Laboratorio Nacional de Informática Avanzada
Centro de Enseñanza LANIA

Optimización de Cadenas de Adición en Criptografía utilizando Programación Evolutiva

TESIS

Que presenta:

Saúl Domínguez Isidro

Para obtener el grado de:

Maestro en Computación Aplicada

Director de Tesis:

Dr. Efrén Mezura Montes

Xalapa, Veracruz, México Noviembre 2011

Agradecimientos

- A DIOS.
- A mis padres y seres queridos por el apoyo brindado durante la realización de la maestría.
- Al Dr. Efrén Mezura Montes por su asesoria y tiempo invertido.
- A todos mis profesores y compañeros por su apoyo y amistad durante mis estudios.
- Al CONACyT por la beca otorgada para la realización de la maestría.
- Se agradece el apoyo de CONACyT mediante el proyecto No. 79809 para asistir al congreso GECCO 2011 a presentar una publicación relacionada con el presente trabajo de tesis.

Resumen

En el presente trabajo de Tesis se propone el uso de un algoritmo de programación evolutiva para la generación de cadenas de adición de longitud mínima, las cuales son usadas en el proceso de cifrado y descifrado de datos en los criptosistemas asimétricos o de llave pública. La generación de cadenas de adición de longitud mínima es considerado un problema NP-Hard. La presente investigación está compuesta por cinco aspectos fundamentales. (1) Se explican los distintos tipos de cadenas de adición así como su función en la Criptografía, (2) se mencionan las principales meta-heurísticas que tienen la función de resolver problemas de optimización, (3) se abordan los principales métodos que resuelven el problema de generación de cadenas de adición de longitud mínima, (4) se explica la adaptación del algoritmo de programación evolutiva para encontrar cadenas de adición de longitud mínima y (5) se describen los experimentos realizados para medir el desempeño del algoritmo.

Indice

Contenido II

Indice de figuras III

Indice de tablas IV

Indice de algoritmos V

1. Introducción 1
- 1.1. Planteamiento del problema . . . 1
- 1.2. Objetivos . . . 5
  - 1.2.1. Objetivo General . . . 5
  - 1.2.2. Objetivos Específicos . . . 5
- 1.3. Hipótesis . . . 5
- 1.4. Justificación . . . 5
- 1.5. Organización del Documento . . . 6

2. Cadenas de Adición 7
- 2.1. Clases de Cadenas . . . 7
  - 2.1.1. Cadenas de Adición . . . 7
  - 2.1.2. Cadenas de Adición Euclidianas . . . 8
  - 2.1.3. Secuencias de Adición . . . 9
  - 2.1.4. Cadenas de Adición Vectoriales . . . 9
  - 2.1.5. Cadenas de Adición-Substracción . . . 10
- 2.2. Aplicaciones de las Cadenas de Adición . . . 10
  - 2.2.1. Aritmética modular . . . 11
  - 2.2.2. Cadenas de adición en la exponenciación modular . . . 12

3. Heurísticas Inspiradas en la naturaleza 15
- 3.1. Algoritmos Evolutivos . . . 16
  - 3.1.1. Algoritmos genéticos . . . 17
  - 3.1.2. Programación genética . . . 18
  - 3.1.3. Estrategias evolutivas . . . 18
  - 3.1.4. Programación evolutiva . . . 19

3.1.5. Evolución diferencial 20
3.2. Algoritmos de Inteligencia Colectiva 20
3.2.1. Algoritmo basado en cúmulos de partículas 21
3.2.2. Algoritmo basado en colonia de hormigas 22
3.2.3. Algoritmo basado en enjambre de abejas 23
3.2.4. Algoritmo basado en bacterias 23
3.3. Sistema Inmune Artificial 24

4. Trabajo Relacionado 26
4.1. Métodos Determinísticos 26
4.1.1. Método binario 26
4.1.2. Método factor 27
4.1.3. Método de ventana 28
4.2. Métodos Estocásticos 33
4.2.1. Algoritmos de Inteligencia Colectiva 33
4.2.2. Algoritmos Evolutivos 34
4.2.3. Sistema inmune artificial para la generación de cadenas de adición cortas 36

5. Algoritmo Propuesto 37
5.1. Elementos fundamentales en programación evolutiva 37
5.2. Representación de la población y función de aptitud 38
5.2.1. Población inicial 39
5.3. Operador de variación 39
5.4. Mecanismo de reemplazo 40
5.5. EP Modificado para exponentes pequeños 42
5.6. Modificación para resolver problemas con exponentes grandes . . 44

6. Experimentos y Resultados 47
6.1. Experimento 1: cadenas de adición acumuladas 48
6.2. Experimento 2: exponentes difíciles 50
6.3. Experimento 3: exponentes diversos 51
6.4. Experimento 4: exponentes grandes 51
6.5. Experimento 5: cadenas de adición Euclidianas 52

7. Conclusiones y Trabajo Futuro 53

8. Anexos 55
8.1. Tablas del experimento 2 55
8.2. Tablas del experimento 3 58
8.3. Tablas del experimento 5 60

Referencias 62

Indice de figuras

1.1. Algoritmo de llave privada 2
1.2. Algoritmo de llave pública 2

2.1. Digrafo de una cadena de adición donde $e = 49$ 8
2.2. Digrafo de una cadena de adición euclidiana donde $e = 49$ 9
2.3. Digrafo de una secuencia de adición para el conjunto $N = \{9, 23, 31, 54\}$ (Nodos oscuros en el grafo) 9
2.4. Flujograma del algoritmo RSA 13

3.1. Tren bala japonés Shinkansen 16
3.2. Modelo de un Algoritmo Evolutivo 17

4.1. Esquema del método binario para $e = 77$ 27
4.2. Árbol de factores generado con el Algoritmo 12 para $e = 55$ . . . 30
4.3. Generador de ventanas para SWM 31
4.4. Método SWM donde $e = 77$ 32

5.1. Diagrama de flujo de EP 38
5.2. Representación de la población 38
5.3. Mutaciones en EP 41
5.4. Diagrama de Flujo del EP_SWM 46

Indice de tablas

6.1. Evaluaciones realizadas por cada meta-heurística del estado del arte . . . 48
6.2. Resultados estadísticos del primer experimento con EP. 49
6.3. Resultados obtenidos de cadenas de adición acumuladas y comparación contra AIS [8], REPLS-GA [31] y PSO [24] 50
6.4. Resultados obtenidos por EP con 240000 evaluaciones para cadenas de adición acumuladas y comparación con AIS [8], GA [31] y PSO [24]. En negritas se remarcan los resultados en el óptimo. En itálicas se indican los mejores resultados sin llegar al óptimo. 50
6.5. Resultados de EP_SWM para exponentes grandes y comparación contra SWM [18] y AIS_SWM [8]. En negritas se marcan los mejores resultados. En itálicas se muestran resultados similares a los previamente reportados. 52

8.1. Resultados del segundo experimento: mejores resultados obtenidos por REPLS-GA[31] y EP para exponentes "difíciles" (1 de 2) . . 56
8.2. Resultados del segundo experimento: mejores resultados obtenidos por REPLS-GA[31] y EP para exponentes "difíciles" (2 de 2) . . 57
8.3. Resultados del tercer experimento: mejores resultados obtenidos por AIS[8], PSO [24] y EP en un conjunto de "diversos" exponentes(1 de 2) 58
8.4. Resultados del tercer experimento: mejores resultados obtenidos por AIS[8], PSO [24] y EP en un conjunto de "diversos" exponentes(2 de 2) 59
8.5. Cadena de adición Euclidiana usando el algoritmo EP en un conjunto de "diversos" exponentes (1 de 2) 60
8.6. Cadena de adición Euclidiana usando el algoritmo EP en un conjunto de "diversos" exponentes (2 de 2) 61

Indice de algoritmos

1. Exponenciación Modular 12
2. Algoritmo genético simple 18
3. Algoritmo de estrategias evolutivas 19
4. Algoritmo de programación evolutiva 19
5. Algoritmo de evolución diferencial 20
6. Algoritmo de optimización mediante cúmulo de partículas 21
7. Algoritmo de optimización mediante colonia de hormigas 22
8. Algoritmo basado en enjambre de abejas 23
9. Algoritmo de optimización basado en bacterias modificado 24
10. Algoritmo de un sistema inmune artificial 25
11. Algoritmo del método binario 27
12. Construccion de árbol 28
13. Calcular cadena 29
14. Algoritmo del método de ventana 30
15. Algoritmo del método de deslizamiento de ventana 32
16. Generador de secuencias de adición [8] 32
17. PSO 34
18. Algoritmo Genético 35
19. **Cadena_Factible**(e) 39
20. **Completar**(U, k, e) 40
21. **Mutación**(U) 41
22. **Reemplazo(Conjunto)** 42
23. EP_Modificado(e) 43
24. EP_Generador_Secuencias_Adición 45

Capítulo 1

Introducción

En este capítulo se presenta el problema a resolver y su impacto. Además se muestran el objetivo principal de esta tesis y los objetivos específicos. También se describe la hipótesis planteada, y por último se justifica el desarrollo de la presente investigación.

1.1. Planteamiento del problema

Con el desarrollo de los sistemas de información y comunicación (SIC), la información ha ido adquiriendo paulatinamente mucho valor, debido a que ésta se genera, se comercializa y se consume por muchos sectores de la sociedad. Debido a ésto, se originan diversos movimientos económicos e intereses [16].

Existen diversos tipos de SIC´s que manejan informacion sensible o de suma importancia, la cual controla o sustenta intereses de sus usuarios. Por ejemplo, en un sistema bancario se maneja la información financiera de personas, tanto físicas como morales, si esta información llegara a ser modificada por terceros, tanto el banco como sus clientes serían afectados. Este tipos de SIC´s son susceptibles a ser atacados por terceros interesados en hurtar la información. Es por ello que surge la seguridad informática, la cual se encarga de proteger la información perteneciente a los usuarios, propensa a posibles amenazas, riesgos y vulnerabilidades. Una de las áreas involucradas en la seguridad para cumplir con su objetivo, es la Criptografía.

La Criptografía es la ciencia que consiste en transformar un mensaje inteligible en otro que no lo es, esto mediante claves que sólo el emisor y el destinatario conocen, para después devolverlo a su forma original, sin que alguien que vea el mensaje cifrado, sea capaz de entenderlo [4].

