Resolución de problemas criptoaritméticos utilizando Algoritmos de Enjambre

Jerónimo Sodero, Matías Wajnman, y Nelson Yaccuzzi

Alumnos de 5to Nivel de Ingeniería en Sistemas de Información, Inteligencia Artificial, Universidad Tecnológica Nacional, Resistencia, Argentina

{jerosode,matiasarw,nelsonyaccuzzi}@gmail.com

Abstract. Los problemas criptoaritméticos son problemas que consisten en la obtención de relaciones matemáticas entre palabras con significado utilizando operadores sencillos como la suma o la resta. El objetivo es asignar dígitos a las letras de las palabras de manera que el resultado de las operaciones aritméticas sea correcto. En este trabajo se implementa una propuesta de solución para problemas de este tipo, utilizando una clase especial de algoritmos de enjambre llamado algoritmo Firefly. Además se analiza su comportamiento mediante la variación de parámetros generales que modifican el rendimiento del mismo.

Keywords: Algoritmo Firefly, Problemas criptoaritméticos, Algoritmos de enjambre, Algoritmos Bio-inspirados, Optimización de funciones discretas.

1 Introducción

La criptoaritmética [1], conocida también como criptoaritmo, aritmética verbal, es un tipo de juego que consiste en una ecuación matemática entre números desconocidos, cuyos dígitos son representados por letras, en donde el objetivo del mismo es identificar el valor de cada una.

La ecuación está basada en una operación aritmética básica, como la suma, resta, multiplicación o división. El ejemplo clásico, el cual se tomará de referencia, fue publicado en Julio de 1924 por Henry Dudeney [1], y es el siguiente:

```
    S E N D
+   M O R E
-----------
= M O N E Y
```

Fig. 1. Problema Criptoaritmético de referencia

Y su respectiva solución es: O = 0, M = 1, Y = 2, E = 5, N = 6, D = 7, R = 8, and S = 9.

Como restricción cada letra solo podrá representar un dígito distinto como en la notación aritmética tradicional. En un buen problema las letras forman una frase como en el ejemplo anterior.

Una alternativa de solución de problemas criptoaritméticos es la desarrollada por Abbasian y Mazloom [2] en el año 2009, mediante la utilización de algoritmos genéticos optimizados con uso de paralelismo para obtener ganancias en términos de escalabilidad y performance. En dicho trabajo se considera a la solución/individuo como un cromosoma y se evalúa de acuerdo a una función fitness que determina la calidad del individuo. Para obtener la nueva generación, se utiliza la mutación evitando mínimos locales.

Otro estudio, llevado a cabo por Fontanari [3] en 2014, hace uso de Aprendizaje Social Heurístico. En el mismo, se parte de la premisa de que un grupo colaborativo de N agentes puede resolver un problema más eficientemente que el mismo grupo trabajando de forma independiente. La cooperación se da mediante el aprendizaje imitativo, que permite que la información se pase de un agente a otro. De esta forma, en cada prueba, un agente puede explorar el espacio de soluciones mediante prueba y error, o copiar pistas de un agente modelo.

El trabajo está estructurado de la siguiente manera: en la sección 2 se presenta el algoritmo de enjambre Firefly, la sección 3 se utiliza para explicar la solución implementada, detallando las funciones importantes y parámetros que influyen en la ejecución del mismo. Luego, en la sección 4 se analiza el algoritmo cuando varían estos parámetros mediante pruebas y sus posteriores resultados. Finalmente, la sección 5 se incluyen las conclusiones obtenidas.

2 Algoritmo de Enjambre Firefly

Los algoritmos basados en inteligencia de enjambre [4] son una clase especial de algoritmos bio-inspirados. Este tipo de algoritmos usa el comportamiento de enjambres para obtener buenas soluciones para muchos problemas de optimización.

