Resolución de problemas criptoaritméticos utilizando el Algoritmo de Enjambre Firefly

Jerónimo Sodero, Matías Wajnman, y Nelson Yaccuzzi

Inteligencia Artificial, Universidad Tecnológica Nacional, Resistencia, Argentina

{jerosode,matiasarw,nelsonyaccuzzi}@gmail.com

Abstract. El presente trabajo tiene como finalidad mostrar la implementación de un algoritmo de enjambre, en este caso el algoritmo Firefly, para la resolución de problemas criptoaritméticos, haciendo un análisis de las simulaciones realizadas qué dejen en claro su comportamiento.

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

1 Introducción

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

La ecuación es típicamente basada en una operación básica aritmética, como la suma, resta, multiplicación o división. En el presente trabajo se implementarán la operación suma y resta únicamente.

El ejemplo clásico, el cual se tomará de referencia, fue publicado en Julio de 1924 por Henry Dudeney, 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. Un buen problema solamente tiene una solución y las letras forman una frase como en el ejemplo anterior.

Este problema fue previamente resuelto utilizando Algoritmos Genéticos [2] y optimizándolo con paralelismo; también, mediante Aprendizaje Social Heurístico [3], con

uso de N agentes trabajando colaborativamente. Estas demuestran ser implementaciones eficientes a comparación de la presentada en este trabajo debido a que, en este último, los procesamientos y comparaciones son realizados de manera secuencial en un solo hilo.

El trabajo está dividido 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 algoritmo. Luego, en la sección 4 se analiza el algoritmo cuando varían estos parámetros mediante pruebas y sus posteriores resultados. La sección 5 detalla aspectos tecnológicos de la implementación y finalmente, la sección 6 expresa las conclusiones finales.

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 en la que se basa 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, dado su gran rendimiento y simplicidad.

3 Solución Implementada

3.1 Representación de las luciérnagas

Cada luciérnaga representa una solución posible, en este caso, representa una posible configuración de valores letra:valor, en el cual cada valor solo puede tomar números enteros de 0 a 9 y puede ser asignado a solo una letra. Esto restringe la cantidad de letras o variables en el problema a un máximo de 10.

Cada luciérnaga fue representada programáticamente como una variable del tipo diccionario, en donde las letras únicas cumplen la función de índices (no varían en el transcurso de la resolución), y estos tienen asignados valores (varían en el transcurso de la resolución). Ejemplo:

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

Prog. Code 1. Ejemplo de la representación de una luciérnaga

3.2 Brillo de las luciérnagas

El brillo de cada luciérnaga fue representado mediante el resultado de la función error, que, en sentido contrario a lo definido por el marco teórico, es mayor cuando menor sea el resultado de la función.

La función error consiste en la comparación de los resultados en ambos miembros de la ecuación presentada como problema:

$$Operador1 + Operador2 = Resultado \quad (1)$$

$$Operador1 + Operador2 - Resultado = 0 \quad (2)$$

$$Operador1 - Operador2 = Resultado \quad (3)$$

$$Operador1 - Operador2 - Resultado = 0 \quad (4)$$

A partir de lo anterior podemos deducir que cuanto más cercano a 0 sea el resultado del primer miembro de la ecuación 2 o 4, más correcta será la solución. En el caso de que la expresión se cumpla, la configuración que la hace posible es una solución óptima.

Para que los valores de la función sean más manejables se toma simplemente la suma de los dígitos del resultado del primer miembro de la ecuación 2 o 4 como resultado de la función error. Por tanto:

$$Error = SumaDigitos(|Resultado - (Operador1 + Operador2)|) \quad (5)$$

$$Error = SumaDigitos(|Resultado - (Operador1 - Operador2)|) \quad (6)$$

En **Prog. Code 1** el error se calcula de la siguiente manera:

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

$$Error = 21$$

3.3 Distancia entre luciérnagas

Para que las luciérnagas sepan cuanto moverse y hacia donde moverse es fundamental conocer la distancia existente entre las mismas.