Uno de los componentes de un sistema criptográfico lo conforman los algoritmos de cifrado. Éstos se dividen en dos tipos: algoritmos de llave privada

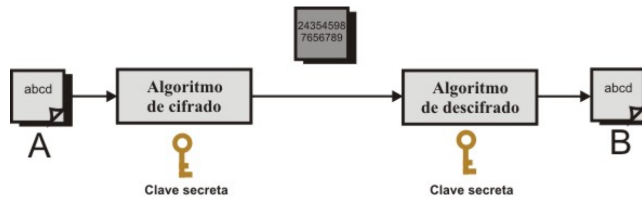

Fig. 1.1: Algoritmo de llave privada

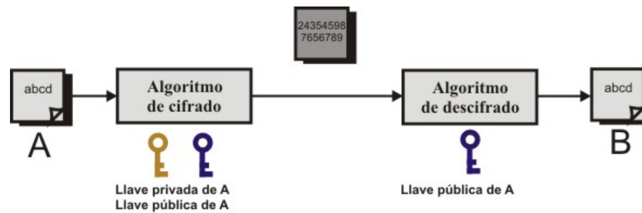

Fig. 1.2: Algoritmo de llave pública

(simétricos) y algoritmos de llave pública (asimétricos).

Los algoritmos simétricos o de llave privada, se caracterizan por la utilización de una sola clave, que sirve para cifrar y descifrar el mensaje. Por ejemplo, si el usuario A, le quiere enviar un mensaje al usuario B, para cifrar el mensaje, A utiliza su llave privada; y si B quiere descifrar el mensaje, necesita conocer la llave privada de A, véase la Figura 1.1.

Por su parte, los algoritmos asimétricos o de llave pública, tienen la particularidad de utilizar dos tipos de llave: privada y pública. Cada usuario necesita tener ambas. Así, si A le quiere enviar un mensaje a B, para cifrar el mensaje, A utiliza su llave privada y la llave pública de B; y si B quiere descifrar el mensaje, necesita su llave privada. Este mecanismo, hace que los algoritmos asimétricos sean más seguros que los simétricos, véase la Figura 1.2.

Existen diversos sistemas de cifrado de llave pública, tales como el algoritmo Rivest, Shamir y Adleman (RSA) [35], Diffie-Hellman [34], Digital Signature Algorithm (DSA) [37], ElGamal [12], entre otros. Este tipo de algoritmos utilizan la exponenciación modular para poder cifrar y descifrar datos. La exponenciación modular consiste en encontrar un entero positivo b que satisfaga la siguiente ecuación:

$$b \equiv a^e \ mod \ p$$

Donde:
a = entero [0,1,2, ..., p-1]

$e =$ entero arbitrario positivo
$p =$ primo grande o producto de primos grandes

Esta operación se realiza para cada dato (a) que se requiera cifrar o descifrar; ya que a es el mensaje cifrado y e es parte de la llave pública. a^e es el objeto de estudio, debido a que, si elevamos un número a a un exponente e, el número base (a) es multiplicado e veces. En resumen, entre mayor número de caracteres contenga la información que se quiera cifrar o descifrar más veces se tiene que realizar esta operación; lo cual conlleva a un mayor costo computacional y mayor tiempo en realizar las operaciones de cifrado y descifrado.

Existen dos formas de atacar el problema citado anteriormente. La primera consiste en aumentar la rapidez de cálculo, mediante técnicas de hardware, como lo es el cómputo distribuido (varias computadoras realizando el cálculo), o el uso de procesadores muy veloces; el problema de esta técnica es que su implementación puede ser costosa. La segunda forma de resolver dicho problema consiste en reducir la cantidad de multiplicaciones que se realizan para elevar un número dado a la e-nesima potencia. Para fines de esta investigación nos enfocaremos en realizar el cálculo mediante la segunda forma.

Para reducir el número de multiplicaciones en la exponenciación modular se pueden ocupar cadenas de adición, descritas con detalle en la Sección 2.1.1. Una cadena de adición es una secuencia de números donde el primer número es 1, el último número es el exponente e al que se quiere llegar y cada número en la cadena es la suma de dos números previos, no siempre distintos.

Por ejemplo, sea $e = 79$ se puede utilizar la siguiente cadena de adición,

$$U = \{1, 2, 4, 6, 10, 16, 26, 42, 68, 74, 78, 79\}$$

donde su longitud $length(U) = 11$, dicha cadena se esquematiza de la siguiente forma:
$a^1 \equiv a$
$a^2 \equiv a^1.a^1$
$a^4 \equiv a^2.a^2$
$a^6 \equiv a^3.a^3$
$a^{10} \equiv a^6.a^4$
$a^{16} \equiv a^{10}.a^6$
$a^{26} \equiv a^{16}.a^{10}$
$a^{42} \equiv a^{26}.a^{26}$
$a^{68} \equiv a^{42}.a^{26}$
$a^{74} \equiv a^{68}.a^6$
$a^{78} \equiv a^{74}.a^4$
$a^{79} \equiv a^{78}.a^1$

Sin embargo para el mismo exponente se puede usar una cadena de adición

$$U = \{1, 2, 3, 6, 12, 13, 26, 52, 78, 79\}$$

de longitud $length(U) = 9$:
$a^1 \equiv a$
$a^2 \equiv a^1.a^1$
$a^3 \equiv a^2.a^1$
$a^6 \equiv a^3.a^3$
$a^{12} \equiv a^6.a^6$
$a^{13} \equiv a^{12}.a^1$
$a^{26} \equiv a^{13}.a^{13}$
$a^{52} \equiv a^{26}.a^{26}$
$a^{78} \equiv a^{52}.a^{26}$
$a^{79} \equiv a^{78}.a^1$

El ejemplo anterior muestra dos distintas cadenas de adición para un sólo exponente e, de las cuales la segunda es mejor que la primera, dado que su longitud es menor. El exponente e del ejemplo anterior es un número de 7-bits y en el área criptográfica se utilizan exponentes de 512-bits a 1024-bits. Esto hace que de dificulte la generación de cadenas de adición de longitud mínima. Computacionalmente se considera un problema NP-Hard [22, 31, 24, 8, 9]. En la Sección 2.2.2 se describe el uso de las cadenas de adición en la Criptografía.

Encontrar cadenas de adición de longitud mínima beneficia directamente a los sistemas criptográficos, debido a que se reduce el costo computacional en el proceso de cifrado y descifrado de datos, así mismo, dicho beneficio se ve traslapado a los usuarios finales de SIC's que utilizan sistemas de cifrado asimétrico para el intercambio de información.

Para construir cadenas de adición de longitud mínima existen diversos mecanismos, los cuales se pueden clasificar como métodos determinísticos y métodos probabilísticos.

Los métodos deterministas, son algoritmos que se caracterizan por ser predictivos, es decir, si el algoritmo es inicializado con datos de entrada iguales en diferentes tiempos, éste siempre producirá la misma salida y se realiza la misma secuencia de estados. Por otro lado, los métodos estocásticos son algoritmos que basan sus resultados en la toma de decisiones aleatorias y hacen uso de heurísticas, es decir, realizan búsquedas de los mejores resultados utilizando diversos tipos de patrones estadísticos.

Dentro de los métodos deterministas se encuentran los siguientes mecanismos: Método Binario, Método Factor, Método Ventana y Deslizamiento de Ventana, [21]. Por otra parte, se han aplicado diversas heurísticas bio-inspiradas

(métodos estocásticos) para resolver dicho problema, tales como: algoritmo de optimización basada en colonia de hormigas (ACO)[30], algoritmos genéticos (GA)[9, 31], sistema inmune artificial (AIS)[8], y el algoritmo de optimización basada en cúmulos de partículas (PSO)[24]. Éstos últimos han obtenido mejores resultados en comparación con métodos deterministas, para instancias del problema con exponentes pequeños (menores a 64-bits) en la mayoría de los casos. Dichos métodos se encuentran descritos en el Capítulo 4. Uno de los algoritmos evolutivos que no ha sido explorado aún es la programación evolutiva, que ha presentado un desempeño competitivo en otros dominios. Este trabajo busca precisamente evaluar el desempeño de este algoritmo para minimizar la longitud de cadenas de adición.

1.2. Objetivos

1.2.1. Objetivo General

Adaptar el algoritmo de programación evolutiva para reducir la longitud de las cadenas de adición, incluso aquellas de exponentes mayores a 160-bits.

1.2.2. Objetivos Específicos

- Encontrar cadenas de adición para exponentes pequeños de menor o igual longitud en comparación con algoritmos encontrados en la literatura especializada pero con un número de evaluaciones igual o menor.
- Realizar pruebas estadísticas para comprobar si los resultados del algoritmo propuesto son mejores a los reportados en la literatura especializada.
- Adaptar el algoritmo propuesto para encontrar cadenas de adición de longitud mínima de números mayores a 160-bits y obtener resultados en desempeño competitivos con los reportados en la literatura especializada.

1.3. Hipótesis

El desarrollo de un algoritmo basado en programación evolutiva, será capaz de reducir la longitud cadenas de adición, incluso para números mayores de 160-bits con un costo computacional, medido por el número de evaluaciones realizadas, igual o menor a los reportados por algoritmos del estado del arte.

1.4. Justificación

Diariamente se generan grandes volúmenes de información de todo tipo, por ejemplo: movimientos bancarios, estrategias de mercado, datos confidenciales de

empresas u organizaciones, entre otras. Además, se está viviendo en una época donde la información tiene un gran valor para todas las sociedades, la cual es propensa a ser robada, o modificada por personas que se dedican a lucrar con la misma. Debido a los avances en computación, tales como el cómputo distribuido, procesadores de alta capacidad y algoritmos de criptoanálisis más eficientes, cada día los sistemas criptográficos se vuelven más propensos a ser corruptos, lo que pone en riesgo la confidencialidad e integridad de la información.

Los sistemas criptográficos deben adecuarse a estos avances y contextos y hacer más difícil o imposible el criptoanálisis. Para lograrlo, es necesario que los criptosistemas utilicen llaves cada vez más grandes, es decir, con mayor número de cifras. Esta necesidad conlleva a ocupar mayores recursos de las computadoras que procesan dicha información. Todo ello, nos lleva al punto central de esta investigación, la cual es minimizar la longitud de las cadenas de adición que son utilizadas en la exponenciación modular para cifrar y descifrar datos en los algoritmos de llave pública.

Como se ha mencionado anteriormente, generar cadenas de adición de longitud mínima impacta directamente a los sistemas criptográficos asimétricos, debido a que antes de cifrar información se generan claves públicas de hasta 2048-bits o de mayor longitud. Al generar cadenas de adición cortas para este tipo de exponentes, se reduciría el costo computacional y tiempo de cifrado y descifrado de datos realizado por los criptosistemas.

Los beneficios que se tendrían al solucionar este problema, se vería reflejado en la seguridad de los sistemas criptográficos, dado que los exponentes que se podrían manejar en algoritmos de cifrado asimétricos serían más grandes, complicando el criptoanálisis o ataques de fuerza bruta (los cuales consisten en probar todas las combinaciones posibles para buscar coincidencias en el cifrado) y beneficiando directamente a los algoritmos de cifrado y por ende, a las aplicaciones que hagan uso del criptosistema, dado que la fortaleza de un algoritmo de cifrado radica en el tamaño de la llave o clave, más que en el propio algoritmo.

