores de cruce

- Cruce en un punto (ej: tercer gen)



- Cruce en dos puntos (ej: segundo y cuarto gen)



Operador de cruce

Implementación

Para la solución de nuestro problema hemos hecho uso del operador de cruce **OnePointCrossover**. Para especificar que queremos utilizar este operador tendremos que añadirlo al algoritmo genético.

```
76      ga.crossover(GAStringOnePointCrossover);
```

Operadores de mutación

- Mutación en un gen (FlipFlop)

  ej: Suponiendo mutación en el segundo gen

- Mutación en dos genes (Swap)

  ej: Suponiendo mutación entre el segundo y cuarto gen

Operador de mutación

Implementación

La biblioteca nos permite implementar dos tipos de mutaciones:

- Mutación FlipFlop.
- Mutación Swap.

Para la implementación de nuestro problema nos hemos decantado por el operador de mutación FlipFlop, ya que hemos comprobado que con este operador obtenemos la solución en un menor número de generaciones (converge más rápido).

```
67      genome.mutator(GAStringFlipMutator);
```

Función de terminación (1)

Implementación

Existen tres métodos de terminación normalmente utilizados:

- Convergencia de la población (TerminateUponConvergence).
- Aparición de la solución que satisface la función objetivo.
- Se alcanza el número máximo de iteraciones estipuladas (TerminateUponGeneration).

```
82      ga.terminator(GATerminateWhenIsFounded);
```

Función de terminación (2)

Implementación

Hemo definido nuestra propia función para realizar la terminación del algoritmo, la cual se basa en parar cuando se llega al número de generaciones máximas establecidas o cuando se encuentra el valor óptimo 0.

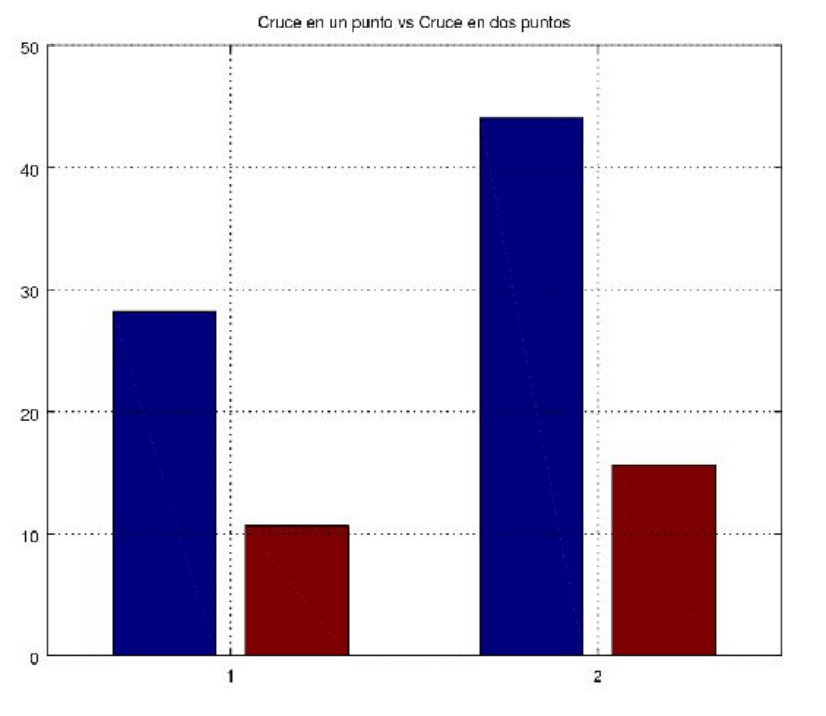
```
154  GABoolean GATerminateWhenIsFounded(GAGeneticAlgorithm &ga)
155  {
156      if(ga.generation() == ga.nGenerations()){
157        return gaTrue;
158      }
159      else {
160        return (ga.statistics().bestIndividual().score() == 0 ? gaTrue : gaFalse);
161      }
162  }
```

Resultados

Comparación de operadores de cruce

- Objetivo: comprobar con qué operador de cruce la velocidad de convergencia es mayor.
- Problemas:
  - el cruce en un punto aporta exploración.
  - el cruce en dos puntos aporta explotación.
- Posible solución: dar un valor adecuado a la probabilidad de mutación.