Teniendo en cuenta que el problema es discreto en cuanto a los valores de las variables, se vio conveniente utilizar la distancia Manhattan como cálculo de la distancia.

$$DistanciaManhattan(x, y) = \sum(|xi - yi|) \quad (7)$$

3.4 Acercamiento

La función implementada siempre acerca mientras que la distancia entre las mismas sea mayor a 0, y consiste simplemente en la duplicación de la luciérnaga en movimiento a la luciérnaga destino.

```
1. luc1 = {yi}; luc2= {xi}
2. Función acercamiento (luc1,luc2,maxpaso)
3.    Repetir
4.      Para cada yi en luc1
```

```
5.        Si (yi=A) y (xj=A)
6.          xj=xi; xi=yi
7.        Sino
8.          xi=yi
9.        FinSi
10.         Si ((nuevaDist > distInicial - maxpaso) o
(nuevaDist < distAnterior))
11.         deshacerUltimoCambio()
12.       FinSi
        FinPara
13.     maxpaso=maxpaso+1
14.   Hasta que distInicial-distFinal>0
15.   FinFuncion
```

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

La función consiste en la iteración sobre todos los elementos x_i e y_i de ambas luciérnagas, su comparación e igualación de los valores de las variables. En cada iteración se controla la nueva distancia resultante y si la misma no sobrepasa el límite deseado y no se aleja de la luciérnaga destino, se establece la misma como distancia de referencia para la iteración siguiente. En caso contrario se eliminan las acciones realizadas en dicha iteración.

En el caso de que las distancias sean muy pequeñas, es muy probable que las luciérnagas no se acerquen debido a que el máximo paso definido siempre se supera, esto genera bucles de acercamiento en las luciérnagas que entorpecen el funcionamiento del algoritmo. Para solucionarlo, se fuerza el acercamiento aumentando en cada bucle el máximo paso en 1, cuando la aproximación de las luciérnagas sea 0.

El resultado de este método si no se aplica los vectores de aleatoriedad es la rápida convergencia de todas las luciérnagas de la población a la misma configuración, la cual es la configuración de aquella luciérnaga que tenga mayor brillo.

3.5 Vector de aleatoriedad

```
1.  Funcion random(luciernaga, randomness)
2.    iter= valor aleatorio entre 1..randomness
2.    Para i=0 hasta i<iter
3.      A= valor aleatorio entre 0..9
4.      B= valor aleatorio entre 0..9
5.      Intercambiar(xi=A,xj=B)
6.    FinPara
7.  FinFuncion
```

Prog. Code 3. Representación en pseudocódigo de la función random

El vector de aleatoriedad que es aplicado a cada luciérnaga en movimiento es implementado mediante la función random(luciérnaga, randomness)

La función consiste en la selección de dos valores aleatorios entre 0 y 9, y el intercambio de los valores de las variables que los tengan asignado. 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 randomness, por lo cual, cuanto mayor sea este valor, más diferente va a ser la luciérnaga resultante de la original.

3.6 Algoritmo Firefly

La función FA es la columna vertebral, contiene en sí misma el comportamiento de las luciérnagas descripto por el marco teórico.

```
1. luciernagas = generarLuciernagas(initialPopulation)
2. Funcion FA(luciernagas)
3.     Mientras (no_solución_optima y ciclo<MAX_GEN)
4.       Para (luciernagas[i] i:1..N)
5.            //N= cantidad de luciernagas
6.         Para (luciernagas[j] j:1..N)
7.           Si (error(luciernagas[i])<error(luciernagas[j]))
8.             acercamiento(luciernagas[j],luciernagas[i], distancia-
Manhattan(luciernaga[i],luciernaga[j])*attractiveness)
9.             random(luciernagas,randomness, j)
10.            Sino Si(i != j y equals(luciernagas[i],luciernagas[j]))
11.            random(luciérnagas[j],randomness)
12.          FinSi
13.        FinPara
14.      FinPara
15.      ciclo=ciclo+1
16.    FinMientras
17.  FinFuncion
```

Prog. Code 4. Representación en pseudocódigo de la función FA

Previo a comenzar esta función, se generan una población aleatoria de luciérnagas de tamaño fijo igual a initialPopulation, que después se pasan como parámetro a la función FA.