1.5. Organización del Documento

El presente manuscrito se organiza de la siguiente manera: En el Capítulo 2 se definen diversos tipos de cadenas de adición y sus aplicaciones. En el Capítulo 3 se introducen los distintos tipos de heurísticas inspiradas en la naturaleza. La revisión del trabajo relacionado se presenta en el Capítulo 4 y en el Capítulo 5 se explica la adaptación del algoritmo de programación evolutiva para encontrar cadenas de adición de longitud mínima. En el Capítulo 6 se presentan los experimentos y resultados obtenidos por el algoritmo propuesto y finalmente en el Capítulo 7 se presentan las conclusiones y trabajos futuros derivados de esta investigación.

Capítulo 2

Cadenas de Adición

Las cadenas de adición pueden ser clasificadas por sus propiedades. En este capítulo se hace mención de las diferentes clases de cadenas encontrados en la literatura especializada, así como su uso en Criptografía.

2.1. Clases de Cadenas

Existen diversos tipos de cadenas de adición, las cuales se han ido definiendo en el transcurso del tiempo y dependiendo de las necesidades que se requirieron en su momento.

2.1.1. Cadenas de Adición

Una cadena de adición U con longitud $length(U)$ se define como una secuencia de números enteros positivos $U = u_1, u_2, u_3, ..., u_i, ...u_{length(U)}$ donde $u_1 = 1, u_2 = 2, u_{length(U)} = e$ y $u_{i-1} < u_i < u_{i+1} < u_{length(U)}$, u_i se obtiene de la suma de dos elementos previos, no siempre distintos [31, 8, 24, 9].

En otras palabras una cadena de adición U para un número dado e es una secuencia de números que cumple con las siguientes propiedades:

- El primer número es uno.
- Cada número es la suma de dos números anteriores.
- El número dado (e) se produce al final de la cadena.

Se denota de la siguiente forma:

$$U = u_1, u_2, u_3, ..., u_i, ...u_{length(U)}$$

Donde:
$u_1 = 1, u_2 = 2, u_{length(U)} = e$ y

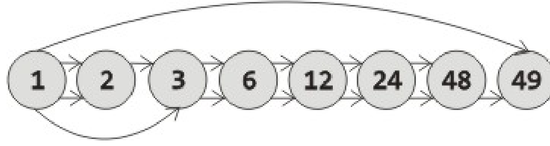

Fig. 2.1: Digrafo de una cadena de adición donde $e = 49$

$u_{i-1} < u_i < u_{i+1} < u_{length(U)}$,
u_i se obtiene de la suma de dos elementos previos, no siempre distintos.

Un ejemplo de cadena de adición U se puede representar en forma gráfica como en la Figura 2.1.

Entonces, una cadena de adición se puede ver como: $U = [n_0, n_1, ...n_{length(U)}]$; siendo el entero $length(U)$ la longitud de la cadena de adición U. Es importante mencionar que la longitud de una cadena de adición es igual al número de elementos de la cadena menos uno. En la Figura 2.1 se observa que todos los números son derivados de la suma de números previos, a excepción del primer número de la cadena que es 1. Por lo tanto, la primera posición de la cadena de adición no se cuenta para medir su longitud.

2.1.2. Cadenas de Adición Euclidianas

Las cadenas de adición Euclidianas son una variante de las cadenas de adición definidas en la sección anterior. La única diferencia es que una cadena Euclidiana no permite el doblado de números, es decir, un número que conforma a la cadena de adición proviene de la suma de dos números previos siempre distintos.

Se puede denotar como una cadena de adición Euclidiana U' con longitud $length(U')$ a una secuencia de números enteros positivos:

$$U' = u'_1, u'_2, u'_3, ..., u'_i, ...u'_{length(U')}$$

Donde:
$u'_1 = 1, u'_2 = 2, u'_{length(U')} = e$ y
$u'_{i-1} < u'_i < u'_{i+1} < u'_{length(U')}$,
u'_i se obtiene de la suma de dos elementos previos, siempre distintos.

Por su naturaleza, la longitud $length(U')$ de una cadena de adición Euclidiana mínima sería de mayor o igual longitud a una cadena de adición tradicional. En la Figura 2.2 se puede ver dicha diferencia respecto a la Figura 2.1.

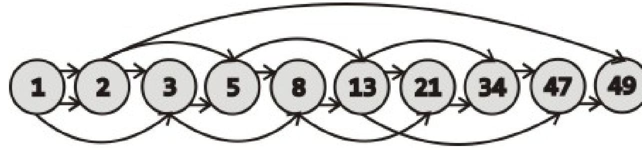

Fig. 2.2: Digrafo de una cadena de adición euclidiana donde $e = 49$

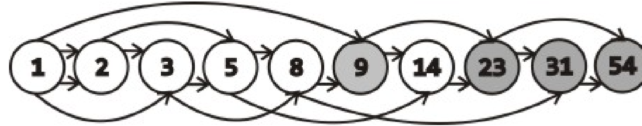

Fig. 2.3: Digrafo de una secuencia de adición para el conjunto $N = \{9, 23, 31, 54\}$ (Nodos oscuros en el grafo)

Cabe destacar que el uso de cadenas de adición Euclidianas en los criptosistemas reduce el riesgo de diversos tipos de ataques criptográficos que se realizan durante el proceso de cifrado y descifrado, tales como: Simple Power Analysis, Differential Power Analysis, Safe Error Attacks y Fault Attacks.

2.1.3. Secuencias de Adición

Una secuencia de adición tiene como fin el formar una lista secuencial S de números a partir de un conjunto de números enteros positivos $N = \{n_1, n_2, ..., n_m\}$. Al finalizar la secuencia S se tiene que cumplir con las siguientes propiedades:

- El primer número es uno.
- Cada número es la suma de dos números anteriores no siempre distintos.
- El último número de la secuencia es el mayor del conjunto N, es decir, n_m

Al final S parece una cadena de adición regular. Lo que la hace diferente, es que se construye a partir de un conjunto dado N y al momento de generar la secuencia de adición, es necesario empezar por el número mayor del conjunto n_m hasta llegar a 1. Todos los números del conjunto dado N aparecen en la cadena [8, 21, 5]. Véase Figura 2.3.

2.1.4. Cadenas de Adición Vectoriales

Las cadenas de adición Vectoriales se caracterizan por estar conformadas por vectores. Donde cada vector es producto de la suma de dos vectores pertenecientes a la cadena. Una cadena vectorial V cumple las siguientes propiedades:

- Los vectores iniciales son los vectores unitarios [1, 0, ..., 0]; [0, 1, 0, ..., 0]; ... ; [0, ..., 0, 1].
- Cada vector es la suma de dos vectores anteriores.
- El último vector es igual al vector dado

Un ejemplo mostrado en [18] sugiere al vector dado como [7, 15, 23], y se obtiene una cadena de adición vectorial con una longitud $length(V)$ de 9:

[1,0,0]
[0,1,0] [0,1,1] [1,1,1] [0,1,2] [1,2,3] [1,3,5] [2,4,6] [3,7,11] [4,8,12] [7,15,23]
[0,0,1]

Las cadenas de adición vectoriales pueden servir para encontrar las secuencias mínimas de adición o en exponenciación con vectores.

2.1.5. Cadenas de Adición-Substracción

Las cadenas de adición-substracción son similares a las cadenas de adición regulares pero éstas se denotan de la siguiente forma:

Una cadena de adición-substracción AS con longitud $length(AS)$ se define como una secuencia de números enteros positivos

$$AS = as_1, as_2, as_3, ..., as_i, ...as_{length(AS)}$$

Donde:
$as_1 = 1, as_2 = 2, as_{length(AS)} = e$ y
$as_i = a_j \pm a_k, j, k < i$

Este tipo de cadena fue desarrollado por la necesidad de que las cadenas de adición regulares no son aplicables a todos los tipos de operaciones, aunado al surgimiento de nuevos algoritmos de cifrado como el de curvas elípticas [22].

2.2. Aplicaciones de las Cadenas de Adición

En el Capítulo 1 de este trabajo se mencionó que las cadenas de adición tienen aplicación en el cifrado y descifrado de datos en algoritmos asimétrico o de llave pública, con el objetivo de reducir el número de multiplicaciones en la exponenciación. Estas operaciones implican un gran consumo de recursos computacionales, tales como el uso de memoria y procesador. A continuación se describe en qué consiste la exponenciación modular usada en el proceso de cifrado y descifrado de datos.

2.2.1. Aritmética modular

La aritmética modular, estudiada sistemáticamente en primer lugar por Carl Friedrich Gauss al final del Siglo XVIII, se aplica en Teoría de números, álgebra abstracta, **Criptografía**, y en artes visuales y musicales [38].

Las operaciones aritméticas que hacen la mayoría de las computadoras son aritmético modulares, donde el módulo es 2^b (b es el número de bits de los valores sobre los que operamos). Ésto se ve claro en la compilación de lenguajes de programación como C, donde todas las operaciones aritméticas sobre enteros toman módulo 2^{32} en la mayoría de las computadoras [38].

La aritmética modular cumple con la siguiente propiedad [27]:

Dados tres números $a, b, n \in \mathbb{N}$, se dice que a es congruente con b módulo n , y se escribe:

$$a \equiv b \ (mod \ n)$$

Un ejemplo particular de la aritmética modular es la aritmética del reloj [26]. En la aritmética del reloj cuando son las 10 de la mañana y se le agregan 5 horas se llega a las 3 de la tarde, es decir $10+5=3$. De la misma forma si a las 2 de la tarde se le quitan 4 horas, el resultado es las 10 de la mañana, lo que equivale a decir $2-4=10$. Este caso, en la arimética modular se le llama *aritmética módulo 12* y se realiza dentro del conjunto $Z_{12} = \{0,1,2,3,4,5,6,7,8,9,10,11\}$ cuyos elementos se llaman *enteros módulo 12*. En realidad cualquier número entero es equivalente a un entero módulo 12, que se obtiene como el residuo (nunca negativo) de la división entre 12.

Ejemplo: 29 es congruente con 5 módulo 12:

$$29 \equiv 5 \ (mod \ 12)$$

Ya que al dividir $29 \div 12$ resulta como cociente 2 y como residuo 5. Otra forma de expresar esto es $mod(29, 12) = 5$.

2.2.1.1. Exponenciación Modular

La exponenciación clásica consiste en ir multiplicando la base por sí misma tantas veces como indique el exponente; esto hace que el número de multiplicaciones aumente en función del exponente.