El algoritmo de enjambre seleccionado para la solución propuesta en este trabajo es el algoritmo Firefly [5,6], el cual está basado en la intensidad de la luz de las luciérnagas (fireflies) y la atracción generada por su brillo. Este tipo de algoritmo es usado ampliamente para todo tipo de problemas, dado su gran rendimiento y simplicidad.

Cada luciérnaga (conjunto de N variables) representa una solución posible del problema, cuya optimalidad se mide a través de su brillo. El brillo es calculado a través de una función objetivo que se define específicamente para el tipo de problema afrontado.

Las luciérnagas distribuidas en un plano N dimensional se mueven constantemente dentro de ciclos predefinidos hacia aquellas de mayor brillo, dependiendo de la distancia que las separa y del coeficiente de acercamiento definido (atractividad).

Cada movimiento que realiza una luciérnaga posee además una componente aleatoria, que permite a las mismas explorar en mayor medida partes del espacio de soluciones desconocidas.

El algoritmo puede finalizar una vez encontrada una solución óptima y/o cuando el número de ciclos supera un valor predefinido.

3 Solución Implementada[1]

3.1 Representación de las luciérnagas

La cantidad de variables del problema está definida por la cantidad de letras distintas que se presenten. Cada luciérnaga será, entonces, una configuración de valores concreta para las mismas. Las variables solo pueden tomar valores enteros de 0 a 9 y entre estas no se pueden repetir.

Cada variable representará entonces a cada una de las letras del problema como se muestra en la Fig.2.

```
{S: 1; E: 2; N: 3; D: 4; M: 5; O: 6; R: 7; Y: 8}
```

Fig. 2. Ejemplo de la representación de una luciérnaga

3.2 Brillo de las luciérnagas

El valor del brillo de cada luciérnaga es calculado mediante las siguientes ecuaciones:

$$Error = SumaDigitos(|Resultado - (Operador1 + Operador2)|) \quad (1)$$

$$Error = SumaDigitos(|Resultado - (Operador1 - Operador2)|) \quad (2)$$

Donde la ecuación (1) se utiliza para la operación suma y la ecuación (2) para la operación resta. En sentido contrario a lo definido por el marco teórico, el brillo es mayor cuando menor sea el resultado de la ecuación.

La función *SumaDigitos()* realiza la suma aritmética de todos los dígitos del valor pasado como parámetro y se utiliza para obtener valores más chicos de error. Una solución óptima del problema es aquella en donde *Error* es igual a 0.

El Error de la luciérnaga expresada en Fig. 2 considerando un problema de suma se calcula de la siguiente manera:

$$Error = SumaDigitos(56328 - (1234 + 5672)) = SumaDigitos(49422)$$

$$Error = 21$$

3.3 Acercamiento de las luciérnagas

El movimiento entre luciérnagas fue implementado mediante la función *acercar()* la cual siempre reduce la distancia entre las luciérnagas, y consiste simplemente en la duplicación parcial de la luciérnaga en movimiento a la luciérnaga destino.

La distancia se calcula como la suma de las diferencias absolutas de las variables (distancia Manhattan) como se define en [7].

[1] La aplicación con la solución implementada puede ser accedida desde http://jerosode.github.io/tpi-ia/

Esta función representada en Prog. Code 1 consiste en la iteración sobre todos los elementos de ambas luciérnagas (líneas 4 a 13), su comparación y duplicación teniendo en cuenta la restricción de unicidad de los valores de las variables (líneas 5 a 9). En cada ciclo, se controla si se reduce la distancia (líneas 10 a 12), caso contrario se deshace la última duplicación realizada (línea 11).

Cuando las distancias son muy pequeñas, es probable que las luciérnagas no se acerquen debido a que el máximo paso definido es superado. Por lo tanto, se fuerza el acercamiento aumentando en cada bucle (líneas 3 a 15) el máximo paso en 1 (línea 14), hasta que ambas luciérnagas reduzcan su distancia.