Esta última consiste en la comparación iterativa de todas las luciérnagas pertenecientes a la población, la aproximación a aquellas que tengan mayor brillo (a menor error), y la aplicación de un vector de aleatoriedad a las luciérnagas en movimiento.

Teniendo en cuenta que la función acercar consiste en la misma duplicación de las luciérnagas, encontramos probable que varias luciérnagas terminen teniendo la misma configuración en el mismo ciclo, lo cual entorpece su funcionamiento ya que dentro del mismo ciclo se realiza el mismo proceso varias veces. Para solucionar este problema, en la línea 10 del Prog. Code 4 se verifica que sus configuraciones sean distintas, en caso contrario se aplica el vector de aleatoriedad a una de las luciérnagas.

Máximos locales.
Durante la ejecución del algoritmo fue un comportamiento observable el de luciérnagas que permanecían con un brillo alto atrayendo a todas las demás durante muchos ciclos, las cuales correspondían a un máximo local dentro del espacio de soluciones. Este comportamiento estancaba al mismo ya que las luciérnagas que se movían alrededor del mismo nunca encontraban la solución óptima.

Para evitar este comportamiento se definió una cantidad máxima de ciclos durante los cuales una luciérnaga puede permanecer con el mismo brillo, una vez superada este límite, se aplica a la misma el vector de aleatoriedad.

3.7 Parámetros

Las variables que se tienen que definir antes de la resolución de cada problema son las siguientes:
MAX_GEN: es la cantidad máxima de ciclos que se deben ejecutar, una vez alcanzado, el proceso termina, haya o no encontrado una solución optima
attractiveness: indica la proporción en la cual la luciérnaga en movimiento va a tratar de acercarse a la de mayor brillo. Define el máximo paso a acercarse multiplicando a la distancia entre las luciérnagas. Puede tomar valores entre 0 y 1.
randomness: indica el grado de aleatoriedad que se aplicará a la luciérnaga. Cuanto más grande el valor, más distinta será la luciérnaga resultante de la original.
initialPopulation: indica la cantidad de luciérnagas que interactuarán durante la ejecución.
mpml: representa la cantidad de ciclos durante los cuales una luciérnaga puede permanecer con el mismo brillo.

4 Simulaciones y resultados

A continuación, se presentarán los resultados de simulaciones realizadas a la implementación que permitirán la evaluación del comportamiento de algoritmo cuando varían los parámetros de entrada, como ser: Atractividad, Aleatoriedad, Cantidad de Luciérnagas, Máxima Permanencia en máximo local.
Como resultado vamos a obtener:

Promedio de ciclos: Cantidad promedio de ciclos que tarda el algoritmo en encontrar la solución, sin tener en cuenta aquellas ocasiones en las cuales supera la cantidad máxima de ciclos.

Porcentaje: Porcentaje de veces que se encontró solución antes de que se supere la cantidad máxima de ciclos.

Tiempo promedio: Tiempo promedio expresado en milisegundos que tarda el algoritmo en encontrar la solución, sin tener en cuenta aquellas ocasiones en las cuales supera la cantidad máxima de ciclos.

Para la obtención de estos resultados se ejecuta el algoritmo 100 veces con el problema expresado en Fig.1 variando un parámetro a la vez.

Los valores estándar para las pruebas son:

- Atractividad = 0.5
- Aleatoriedad = 1
- Cantidad de Luciérnagas = 10
- Máxima Permanencia en máximo local = 4

Para calcular el tiempo promedio hay que tener en cuenta que las pruebas realizadas fueron realizadas en un procesador Intel Core i7-4770 3.40GHz, con 8GB de RAM y ejecutándolo en un navegador Chrome Versión 51.0.2704.84 m.

4.1 Cantidad de luciérnagas N

Table 1. Resultados simulación del Algoritmo variando la cantidad de luciérnagas

Cantidad de luciérnagas	Promedio Ciclos	Porcentaje	Tiempo Promedio
5	399,9	90	85,18
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

Como se puede observar, el promedio de ciclos en los que encuentra la solución disminuye a medida que aumenta N. Además, a partir de N= 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, que se puede apreciar en la última columna de la Tabla 1.