La exponenciación modular es una técnica de amplio uso en algoritmos de cifrado asimétrico en Criptografía y a diferencia de la exponenciación clásica, la exponenciación modular es menos costosa de realizar. Esto se debe a que después de cada multiplicación se calcula el módulo del resultado para que el resultado temporal no crezca excesivamente.

Se puede notar que $a \cdot b \ (mod\ p) \ \equiv \ [(a\ mod\ p)\ (b\ mod\ p)]\ mod\ p$, y por grande que sea el exponente, nunca es necesario multiplicar por enteros mayores que p [1].

2.2.2. Cadenas de adición en la exponenciación modular

Algoritmo 1 Exponenciación Modular

Entrada: Números enteros: a base, e exponente y m módulo
Salida: Número entero $exp = a^e\ mod\ m$

1: $exp := a$
2: **para** $i := 0;\ i < e;\ i++$ **hacer**
3: $\quad exp := (exp * a)\ mod\ m$
4: **fin para**
5: **devolver** exp

Usando el Algoritmo 1 se realizan tantas multiplicaciones y operaciones de módulo como indique el exponente. Es decir, si el exponente es 3000, necesitaremos 3000 multiplicaciones y 3000 divisiones (el cálculo del módulo es una división de la cual nos quedamos con el residuo de la división en lugar del resultado de la división) [25].

La exponenciación modular es menos costosa que la exponenciación clásica, sin embargo como se muestra también en el Algoritmo 1, el número de multiplicaciones y divisiones no se reduce, lo cual produce una latencia al momento de procesar dicho cálculo. Una forma de optimizar el cálculo del exponente de un número, es mediante una cadena de adición.

Por ejemplo, si se desea calcular $a^{12}\ (mod\ 9)$ la manera más obvia de realizar el cálculo es multiplicar $a \times a ... \times a$ 11 veces y de igual forma calcular el módulo. Una manera más eficiente para calcular es a través del uso de cadenas de adición. Aplicando la cadena de adición U, donde $U = \{1, 2, 4, 6, 12\}$ gracias a las leyes de los exponentes, el cálculo se reduciría a 4 multiplicaciones.

$a^1 = a\ (mod\ 9)$
$a^2 = a^1\ .\ a^1\ (mod\ 9)$
$a^4 = a^2\ .\ a^2\ (mod\ 9)$
$a^6 = a^2\ .\ a^4\ (mod\ 9)$
$a^{12} = a^6\ .\ a^6\ (mod\ 9)$

En Criptografía, dicho cálculo se ejecuta en el proceso de cifrado y descifrado de datos en algoritmos de llave pública. Por ejemplo el algoritmo RSA [37, 35] esquematizado en la Figura 2.4.

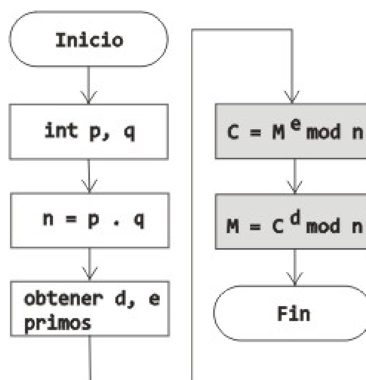

Fig. 2.4: Flujograma del algoritmo RSA

En el primer paso de la Figura 2.4 se obtienen los números primos aleatorios (se recomiendan que sean números de 100-dígitos)p y q; para ello se pueden ocupar métodos para probar su primalidad, tal como prueba Monte-Carlo para primalidad [35]. En el paso 2 se genera n como producto de $p \cdot q$. Posteriormente se obtiene d el cual es un número entero aleatorio que satisfaga:

$$mcd(d, (p-1)(q-1)) = 1$$

Dónde mcd significa "máximo común divisor". Finalmente e se calcula a partir de p, q y d, debe ser el inverso multiplicativo de d módulo $(p-1)(q-1)$:

$$e \cdot d \ \equiv \ 1 \ (mod(p-1)(q-1))$$

Teniendo como llave pública (d,n) y la llave privada $(e,\ n)$, las cuales se utilizan para cifrar y descifrar datos (pasos 4 y 5 de la Figura 2.4). Para realizar el cifrado se requieren $2 \cdot \log_2(e)$ multiplicaciones y $2 \cdot \log_2(e)$ divisiones. Para el descifrado es similar pero usando d en lugar de e. Dichos procesos se detallan en [35], [37], [28].

Ejemplo: Un cliente bancario, requiere una operación en línea, por lo tanto ingresa su contraseña la cual es "'HOME"'. esta es cifrada por la máquina cliente y descifrada por el servidor para verificar que ésta sea correcta. El cliente cifra el mensaje con su llave-privada y el servidor descifra el mensaje con la llave-pública del cliente. De igual forma si el servidor envía un mensaje al cliente, el cual es cifrado con la llave-privada del servidor y descifrado por el cliente con la llave-pública del servidor.

Suponiendo que la llave-privada del cliente es (53, 143) y su llave-pública (77,143).

El cliente cifra el mensaje con su llave-privada:

H $\Rightarrow 72^{53}$ mod $143 = 128$
O $\Rightarrow 79^{53}$ mod $143 = 118$
M $\Rightarrow 77^{53}$ mod $143 = 15$
E $\Rightarrow 69^{53}$ mod $143 = 49$

Para ejecutar dicho proceso se puede usar la siguiente cadena de adición $U = \{1, 2, 4, 6, 7, 13, 20, 33, 53\}$ con longitud $length(U) = 8$, para fines prácticos sólo se muestra el esquema para cifrar la letra "H", sin embargo para cada letra a cifrar se realiza el mismo proceso solo cambia el valor de M:

$M^1 = M \rightarrow 72$
$M^2 = M \,.\, M \rightarrow 72 \,.\, 72$ mod $143 = 36$
$M^4 = M^2 \,.\, M^2 \rightarrow 36 \,.\, 36$ mod $143 = 9$
$M^6 = M^4 \,.\, M^2 \rightarrow 9 \,.\, 36$ mod $143 = 38$
$M^7 = M^6 \,.\, M \rightarrow 38 \,.\, 72$ mod $143 = 19$
$M^{13} = M^7 \,.\, M^6 \rightarrow 19 \,.\, 38$ mod $143 = 7$
$M^{20} = M^{13} \,.\, M^7 \rightarrow 7 \,.\, 19$ mod $143 = 133$
$M^{33} = M^{20} \,.\, M^{13} \rightarrow 133 \,.\, 7$ mod $143 = 73$
$M^{53} = M^{33} \,.\, M^{20} \rightarrow 73 \,.\, 133$ mod $143 = 128$

El mensaje cifrado viaja por el canal inseguro como: 128 118 15 49. Finalmente llega al servidor y éste descifra el mensaje con la llave-pública del cliente:

$128 \Rightarrow 128^{77}$ mod $143 = 72 \rightarrow$ H
$118 \Rightarrow 118^{77}$ mod $143 = 79 \rightarrow$ O
$15 \Rightarrow 15^{77}$ mod $143 = 77 \rightarrow$ M
$49 \Rightarrow 49^{77}$ mod $143 = 69 \rightarrow$ E

Para ejecutar el descifrado, se puede usar la siguiente cadena de adición $U = \{1, 2, 4, 5, 9, 14, 23, 27, 50, 77\}$, con $length(U) = 9$; es decir, que para descifrar cada letra seran necesarias 9 multiplicaciones y 9 divisiones en lugar de 77. Sólo se muestra el esquema del descifrado de la primera cifra "128":
$C^1 = C \rightarrow 128$
$C^2 = C \,.\, C \rightarrow 128 \,.\, 128$ mod $143 = 82$
$C^4 = C^2 \,.\, C^2 \rightarrow 82 \,.\, 82$ mod $143 = 3$
$C^5 = C^4 \,.\, C \rightarrow 3 \,.\, 128$ mod $143 = 98$
$C^9 = C^5 \,.\, C^4 \rightarrow 98 \,.\, 3$ mod $143 = 8$
$C^{14} = C^9 \,.\, C^5 \rightarrow 8 \,.\, 98$ mod $143 = 69$
$C^{23} = C^{14} \,.\, C^9 \rightarrow 69 \,.\, 8$ mod $143 = 123$
$C^{27} = C^{23} \,.\, C^4 \rightarrow 123 \,.\, 3$ mod $143 = 83$
$C^{50} = C^{27} \,.\, C^{23} \rightarrow 83 \,.\, 123$ mod $143 = 56$
$C^{77} = C^{50} \,.\, C^{27} \rightarrow 56 \,.\, 83$ mod $143 = 72 \rightarrow$ H

Nota: en la práctica el cifrado y descifrado se realizan por bloques y no letra por letra.

Capítulo 3

Heurísticas Inspiradas en la naturaleza

Los científicos o ingenieros de varias áreas de investigación se han dado cuenta que los procesos naturales, comportamiento de algunas especies o el diseño y la estructura de ciertos seres vivientes, pueden ser benéficos para resolver diversos tipos de problemas; por ejemplo se sabe que la piel de un tiburón varía en rugosidad de acuerdo a las variaciones en el flujo de agua sobre su cuerpo por lo cual ingenieros y diseñadores han desarrollado un traje de baño con la forma y textura de la piel de tiburón [23].

Otro caso es el tren bala japonés Shinkansen (Figura 3.1) es uno de los más rápido del mundo, alcanzando una velocidad de 320 Km. por hora. Su andar es sorprendentemente tranquilo, gracias a su sistema de amortiguación inspirado en las plumas de los búhos y en el pico de los pájaros. Imitando la fisonomía de los búhos, que pueden volar a hurtadillas a través de la noche, fueron incorporados modificaciones en la parte superior del dispositivo para conectar el tren con los cables eléctricos de alimentación. La nariz del tren se asemeja a un pico de pájaro pescador. Esto permite a las aves zambullirse en el agua con la mínima pérdida de energía, y permite a los trenes salir de los túneles emitiendo bajos niveles de ruido [10].

En las ciencias computacionales a través del área de cómputo inteligente (rama de la inteligencia artificial), se ha dado cuenta que diversos procesos naturales tienen características que pueden ayudar a cumplir con el propósito de esta rama, el cuál consiste en el estudio de mecanismos adaptativos para generar o facilitar el comportamiento inteligente en ambientes complejos, inciertos y cambiantes. De ésto surgen los algoritmos bio-inspirados, con la motivación de mejorar búsquedas de soluciones y resolver problemas de optimización a través de la emulación de procesos inteligentes y colaborativos.