Prog. Code 1. Representación en pseudocódigo de la función acercar

```
1. luc1 = {xi}; luc2= {yi}
2. Función acercar(luc1,luc2,maxpaso)
3.    Repetir
4.      Para cada xi en luc1
5.        Si xi==yj
6.          yj=yi; yi=xi
7.        Sino
8.          yi=xi
9.        FinSi
10.         Si ((nuevaDist < (distIni - maxpaso) o
              (nuevaDist >= distAnt))
11.         deshacerUltimoCambio()
12.       FinSi
13        FinPara
14.     maxpaso=maxpaso+1
15.   Hasta que distIni-distFin>0
16.   FinFuncion
```

3.4 Vector de aleatoriedad

La componente aleatoria aplicada al movimiento de las luciérnagas es implementada mediante la función *random(luciérnaga,randomness)*. Consiste en la selección de dos números aleatorios entre 0 y 9, la identificación de las variables que los tengan asignados, y el posterior intercambio de los valores entre las mismas. Este proceso implica la generación de otra configuración distinta a la anterior y se repite un numero aleatorio de veces, de 1 al valor definido por el parámetro general *randomness*, por lo cual, cuanto mayor sea este valor, más desemejante va a ser la luciérnaga resultante de la original.

3.5 Algoritmo Firefly

La función *FA()* descripta en Prog Code 2 es la columna vertebral del algoritmo, contiene en sí misma el comportamiento de las luciérnagas descripto por el marco teórico.
En primer lugar, se genera una población aleatoria de luciérnagas de tamaño fijo igual al parámetro general *initialPopulation*.

La función *FA()* consiste en la comparación iterativa (línea 6) de todas las luciérnagas pertenecientes a la población, la aproximación a aquellas que tengan mayor brillo/menor error (línea 7) en una proporción de la distancia entre ambos definida por el parámetro general *attractiveness*, y la aplicación de un vector de aleatoriedad a las luciérnagas en movimiento (línea 8).

En la línea 10 se verifica si dos luciérnagas tienen la misma configuración en el mismo ciclo, en ese caso se aplica el vector de aleatoriedad a una de las luciérnagas.

Esta función se repite hasta que se encuentre la solución o la cantidad de ciclos iguale al parámetro general *MAX_GEN*

Prog. Code 2. Representación en pseudocódigo de la función FA

```
1. lucs = generarLucs(initialPopulation)
2. Función FA(lucs)
3.    Mientras (no_solución_optima y ciclo < MAX_GEN)
4.        Para (lucs[i] i:1..N)
5.          Para (lucs[j] j:1..N)
6.            Si (error(lucs[i]) < error(lucs[j]))
7.              acercar(lucs[j],lucs[i],
                    dist(lucs[i],lucs[j])*attractiveness)
8.              random(lucs[j],randomness)
9.            Sino Si(i != j y equals(lucs[i],lucs[j]))
10.           random(lucs[j],randomness)
11.         FinSi
12.       FinPara
13.     FinPara
14.     ciclo=ciclo+1
15.   FinMientras
16. FinFuncion
```

Máximos locales.

Para evitar el estancamiento del algoritmo en máximos locales se definió un parámetro general *mpml* que define la cantidad máxima de ciclos durante los cuales una luciérnaga puede permanecer con el mismo brillo. Una vez superado este límite, se aplica a la misma la función *random()*.

4 Simulaciones y resultados

A continuación, se presentan los resultados de simulaciones realizadas a la implementación, que permiten la evaluación del comportamiento del mismo cuando varían los parámetros generales: *attractiveness*, *randomness*, *initialPopulation*, *mpml*. Se pretende como resultado encontrar los valores de los parámetros anteriores que optimice el rendimiento del algoritmo, lo cual se define con el **promedio de ciclos** en encontrar la solución, el **tiempo promedio** en encontrar la solución y el **porcentaje** de ejecuciones en el cual se encontró la solución.