4.2 Atractividad A

Table 2. Resultados simulación del Algoritmo variando la atractividad

Atractividad	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

A partir de A= 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, ya que las luciérnagas se acercan con mayor rapidez aumentando la convergencia a la solución óptima.

4.3 Aleatoriedad R

Table 3. Resultados simulación del Algoritmo variando la aleatoriedad

Aleatoriedad	Promedio Ciclos	Porcentaje	Tiempo Promedio
0	0	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
6	307,04	89	316,92

A medida que R 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 R, 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 R es pequeño, las luciérnagas permanecen cercanas unas a las otras, analizando el espacio de soluciones alrededor de la luciérnaga de mayor brillo.

Cuando R es 0, significa la ausencia de aleatoriedad, lo que implica simplemente la aproximación de la población de luciérnagas a aquella que tenga mayor brillo, resultando en la clonación de todas las luciérnagas, lo que estanca la ejecución del algoritmo no encontrando solución.

4.4 Máxima permanencia en máximo local.

Luego de realizadas varias pruebas de rendimiento con variados valores de la variable, podemos concluir que no influye en su rendimiento. Pero consideramos imperativo la asignación de un valor, ya que, si este no existiera, el algoritmo podría quedar estancado en algún máximo local.

5 Implementación Aplicación

Para la implementación de este algoritmo y para la realización de todas las pruebas se eligieron las siguientes tecnologías y librerías.

- Javascript
- Jquery [7]
- Materialize [8]
- Chance.js [9]

La misma puede ser accedida desde el siguiente vinculo:
http://jerosode.github.io/tpi-ia/

6 Conclusión

La primera impresión que se tuvo al aplicar este tipo de Algoritmos de enjambre, es la gran dependencia de la aleatoriedad, sin la cual este no funcionaría. Está presente desde sus comienzos mediante la generación de luciérnagas hasta en los movimientos que realizan las mismas, y es la responsable de la variabilidad de la cantidad de ciclos necesarios para resolver un problema. La ventaja que brinda este algoritmo es que la aleatoriedad está controlada mediante la definición de los parámetros, 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.

La segunda impresión es la adaptabilidad del algoritmo 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 la búsqueda de soluciones.

Otra de las grandes dificultades que se enfrenta, son las restricciones que se imponen a las variables, cuyos valores siempre tienen que ser desiguales. Además, pueden afectar de distinta manera a la función a optimizar de acuerdo a la estructura que posea el problema. Sin embargo, este algoritmo logra cumplir con las mismas y encontrar la solución óptima.

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.

Concluimos en que este tipo de algoritmos posee un amplio espectro de aplicación, demostrando en este trabajo la misma a funciones restringidas, discretas y no continuas, sin embargo, esto no quiere decir que sea el más apropiado, sino una propuesta que puede ayudar a optimizar cualquier tipo de función.

Referencias

1. H. E. Dudeney, in Strand Magazine vol. 68 (July 1924), pp. 97 and 214.
2. R. Abbasian and M. Mazloom, "Solving Cryptarithmetic Problems Using Parallel Genetic Algorithm," *Computer and Electrical Engineering, 2009. ICCEE '09. Second International Conference on*, Dubai, 2009, pp. 308-312.
3. J. F. Fontanari, "Solving a cryptarithmetic problem using a social learning heuristic," *Computational Intelligence, Cognitive Algorithms, Mind, and Brain (CCMB), 2014 IEEE Symposium on*, Orlando, FL, 2014, pp. 65-70.
4. J. Kennedy and R. Eberhart, "Particle swarm optimization," *Neural Networks, 1995. Proceedings., IEEE International Conference on*, Perth, WA, 1995, pp. 1942-1948 vol.4.
5. Fateen, Seif-Eddeen K., and Adrián Bonilla-Petriciolet. "Intelligent firefly algorithm for global optimization." *Cuckoo Search and Firefly Algorithm*. Springer International Publishing, 2014. 315-330.
6. Yang, Xin-She. "Firefly algorithm, stochastic test functions and design optimisation." International Journal of Bio-Inspired Computation 2.2 (2010): 78-84.
7. Jquery Javascript library, https://jquery.com/
8. Materialize Library, http://materializecss.com/
9. Chance.js library, http://chancejs.com/