Fig. 3.1: Tren bala japonés Shinkansen

De acuerdo al fenómeno natural en que basan su diseño, estos algoritmos se clasifican en dos grupos: los que emulan el proceso evolutivo de las especies (algoritmos evolutivos) y los que emulan el comportamiento colaborativo de organismos muy simples como aves o insectos (algoritmos de inteligencia colectiva)[13]. Cabe mencionar que existen otros mecanismos que no entran en esta clasificación pero están inspirados en el funcionamiento del sistema inmunológico del ser humano [6].

3.1. Algoritmos Evolutivos

Los algoritmos evolutivos (EA's *evolutionary algorithms*) son un conjuntos de meta-heurísticas utilizadas para resolver una gran gama de problemas, que van desde la optimización de problemas combinatorios y numéricos, diseño de artefactos, búsqueda de información, control de dispositivos y aprendizaje automático, entre otros [2]. Los EA´s están basados en fenómenos relacionados con la evolución de las especies y la supervivencia del más apto.

En el modelo general de los EA's se tiene una población de posibles soluciones al problema a resolver; estas soluciones interactúan entre sí siguiendo principios Darwinianos de reproducción y selección del más apto, es decir, los individuos (soluciones) que estén mejor posicionados en el espacio de búsqueda según sea el problema, maximización o minimización, tienen una gran probabilidad de sobrevivir y reproducirse en futuras generaciones. La reproducción se hace mediante operadores de variación (cruza y mutación principalmente). Véase Figura 3.2.

En vista que los EA's son una emulación del proceso de selección natural se abstraen los siguientes componentes:

1. Representación de soluciones. Liga al mundo real con el EA, puede ser a nivel fenotipo (soluciones reales) o genotipo (bits o soluciones representadas mediante un código).

2. Función de aptitud o calidad. Representa la tarea a resolverse por lo tanto define la calidad de las soluciones, es decir, es la representación del ambien-

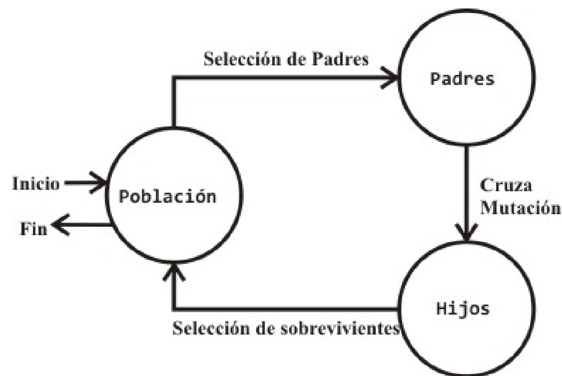

Fig. 3.2: Modelo de un Algoritmo Evolutivo

te natural en el que se mueven los individuos.

3. Población de soluciones. Grupo de soluciones potenciales al problema.
4. Mecanismos de selección de padres. El rol que tiene este mecanismo es el de distinguir entre individuos considerando su calidad, prefiriendo, en principio, a los mejores. Usualmente tiene elementos probabilistas.
5. Operadores de variación. Su función es crear nuevos individuos a partir de los ya existentes.
6. Mecanismo de reemplazo. Tiene como objetivo distinguir entre individuos con base en su calidad para mantener el tamaño de población fijo; este mecanismo por lo general es determinista.

Los principales paradigmas de los algoritmos evolutivos son:

- Algoritmos genéticos
- Programación genética
- Estrategias evolutivas
- Programación evolutiva
- Evolución diferencial

3.1.1. Algoritmos genéticos

Los algoritmos genéticos (GA *genetic algorithms*) son algoritmos de búsqueda, basados en los mecanismos de selección natural y la genética natural [15]. *Combinan la supervivencia del más apto entre estructuras de cadenas, con un cambio de información estructurado pero aleatorio, para formar un algoritmo*

Algoritmo 2 Algoritmo genético simple

1: Iniciar población
2: **mientras** No se satisfaga el criterio de paro **hacer**
3: Evaluar población
4: Seleccionar padres
5: Aplicar recombinación de hijos
6: Aplicar mutación de hijos
7: Realizar reemplazo
8: **fin mientras**

de búsqueda con algo del innovador talento de la búsqueda humana [15].

Un GA puede representar sus soluciones a un nivel genotipo o fenotipo, dependiendo del tipo de problema que se quiere resolver. Los GA's contienen los elementos de los EA's que son: selección de padres, mediante técnicas probabilistas (ruleta, sobrante y universal estocástico, muestreo determinístico y torneos), recombinación o cruza (uniforme, aritmética simple y completa), mutación (simple, uniforme o reordenamiento) y reemplazo. La forma general de trabajar se muestra en el Algoritmo 2.

3.1.2. Programación genética

La programación genética (GP *genetic programing*) [20] es muy similar a los algoritmos genéticos, su diferencia radica en que los individuos de la PG son estructuras en forma de árbol en lugar de cadena de bits como los GA's; lo cual hace que su representación sea vista como estructuras no lineales.

La meta de la GP es lograr que las computadoras aprendan a resolver problemas sin ser explícitamente programadas, generando soluciones a problemas a partir de la inducción de programas. El programador no especifica el tamaño, forma y complejidad estructural de los programas-solución, sino que los programas evolucionan hasta generar soluciones satisfactorias [36].

3.1.3. Estrategias evolutivas

Las estrategias evolutivas (ES's *evolutionary strategies*) emulan la evolución a nivel de los individuos, existe un operador de cruza, ya sea sexual o panmíctico (más de dos padres); este operador es secundario en las ES's [3]. El operador principal para este tipo de algoritmo es la mutación y utiliza números aleatorios generados con una distribución Gaussiana. Esta estrategia tiene la particularidad de que los valores de mutación varian con el tiempo y son auto-adaptables. La representación en las ES's es a nivel fenotípico y el proceso de selección de sobrevivientes es determinístico y extintivo, es decir, el peor de los individuos tiene cero probabilidad de sobrevivir.

Algoritmo 3 Algoritmo de estrategias evolutivas

1: Generar aleatoriamente una población inicial de soluciones.
2: Calcular la aptitud de la población inicial.
3: **mientras** No se satisfaga una condición de paro **hacer**
4: Seleccionar dos o más padres aleatoriamente.
5: Aplicar la cruza para crear hijos.
6: Aplicar el operador de mutación a todos los hijos.
7: Evaluar cada hijo.
8: Seleccionar los mejores individuos para la próxima generación basándose en su aptitud.
9: **fin mientras**

La selección de padres no es sesgada por los valores de aptitud ya que los padres son seleccionados de entre la población de manera aleatoria con una distribución uniforme. Cada padre puede ser seleccionado más de una vez o bien puede no ser seleccionado [3]. La manera en que trabajan las ES's se muestra en el Algoritmo 3.

3.1.4. Programación evolutiva

Programación evolutiva (EP *evolutionary programing*) fue propuesta por Fogel en 1960, donde remarca las relaciones de herencia y el comportamiento entre padres e hijos. La adaptación se concibe en este paradigma como un tipo de inteligencia [14].

EP emula la evolución a nivel de las especies, por lo tanto, no hay un operador de cruza, ya que en la naturaleza no es posible cruzar diferentes especies. La técnica de selección de sobrevivientes se basa en torneos estocásticos donde compiten los padres e hijos y sobreviven aquellos que en los torneos obtuvieron mayor puntaje, el cual se determina comparando la aptitud de cada individuo q veces con otros individuos de la población escogidos aleatoriamente.

Algoritmo 4 Algoritmo de programación evolutiva

1: Generar aleatoriamente un población inicial de soluciones.
2: Calcular la aptitud de la población inicial.
3: **mientras** No se satisfaga una condición de paro **hacer**
4: Aplicar la mutación a toda la población para crear hijos.
5: Evaluar cada hijo.
6: Seleccionar (mediante torneos estocásticos) los individuos de la próxima generación.
7: **fin mientras**

El principal y único operador de variación es la mutación. La selección de padres es determinista, ya que cada padre genera un hijo mediante el operador

de mutación (ver Algoritmo 4). Otra característica de EP es que usualmente opera a nivel fenotípico.

3.1.5. Evolución diferencial

Evolución diferencial (DE *differential evolution*) es una estrategia de búsqueda poblacional, muy similar a un algoritmo evolutivo estándar y fue propuesta por Storn y Price en 1995. La principal diferencia son sus operadores de cruza y mutación. El operador de mutación de evolución diferencial no está basado en una función de distribución de probabilidad predefinida, sino que la distribución del éste depende en la distribución actual de las soluciones (llamadas vectores) en la población [33].

Algoritmo 5 Algoritmo de evolución diferencial

1: Generar aleatoriamente una población inicial de vectores.
2: Calcular la aptitud de la población inicial.
3: **mientras** No se satisfaga una condición de paro **hacer**
4: Por cada padre, seleccionar tres vectores aleatoriamente
5: Crear un hijo usando los operadores de DE
6: Si el hijo es mejor que el padre
7: el hijo tomará el lugar del padre
8: **fin mientras**

DE tiene una representación a nivel fenotipo y el operador de mutación consiste en una diferencia aritmética entre pares de vectores seleccionados aleatoriamente. Cada vector padre (*target*) generará solo un vector hijo (*trial*). Para hacer esto, tres vectores seleccionados aleatoriamente de entre la población actual participarán (con el vector *target*) para generar al vector *trial*, por lo que hace que la selección de padres sea determinística.

La selección de sobrevivientes se lleva acabo de forma determinística, ya que si el vector hijo es mejor que el padre, este último será reemplazado en la siguiente generación, es decir, el vector más apto entre el target y el trial permanecerá en la población (véase Algoritmo 5).

3.2. Algoritmos de Inteligencia Colectiva

A diferencia de los EA's, los algoritmos de inteligencia colectiva (SIA's *swarm intelligence algorithms*) se basan en el comportamiento de animales que trabajan en grupo, ya sea para adquirir alimento, defenderse de depredadores o agruparse para obtener un beneficio en común [13]. De igual forma los SIA's difieren de los

AE's en la forma en que se busca la solución óptima en el espacio de búsqueda.

La población en SIA's se caracteriza por la colaboración y no por la competencia sobre quién es el más apto, como sucede con los EA's. Además, usualmente, no existe reemplazo y la población permanece constante durante todas las iteraciones.

Los principales algoritmos de inteligencia colectiva son:

- Algoritmo basado en Cúmulos de Partículas.
- Algoritmo basado en Colonia de Hormigas.
- Algoritmo basado en Enjambre de Abejas.
- Algoritmo basado en Bacterias.

3.2.1. Algoritmo basado en cúmulos de partículas

Kennedy y Eberhart en 1995 propusieron la optimización mediante cúmulos de partículas (PSO, *particle swarm optimization*), que es un algoritmo de búsqueda basado en la simulación del comportamiento social de los pájaros dentro de una parvada [13]. En PSO, los individuos, llamados partículas vuelan a través de un espacio de búsqueda multidimensional y sus cambios de posición dentro del espacio de búsqueda están basados en la tendencia socio-psicológica de los individuos a emular el éxito de otros individuos.