Para la obtención de estos resultados se ejecuta[2] el algoritmo 100 veces con el problema expresado en Fig.1 (suma) variando un parámetro a la vez dejando a los demás constantes.

- *MAX_GEN:* Cantidad máxima de ciclos que se ejecutara el algoritmo antes de detenerse. Valor por defecto: 1000
- *attractivenes:* Coeficiente que determina el grado de movimiento entre las luciérnagas. Valor por defecto: 0.5
- *randomness*: Vector de movimiento aleatorio de las luciérnagas. Valor por defecto:1
- *initialPopulation*: Cantidad de luciérnagas que se van a utilizar para la ejecución del algoritmo. Valor por defecto: 10
- *mpml*: Cantidad máxima de ciclos que una luciérnaga puede permanecer con el mismo brillo. Valor por defecto: 4

4.1 Variación de *initialPopulation*

En Table 1 se observan los resultados de la ejecución del algoritmo mientras varía el parámetro general *initialPopulation*.

Table 1. Resultados simulación del Algoritmo variando *initialPopulation*

initialPopulation	**Promedio Ciclos**	**Porcentaje**	**Tiempo Promedio [ms]**
5	399,9	90	85,18
6	238	97	95,96
7	168,82	100	97,75
10	71,85	100	84,89
15	37,9	100	102,02
20	28,28	100	134,92
25	14,73	100	125,58
30	11,03	100	133,08
35	8,16	100	133,67
40	7,19	100	152,98

[2] Ejecutado en un procesador Intel Core i7-4770 3.40GHz, con 8GB de RAM y en un navegador Chrome Versión 51.0.2704.84 m.

De Table 1, se concluye que el promedio de ciclos en los que encuentra la solución disminuye a medida que aumenta *initialPopulation*. Además, a partir de *initialPopulation* = 7, se encuentra una solución óptima el 100% de las veces.

Esto es consecuencia de que, a medida que aumentan la cantidad de luciérnagas, la cantidad de comparaciones entre luciérnagas y los movimientos que estas realizan aumenta exponencialmente dentro de cada ciclo, permitiendo a las mismas abarcar un espacio de soluciones mucho mayor y encontrar la solución en la menor cantidad de ciclos.

La contrapartida de esta apreciación es que, si bien la cantidad de ciclos disminuyen, la cantidad de comparaciones aumentan, lo que exige mayor tiempo de procesamiento, lo que se puede apreciar observando el tiempo promedio de la Table 1.

4.2 Variación de *attractiveness*

En Table 2 se observan los resultados de la ejecución del algoritmo mientras varía el parámetro general *attractiveness*.

Table 2. Resultados simulación del Algoritmo variando *attractiveness*

attractiveness	Promedio Ciclos	Porcentaje	Tiempo Promedio
0,1	625,67	65	265,55
0,2	384,32	93	247,17
0,3	160,3	100	134,44
0,4	116,34	100	88,87
0,5	128,63	100	98,22
0,6	121,23	100	92,61
0,7	127,3	100	96,99
0,8	99,4	100	72,78
0,9	105,34	100	73,52

En Table 2, se puede observar que a partir de un valor de attractiveness=0.3, se encuentra una solución óptima el 100% de las veces.

A medida que la atractividad aumenta, la eficiencia del algoritmo también lo hace, debido a que las luciérnagas se acercan con mayor rapidez unas a otras aumentando la convergencia hacia la solución de mayor brillo, que en última instancia es la solución óptima.

4.3 Variación de *randomness*

En Table 3 se observan los resultados de la ejecución del algoritmo mientras varía el parámetro general randomness.

De Table 3 se deduce que a medida que *randomness* aumenta, la eficacia y eficiencia del algoritmo disminuye, aumentando el promedio de ciclos en los que encuentra la solución y disminuyendo el porcentaje de veces en las que se encuentra.