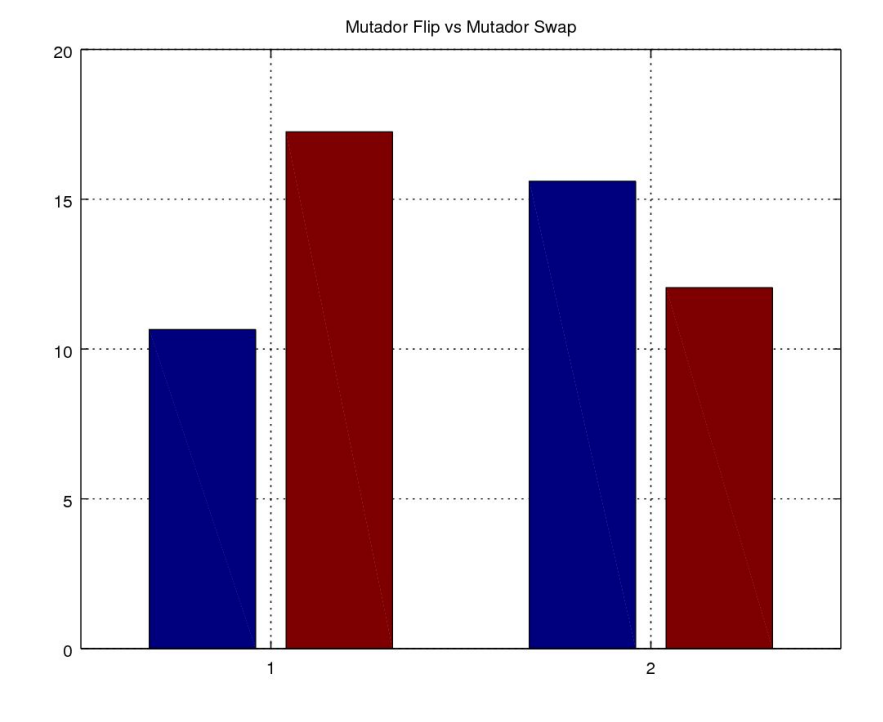
Resultado



Comparación de operadores de mutación

- Objetivo: comprobar con qué operador de mutación la velocidad de convergencia es mayor.
- Se deja establecido el 5% de mutación calculado anteriormente
- Problema: Dependiendo de si el cruce aporta exploración o explotación unos operadores de mutación funcionarán mejor que otros.

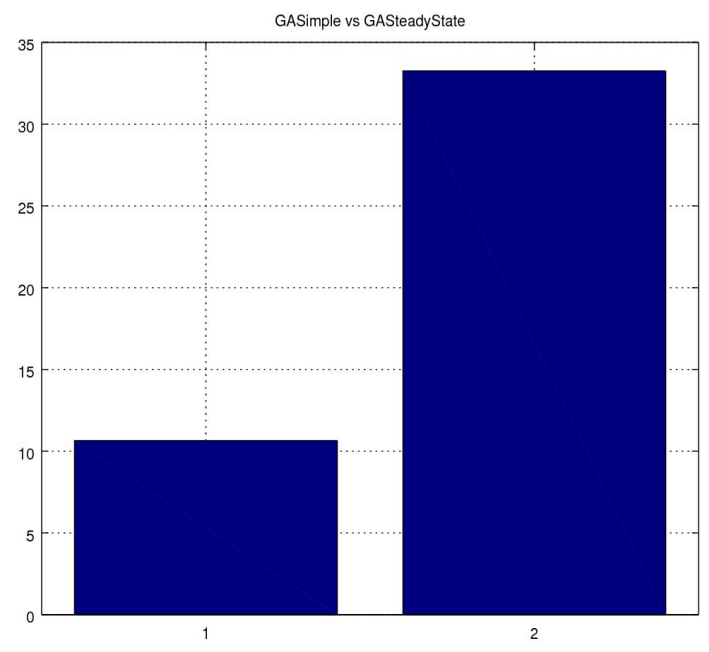
Resultado



Comparación entre algoritmos genéticos

- **GASimple**: en cada generación reemplaza por completo su población.
- **GASteadyState**: en cada generación reemplaza sólo unos pocos individuos tomando los mejores de la nueva generación obtenida tras el cruce y mutación.
- Datos de las simulaciones:
  - Probabilidad de mutación = 5%; Probabilidad de cruce = 90%
  - Operador de mutación: FlipFlop; Operador de cruce: Cruce en un punto

Resultados



Conclusiones

Conclusiones

- Los algoritmos genéticos son un proceso de prueba y error. Aunque en nuestro caso el problema es bastante sencillo, existen problemas más complicados que pretenden llegar más rápido a la solución.

- Para poder sacar más provecho a los algoritmos genéticos es importante conocer cada uno de los operadores de los algoritmos.

- Es importante la correcta selección de los parámetros, ya que influirá en la convergencia del problema.

Referencias

Referencias

- Font F., J. M., Manrique G., D., Ríos C., J., “Redes neuronales artificiales y computación evolutiva”, 2009.
- Mateos Caballero, Alfonso. Búsqueda inteligente basada en metaheurísticas. Transparencias de clase.
- Galib documentation, version 2004, http://lancet.mit.edu/galib-2.4/.
- Arranz de la Peña, J., Parra Truyol, Antonio, “Algoritmos genéticos”, http://www.it.uc3m.es/jvillena/irc/practicas/06-07/05.pdf.
- “Implementación de juegos usando algoritmos evolutivos”, http://eprints.ucm.es/9144/1/TC2004-4.pdf.
- “Using Galib”, http://www.egr.unlv.edu/~bein/teaching/adaptive/UsingGAlib.pdf.