Algoritmo 6 Algoritmo de optimización mediante cúmulo de partículas

```
1: Generar aleatoriamente un cúmulo inicial de soluciones.
2: Calcular la aptitud del cúmulo inicial.
3: mientras No se satisfaga la condición de paro hacer
4:    Seleccionar al líder (o líderes del cúmulo)
5:    Para cada partícula, actualizar la posición (vuelo)
6:    Evaluar cada partícula
7:    Actualizar la memoria de cada partícula
8: fin mientras
```

Cada partícula representa una solución potencial al problema; su posición cambia de acuerdo a su propia experiencia y a la de sus vecinos, un conjunto de dichas partículas constituyen a un cúmulo.

Una de las características del PSO es que converge rápidamente, esto quiere decir que todas las partículas se colocan rápidamente en la vecindad del líder (mejor solución) del cúmulo, lo cual lo vuelve propenso a caer en óptimos locales; por ello se recomienda usar al operador de mutación para que el cúmulo

explore en otros espacios de búsqueda o utilizar la variante local-best que considera el uso de vecindades para controlar la convergencia. En el Algoritmo 6 se esquematiza el funcionamiento de un PSO genérico.

3.2.2. Algoritmo basado en colonia de hormigas

La optimización basada en colonia de hormigas (ACO, *ant colony optimization*) es una meta-heurística que abarca un conjunto de técnicas de optimización inspiradas en el comportamiento colectivo del forrajeo de las hormigas, las cuales son capaces de encontrar un camino corto entre el nido y la fuente de alimento por medio de la comunicación mediante rastros de feromonas artificiales. Fue desarrollado por Dorigo en 1995 [11] para resolver problemas de optimización combinatoria.

La parte crucial de ACO es que cada agente (hormiga) deja un rastro de feromona en el espacio de búsqueda que éste recorre; el rastro de feromona que deja cada hormiga es proporcional al encontrado, esto quiere decir que si en ese espacio no han pasado hormigas la cantidad de feromona en ese lugar será baja, y por lo tanto la hormiga que recorra ese lugar dejará un rastro de feromona débil. Las feromonas en ACO se representan mediante una matriz de números reales.

El éxito de ACO para encontrar el camino más corto se debe a que éste, con el tiempo, estará más cargado de feromonas, mientras que el camino más largo no contendrá grandes concentraciones de feromonas, debido a la evaporación que ocurre con el paso del tiempo.

Algoritmo 7 Algoritmo de optimización mediante colonia de hormigas

```
1: mientras No se satisfaga condición de paro hacer
2:     Construcción de soluciones por hormigas
3:     Actualización de feromona
4:     Servidor de acciones
5: fin mientras
```

En ACO existen tres procedimientos fundamentales [11], véase Algoritmo 7:

- Construcción de soluciones por hormigas: Administra a la colonia de hormigas las cuales recorren el espacio de búsqueda (un grafo dirigido) de manera aleatoria. Cada hormiga puede moverse aplicando una toma de decisiones estocástica usando la información de los rastros de feromona e información heurística. De esta forma, las hormigas construyen una solución subóptima al problema.
- Actualización de feromona: El valor del rastro puede incrementarse debido a que las hormigas depositan feromona en cada uno de los componentes

o conexiones que usan para moverse de un punto a otro en el espacio de búsqueda.

- Control de acciones: procedimiento utilizado para llevar a cabo acciones centrales que las hormigas no tiene la capacidad para desarrollar en forma individual.

3.2.3. Algoritmo basado en enjambre de abejas

El algoritmo basado en enjambre de abejas (ABC *artificial bee colony*) se basa en el comportamiento de forrajeo de las abejas [17]. Se caracteriza por ser fácil de implementar y encontrar buenas soluciones para problemas complejos.

Algoritmo 8 Algoritmo basado en enjambre de abejas

```
1: Inicializar población con fuentes de alimento
2: Evaluar el desempeño de la población
3: mientras No se satisfaga condición de paro hacer
4:    Aplicar fase de abejas empleadas
5:    Aplicar fase de abejas observadoras
6:    Aplicar fase de abejas exploradoras
7: fin mientras
```

La característica principal de ABC (Algoritmo 8) es que se basan en el movimiento o danza que realizan las abejas ya sea cuando se reproducen o van en búsqueda de alimento Debido a éste tipo de movimientos, los individuos de la población (abejas) recorren el espacio de búsqueda de tal forma que se encuentran soluciones sub-óptimas [17].

3.2.4. Algoritmo basado en bacterias

Existen dos formas de representar el algoritmo basado en bacterias, en uno se trata de representar el comportamiento quimiotáctico (QB-OA), el cual consiste en la forma de desplazamiento de las bacterias; y la otra se basa en la forma en que las bacterias buscan su alimento (BFOA), para ello se considera la forma de desplazamiento, reproducción y eliminación-dispersión.

QB-OA fue propuesto por Hans Bremermann en 1974 y el BFOA por Passino en el año 2002 [32]. Hernández Ocaña en [29] propuso el BFOA modificado (MBFOA) donde se reduce el número de condiciones que se requiere para ejecutar el BFOA original (ver Algoritmo 9). En MBFOA se simplifica la comunicación entre bacterias, lo cual produce un ambiente colaborativo entre ellas permitiendo encontrar zonas con altos contenidos de nutrientes, es decir, las mejores soluciones al problema.

Algoritmo 9 Algoritmo de optimización basado en bacterias modificado

1: Inicialización de parámetros
2: Generar cúmulo inicial aleatorio de bacterias
3: Evaluar población
4: **mientras** No satisfaga condición de paro **hacer**
5:     **mientras** No satisfaga condición de paro **hacer**
6:         Realizar el paso quimiotáctico para cada bacteria
7:     **fin mientras**
8:     Realizar el paso de reproducción
9:     Eliminar a las peores bacterias de la población
10:     Generar nuevas bacteria aleatoriamente.
11: **fin mientras**

3.3. Sistema Inmune Artificial

El sistema inmune artificial (AIS, *artificial immune system*) (ver Algoritmo 10), está inspirado en el sistema inmunológico humano. Los AIS utilizan el aprendizaje, la memoria y la recuperación asociativas. Dicho algoritmo se puede aplicar para resolver problemas de optimización, reconocimiento de patrones y clasificación de tareas [6].

Cabe destacar que los AIS's se basan en cuatro teorías de inmunidad; la teoría clásica, selección clonal, basada en red y basada en peligro.

La teoría clásica consiste en que el reconocimiento de antígenos (elementos foráneos en el cuerpo) motiva la generación de anticuerpos que los destruyen. Las epítopes (segmentos en la superficie de los antígenos) y los paratopes (segmentos en superficie de anticuerpos) sirven para medir la afinidad entre ellos.

Lo que hace exitoso al AIS es el mecanismo de selección clonal, el cual establece la idea de que sólo aquellas células inmunes (linfocitos B) que mejor reaccionen ante el estímulo de un antígeno serán clonadas. Los antígenos son moléculas que se encuentran expresadas en la superficie de los patógenos que pueden ser reconocidos por el sistema inmune y que además son capaces de dar inicio a la respuesta inmune para eliminarlos [7].

Los AIS, siguiendo la teoría basada en red, se caracterizan porque las células-B se interconectan para formar redes, mientras que las células-T tienen la función de ayudar a detectar antígenos y eliminarlos; y la diferencia con la teoría clásica es que aquí se asume que un linfocito puede ser estimulado por otro(s) linfocito(s) vecino(s) además de por un antígeno [6].

Finalmente los AIS que siguen la teoría basada en peligro se distinguen por que las células inmunes además de responder a los antígenos, también son capaces de reaccionar a células que representen peligro, incluyendo a otros agentes

Algoritmo 10 Algoritmo de un sistema inmune artificial

1: Inicializar un conjunto de ALCs C
2: Determinar conjunto de antígenos Dt
3: **mientras** No se cumpla condición de paro **hacer**
4:     **para** cada antígeno zpen Dt **hacer**
5:         Seleccionar un subconjunto de ALCs S para exponerlos a zp
6:         **para** cada ALC xi en S **hacer**
7:             Calcular la afinidad con el antígeno zp
8:             Seleccionar un subconjunto de S, llamado H con los ALCs con las más altas afinidades
9:             Adaptar los ALCs de H con algún método de selección basado en afinidad con el antígeno y/o con afinidad en la red entre ALCs de H
10:             Actualizar el nivel de estimulación de cada ALC en H
11:         **fin para**
12:     **fin para**
13: **fin mientras**

del mismo cuerpo.

Capítulo 4

Trabajo Relacionado

En este Capítulo se describen los diversos métodos diseñados para obtener cadenas de adición de longitud mínima con el fin de disminuir el número de multiplicaciones y divisiones que se realizan en la exponenciación modular de los algoritmos de cifrado asimétricos. Gracias a estas propuestas es posible reducir el costo computacional que implica realizar dicho cálculo. Estos métodos pueden clasificarse como métodos determinísticos y métodos estocásticos.

4.1. Métodos Determinísticos

Los métodos determinísticos, son algoritmos que realizan una rutina procedural, éstos se distinguen por obtener la misma salida para un parámetro, no importando las veces que sea ejecutado.

4.1.1. Método binario

Consiste en expandir un exponente e a su representación binaria $e = (e_{m-1}$ - $e_{m-2}\ \dots\ e_1\ e_0)$. La cadena binaria es escaneada bit por bit de izquierda a derecha o viceversa [39]. Por cada bit escaneado el último número de la cadena de adición es multiplicado por dos. Si el bit escaneado es uno, al último valor de la cadena de adición se le suma uno. El proceso se repite hasta recorrer toda la cadena binaria del exponente (Algoritmo 11). Básicamente el método binario trabaja con el peso Hamming del exponente [21].

Por ejemplo: sea el exponente $e = 77 = (1001101)_2$, aplicando el Algoritmo 11 el proceso se puede esquematizar en la Figura 4.1, de la cual se obtiene la cadena $U = \{1, 2, 4, 8, 9, 18, 19, 38, 76, 77\}$ con una longitud $length(U) = 9$.