Cuanto mayor sea *randomness*, el movimiento aleatorio será mayor y más se diferenciará de la luciérnaga a la que intenta acercarse, convirtiéndose cada vez más en un buscador aleatorio más que un algoritmo de enjambre. Cuando *randomness* es pequeño, las luciérnagas permanecen cercanas unas a las otras, analizando el espacio de soluciones alrededor de la luciérnaga de mayor brillo.

Table 3. Resultados simulación del Algoritmo variando *randomness*

randomness	**Promedio Ciclos**	**Porcentaje**	**Tiempo Promedio [ms]**
0	-	0	-
1	68,53	100	79,04
2	116,68	100	136,03
3	196,13	99	225,79
4	231,61	95	256,8
5	285,59	93	303,77

Cuando *randomness* es 0, implica la ausencia de aleatoriedad, resultando simplemente en la aproximación de la población de luciérnagas a aquella que tenga mayor brillo. Al final, todas las luciérnagas quedan iguales a la de mayor brillo. Por lo tanto, es imperativo el uso de un valor de *randomness* mayor a 0.

4.4 Variación de *mpml*

Luego de la ejecución del algoritmo variando el parámetro general *mpml*, se concluye que el mismo no influye en su rendimiento. De todas maneras, se considera imprescindible su existencia para evitar el estancamiento en máximos locales, que obstaculicen encontrar la solución óptima.

5 Conclusión

En este trabajo se presenta la implementación de un solucionador de problemas criptoaritméticos utilizando el algoritmo de enjambre Firefly.

Teniendo en cuenta los resultados durante la variación del parámetro general *randomness* se concluye que el algoritmo implementado depende en gran medida de la aleatoriedad de sus variables, factor sin el cual no funciona adecuadamente. La ventaja que brinda este algoritmo es que la aleatoriedad está controlada mediante la definición de *randomness*, permitiendo focalizarla en alguna parte del espacio de soluciones que sea de interés. Esto es lo que lo diferencia de un buscador aleatorio de soluciones.

Este algoritmo se adapta muy bien a funciones no continuas y discretas. Esta es una de las grandes dificultades que tiene que enfrentar con este tipo de problemas, ya que las soluciones óptimas pueden encontrarse aisladas, entorpeciendo así su búsqueda.

La implementación descripta en este trabajo se ajusta al comportamiento genérico de los algoritmos de enjambre Firefly, sin embargo, en estudios futuros, este podría ser optimizado agregando heurísticas propias de los problemas criptoaritméticos para el manejo individual de cada luciérnaga, aplicando reglas aritméticas que comprueben la validez parcial de los resultados.

Referencias

1. H. E. Dudeney: Strand Magazine vol. 68, pp. 97 y 214 (1924)
2. R. Abbasian and M. Mazloom: Solving Cryptarithmetic Problems Using Parallel Genetic Algorithm. In: Computer and Electrical Engineering, 2009. ICCEE '09. Second International Conference, pp. 308-312. Dubai (2009)
3. J. F. Fontanari: Solving a cryptarithmetic problem using a social learning heuristic. In: Comutational Intelligence, Cognitive Algorithms, Mind, and Brain (CCMB), IEEE Symposium pp. 65-70. Orlando, FL (2014)
4. J. Kennedy and R. Eberhart: Particle swarm optimization. IEEE International Conference, pp. 1942-1948 vol.4. Perth, WA (1995)
5. Fateen, Seif-Eddeen K., and Adrián Bonilla-Petriciolet: Intelligent firefly algorithm for global optimization. Cuckoo Search and Firefly Algorithm. In: Springer International Publishing pp. 315-330 (2014)
6. Yang, Xin-She: Firefly algorithm, stochastic test functions and design optimization. In: International Journal of Bio-Inspired Computation 2.2 pp. 78-84 (2010)
7. Manhattan Distance, `https://xlinux.nist.gov/dads//HTML/manhattanDistance.html`