Algoritmo 11 Algoritmo del método binario

Entrada: Número entero: $e = (e_{m-1} \dots\ e_1\ e_0)$
Salida: Cadena de adición $U = (1, 2, ..., e)$
1: $x = 1$
2: Añadir x a la cadena U
3: **para** $i = m - 2$ hasta e_0 **hacer**
4: $x = 2 * x$
5: Agregar x al final de la lista U
6: **si** $e_i == 1$ **entonces**
7: $x = x + 1$
8: Agregar x al final de la lista de U
9: **fin si**
10: **fin para**
11: **devolver** U

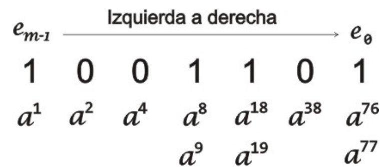

Fig. 4.1: Esquema del método binario para $e = 77$

El método binario es muy popular en Criptografía debido a su fácil implementación, sin embargo las cadenas de adición que se generan para exponentes mayores de 128-bits no son lo suficientemente cortas, a comparación de las obtenidas por otras técnicas, tal como el método de ventana que se discutirá más adelante.

4.1.2. Método factor

Consiste en la factorización del exponente e basándose en la regla se la multiplicación [18]. Se construye un árbol de números enteros, dónde el nodo raíz es el exponente e y sus n nodos hijos lo conforman p y q tal que $e = p \cdot q$, donde p es el número primo más pequeño que divide a e y $q > 1$ [19]. Para cada nodo hijo se repite el mismo proceso hasta que el nodo no pueda factorizarse, es decir, sea igual a 1. Si el nodo n es un número primo el nodo hijo izquierdo toma el valor de $n-1$ y el nodo derecho toma el valor de 1. Ver Algoritmo 12.

Por ejemplo, siguiendo el Algoritmo 12. Sea $e = 55$, $p = 5$ ya que el 5 es el divisor más pequeño de 55 y $q = 11$. Se repite el proceso para el nodo derecho 5, pero como es un número primo entonces $p = 1$ y $q = (5 - 1) = 4$. Repetir el proceso para cada nodo hasta llegar a 1. Posteriormente se procede a calcular el nodo izquierdo del árbol. Dicho proceso se esquematiza en la Figura 4.2.

Algoritmo 12 Construccion de árbol

Entrada: Número entero (n)

1: Añadir n como nodo raiz del arbol
2: **si** n != 1 **entonces**
3: Tomar p y q tal que $n = p.q$ y p is el primo mas pequeño dividiendo n y $p > 1$
4: **fin si**
5: **si** $p == n$ y $q == 1$ **entonces**
6: $q = n - 1$ y $p = 1$
7: Agregar p como nodo hijo de la derecha de n y q como nodo hijo de la izquierda de n
8: **si** $p > 1$ **entonces**
9: Llamar *construcción de árbol*(p)
10: **fin si**
11: **si** $q > 1$ **entonces**
12: Llamar *construcción de árbol*(q)
13: **fin si**
14: **fin si**

Después de haber generado el árbol, es necesario calcular la cadena de adición usando el Algoritmo 13. Para el ejemplo donde $e = 55$ la cadena generada es U ={1, 2, 4, 5, 10, 20, 40, 50, 55} con longitud $length(U) = 8$.

El método factor requiere de mucho procesamiento para exponentes mayores de 64-bits, debido al problema de la factorización en la construcción del árbol de factores y posteriormente obtener la cadena de adición,, haciéndolo un tanto lento y en ocasiones ineficiente en comparación con otros métodos, por ejemplo el método binario [21].

4.1.3. Método de ventana

El método de ventana está basado en una expansión $k - aria$ del exponente e [19], es decir, se usa la representación binaria del exponente $e = (e_{m-1}\ e_{m-2} \ldots\ e_1\ e_0)$. Los bits del exponente son divididos en k bits llamados palabras o ventanas, cada ventana tiene una longitud máxima la cual tiene que ser definida por el usuario (ver Algoritmo 14). En algunas investigaciones como en [5] se opta por tomar ventanas de 5-bits ya que el uso de ventanas grandes hace que las cadenas puedan tener una mayor longitud.

Así mismo k divide a m tal que $h = m/k$, donde m es la longitud de la representación binaria de e. Esto quiere decir que h es el número de ventanas k. Pero si k no divide a m, entonces la representación binaria del exponente debe ser completada por los ceros necesarios hasta que m divida a k [8].

Algoritmo 13 Calcular cadena

Entrada: Nodo raíz n y nodos hijos derecho e izquierdo p y q
Salida: Cadena de adición $U = (1, 2, ..., e)$

1: **si** $p == 2$ **entonces**
2: Doblar x
3: Agregar x al final de la cadena
4: **si no**
5: **si** $p\ != 1$ **entonces**
6: Pasar a siguiente rama: $n = p$, p nodo derecho y q nodo izquierdo
7: llamar Calcular Cadena(n)
8: **fin si**
9: **fin si**
10: Guardar el valor de actual de x en el nodo n
11: **si** $q == 2$ **entonces**
12: Doblar x
13: Agregar x al final de la cadena
14: **si no**
15: **si** $q\ != 1$ **entonces**
16: Pasar a siguiente rama: $n = q$, p nodo derecho y q nodo izquierdo
17: Llamar calcular cadena(q)
18: **fin si**
19: **fin si**
20: **si** $n\ != e$ y $p == 1$ **entonces**
21: Agregar los datos almacenados en el nodo actual de x
22: Agregar x a la cadena
23: **si no**
24: **si** $n == e$ y $p == 1$ **entonces**
25: Agregar el valor almacenado a la derecha del nodo de x
26: Agregar x a la cadena
27: **fin si**
28: **fin si**
29: **devolver** U

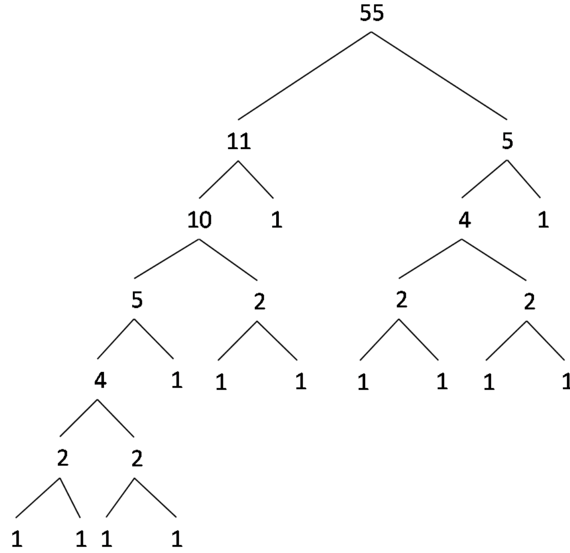

Fig. 4.2: Árbol de factores generado con el Algoritmo 12 para $e = 55$

Algoritmo 14 Algoritmo del método de ventana

Entrada: Entero $e = (e_{m-1} \dots\ e_1\ e_0)$, $h = m/k$
Salida: Cadena de Adición $U = (1, 2, \dots, e)$
1: Generar y guardar $Temp = 1, 2, 3, 4, \dots, 2^k - 1$
2: Dividir e en ventanas k-bit W_i para $i = 0, 1, 2, \dots, h - 1$
3: $x = W_{h-1}$ y agregar x al final de la lista U {valor decimal de la ventana más a la izquierda}
4: **para** $i = e_{h-2}$ hasta e_0 **hacer**
5: $x = 2 * x$
6: Agregar x al final de la lista U
7: **si** $e_i ==$ último bit de la ventana actual **entonces**
8: $x = x+$ valor decimal de la ventana actual
9: Agregar x al final de la lista U
10: **fin si**
11: **fin para**
12: $U = sort(Temp \cup U)$
13: **devolver** U

Por ejemplo, sea $e = 2011 \rightarrow (11111011011)_2$ con $m = 11$ y $k = 3$, es necesario agregar un cero a la cadena binaria para dividir en 3-bit ventanas $(\mathbf{0}11\ 111\ 011\ 011)_2$. Usando el Algoritmo 14 se obtiene:

$$Temp = 1 \rightarrow 2 \rightarrow 3 \rightarrow 5 \rightarrow 7$$
$$U = 3 \rightarrow 6 \rightarrow 12 \rightarrow 24 \rightarrow 31 \rightarrow 62$$
$$\rightarrow 124 \rightarrow 248 \rightarrow 251 \rightarrow 502 \rightarrow 1004 \rightarrow 2088 \rightarrow 2011$$

4.1.3.1. Método de deslizamiento de ventana

El método del deslizamiento de ventana (SWM, *sliding window method*) [19] sigue el principio del método de ventana con la particularidad de que las ventanas que se generan en el SWM empiezan con 1_2 y terminan en 1_2. Haciendo que se generen ventanas de ceros (Z)y no-ceros (NZ). Ver Figura 4.3 Esta separación de ventanas hace más eficiente la generación de la cadena de adición [18].

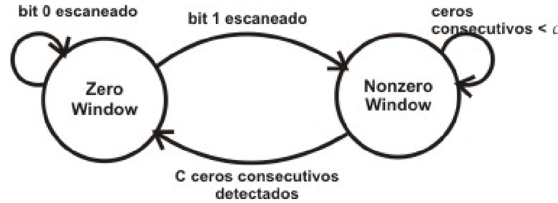

Fig. 4.3: Generador de ventanas para SWM

El algoritmo SWM (Algoritmo 15) es similar a su predecesor, con la diferencia en la generación de ventanas y el uso de secuencias de adición, descritas en la Sección 2.1.3. Para generar una secuencia de adición se puede ocupar el método publicado en [8]. El cual se describe en el Algoritmo 16.

Por ejemplo, sea $e = 77$ y el número de ceros consecutivos permitidos $C = 1$, siguiendo el método de la Figura 4.3 las ventanas quedarían de la siguiente manera: $\underline{1}00\underline{11}0\underline{1}$ por lo tanto su representación decimal es $NZ = \{1, 1, 3\}$. A continuación se genera una secuencia de adición para las ventanas NZ. Seq = $\{\underline{1}, 2, \underline{3}\}$ usando el Algoritmo 16. Por último, se recorre la cadena binaria de e y por cada bit el último valor de U es multiplicado por 2. Si el bit escaneado es el último de una ventana NZ entonces al último valor de U se le suma el valor decimal de la ventana NZ, dicho proceso de muestra en la Figura 4.4. Al final la cadena de adición $U = \{1, 2, 3, 4, 8, 16, 19, 38, 76, 77\}$ con longitud $length(U) = 9$.

El método SWM tiene un buen desempeño para exponentes grandes, es decir, mayores de 128-bits [18], es decir, construye cadenas de adición con menor

Algoritmo 15 Algoritmo del método de deslizamiento de ventana

Entrada: $e = (e_{m-1} \dots\ e_1\ e_0)$
Salida: Cadena de Adición $U = (1, 2, \dots, e)$

1: Descomponer e en h ventanas cero y no-cero W_i
2: Generar una secuencia de adición para las ventanas NZ $[W_0, W_1, \dots, W_{NZ-1}]$
3: Agregar la secuencia de adición al final de la lista U
4: $x = W_{h-1}$
5: **para** $i = e_{h-2}$ hasta e_0 **hacer**
6: $x = 2 * x$
7: Agregar x al final de la lista U
8: **si** $e_i ==$ último bit de la ventana actual NZ **entonces**
9: $x = x+$ valor decimal de la ventana NZ
10: Agregar x al final de la lista U
11: **fin si**
12: **fin para**
13: **devolver** U

Algoritmo 16 Generador de secuencias de adición [8]

Entrada: Conjunto ordenado de n enteros $U = e_1, e_2, \dots\ e_{n-1},\ e_n$ en orden ascendente.
Salida: Secuencia de adición $e_1, e_2, \dots\ e_{n-1},\ e_n$.

1: $k = n - 2$
2: Set $U = u_1 = e_1, u_2 = e_2, \dots, u_k = e_{n-2}$
3: $H = h_1 = e_{n-1}, h_2 = e_n$
4: $W =$
5: **mientras** $U \neq$ **hacer**
6: $\Delta = (h_2 - h_1)$
7: $W = W \cup h_2$
8: $h_1, h_2 = max_two_elements(u_k, \Delta, h_1)$
9: **si** $\Delta < u_k\ y\ \Delta \notin U$ **entonces**
10: $U = sortSet(U \cup \Delta)$
11: **fin si**
12: **si** $\Delta \in U$ **entonces**
13: $k = k - 1$
14: **fin si**
15: **fin mientras**
16: **devolver** W

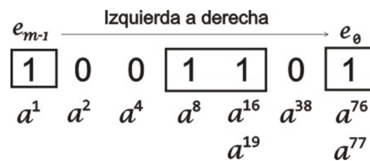

Fig. 4.4: Método SWM donde $e = 77$

longitud en comparación de otros métodos deterministas, por ejemplo, el método Binario.

4.2. Métodos Estocásticos

Existen diversos esfuerzos donde se ocupan técnicas de cómputo inteligente para optimizar el problema de reducción de cadenas de adición. A continuación se presentan las más sobresalientes.

4.2.1. Algoritmos de Inteligencia Colectiva

4.2.1.1. Cadenas de adición mínimas usando la colonia de hormigas

En [30] se propone utilizar el algoritmo de optimización basado en colonia de hormigas (ACO) basado en un esquema multiagente, donde cada agente representa a una hormiga. Cada hormiga o agente cuenta con una memoria local, y se comunica con otras hormigas a través de una memora compartida.

Las hormigas siguen un camino de feromonas que ellas mismas dejan en el camino para que las demás hormigas puedan llegar a un objetivo; en este caso el objetivo es reducir la longitud de la cadena de adición.

La memoria compartida es la que permite la comunicación entre las hormigas. Por medio de ésta, las hormigas colaboran en la construcción de la cadena de adición. La memoria compartida es representada mediante una matriz de 2 dimensiones con e número de filas, donde e es el exponente. Y el número de columnas es proporcional al número de filas.

Por otro lado, la memoria local está dividida en dos partes: la primera parte almacena la cadena de adición encontrada por la hormiga, la cual se representa en un matriz de e entradas; y la segunda parte almacena las características de la solución, es decir, cómo fue que llegó a ese resultado.

En [30] se reportan resultados de pruebas con exponentes de 32-bits, 64-bits y 128-bits. No se reportan pruebas del algoritmo con exponentes difíciles (exponentes que con métodos deterministas no se puede reducir considerablemente las cadenas de adición) ni exponentes de longitud mayores de 128 bits.

4.2.1.2. Cadenas de adición mínimas con un algoritmo de optimización basado en cúmulo de partículas

En [24] es propuesto un algoritmo de Optimización basado en Cúmulo de Partículas (PSO). Una partícula es representada por una cadena de adición válida $U = u_1, u_2, \ldots, u_e$. La cual es generada por un método estocástico que

permite tres maneras de construcción:

1. Doblado de número $u_{i+1} = 2.u_i$
2. Suma de números previos $u_{i+1} = u_i + u_{i-1}$
3. Suma de un número aleatorio $u_{i+1} = u_i + u_{rand}$

La adaptación del PSO original, se muestra en el Algoritmo 17, donde se representa al pBest (mejor posición de cada partícula) como la longitud de la cadena de adición.

Algoritmo 17 PSO

Entrada: Número entero e
Salida: Cadena de adición $U = (1, 2, ..., e)$

```
1: Generar poblacion inicial
2: mientras El número máximo de generaciones no sea alcanzado hacer
3:   para Cada partícula hacer
4:     Calcular aptitudes de cada individuo
5:     si aptitud de la partícula es mejor que la pBest entonces
6:       pBest = aptitud de la partícula actual
7:     fin si
8:     Buscar al mejor vecino y asignarle la posición gBest
9:     Actualizar la posición y la velocidad de la partícula actual
10:  fin para
11: fin mientras
12: U = partícula más corta encontrada
13: devolver U
```

El PSO publicado en [24] es uno de los trabajos más recientes en búsqueda de cadenas de adición mínimas. Sin embargo en dicha investigación sólo reportan resultados para exponentes pequeños y no realizan pruebas con exponentes difíciles. Sin embargo, reportan algunos resultados mejores que los publicados en [8].

4.2.2. Algoritmos Evolutivos

4.2.2.1. Cadenas de adición óptimas usando un algoritmo genético

En [9] se propone un algoritmo genético (GA). Los cromosomas son representados como una cadena de adición de un exponente dado e. Por ejemplo: Si se quiere minimizar la cadena de adición para $e = 6271$, una posible cromosoma es U ={1, 2, 4, 8, 10, 20, 30, 60, 90, 180, 360, 720, 1440, 2880, 5760, 5970, 6150, 6240, 6270, 6271}. La manera en que se construyen los individuos es similar a la mencionada para el algoritmo de PSO en la sección anterior.

La adaptación del GA se muestra en el Algoritmo 18, donde la aptitud, al igual que para el caso del PSO de la sección anterior, está en función de la longitud de la cadena de adición y donde se utilizan operadores tradicionales de cruza y mutación.

Algoritmo 18 Algoritmo Genético

Entrada: Números enteros: exponente (e), Max_población (N)
Salida: Cadena de adición $U = (1, 2, ..., e)$
1: Generar poblacion inicial N
2: **mientras** El número máximo de generaciones no sea alcanzado **hacer**
3: Calcular aptitudes de cada individuo
4: Seleccionar N padres para reproducirse
5: Con la probabilidad Pc aplicar cruza a los padres
6: Aplicar mutación a los hijos con probabilidad Pm
7: Reemplazar a los padres por los hijos para la siguiente generación
8: **fin mientras**
9: Calcular aptitudes de cada individuo
10: U := individuo con mejor aptitud
11: **devolver** U

Los resultados obtenidos por el algoritmo genético publicado en [9] fueron comparados con resultados de métodos deterministas, GA obtuvo mejores resultados, sin embargo no se reportan pruebas con exponentes difíciles ni exponentes grandes (mayores a 128-bits). Por otro lado, el GA publicado en [9] permite la generación de cadenas de adición inválidas y requiere de un proceso de reparación.

4.2.2.2. Algoritmo genético con reparación y mecanismos de búsqueda local para encontar cadenas de adición de longitud mínima

El algoritmo genético con reparación y mecanismos de búsqueda local (REPLS-GA, repair and local search genetic algorithm) propuesto en [31], es una mejora del GA propuesto en [9]. REPLS-GA trabaja solamente con un conjunto de cadenas de adición válidas (descartando a las cadenas invalidas) a diferencia de su predecesor. La representación de soluciones se realiza de la misma forma que en [9]. Otra diferencia importante de REPLS-GA es el uso de una búsqueda local en el momento de la mutación. El operador de cruza de genes en REPLS-GA utiliza dos puntos de cruza para generar hijos.

Los resultados obtenidos con REPLS-GA [31] son significativamente mejores que su predecesor [9]. Sin embargo no se reportan resultados de pruebas con exponentes mayores a 128-bits. Además que se utilizaron un alto número de

evaluaciones por ejecución del algoritmo.

4.2.3. Sistema inmune artificial para la generación de cadenas de adición cortas

En [8] se propone un sistema inmune artificial AIS que hace uso de dos elementos de este tipo de algoritmos: los antígenos (microorganismos extraños) y los anticuerpos (los actores principales de la respuesta inmune adaptativa). El algoritmo AIS se basa en un mecanismo llamado principio de selección clonal, que es la forma en que los anticuerpos eliminan a un antígeno extraño [8].

Un antígeno es representado como el exponente que se desea alcanzar. Los anticuerpos, por otra parte, están representados por la pareja $(U, length(U))$, donde U es la secuencia de la cadena de adición, la cual es construida de siguiendo los mismos principios que los métodos mencionados previamente. $length(U)$ es un número entero positivo que representa la longitud de U, el número de pasos necesarios para lograr el objetivo deseado. La población de anticuerpos representa las posibles soluciones para el problema.

AIS es una meta-heurística que por su arquitectura no necesita muchas evaluaciones para poder encontrar resultados competitivos en la reducción de cadenas de adición. Sin embargo su implementación no es tan simple, se necesita cierto dominio del tema de sistemas inmunes artificiales. En [8] los autores combinaron también el AIS con el método determinista Deslizamiento de Ventana para generar cadenas de adición mínimas para exponentes desde 128-bits hasta 1024-bits.

Capítulo 5

Algoritmo Propuesto

Para encontrar cadenas de adición mínimas, en esta investigación se propone utilizar un algoritmo de programación evolutiva (EP), dado que no se ha reportado en la literatura especializada el uso de EP para resolver el problema de optimizar cadenas de adición. Además EP es un algoritmo sencillo de implementar a comparación de otras meta-heurísticas. En este capítulo se describe la implementación de EP para encontrar cadenas de adición mínimas.

5.1. Elementos fundamentales en programación evolutiva

El elemento principal en EP es el individuo, el cual compite para pasar a las siguientes generaciones. EP, introducida en la Sección 3.1.4, consta de dos elementos fundamentales: (1) El operador de variación (Mutación) y (2) el mecanismo de reemplazo (torneos estocásticos). EP comienza con una población inicial generada de forma aleatoria. Posteriormente cada individuo realiza una mutación para generar un solo hijo. Los individuos que pasen a la siguiente generación serán aquellos que tengan mayor número de victorias en torneos estocásticos y no los que tengan mejor aptitud. Este proceso permite que haya mayor diversidad y se intenta evitar la convergencia prematura del algoritmo.

Es importante resaltar que EP no hace selección de padres, ya que todos los miembros de la población generan un hijo mediante la mutación. Y los individuos con mayor número de victorias (en los torneos estocásticos) son los que sobreviven para la siguiente generación. Dicho proceso se puede esquematizar en la Figura 5.1.

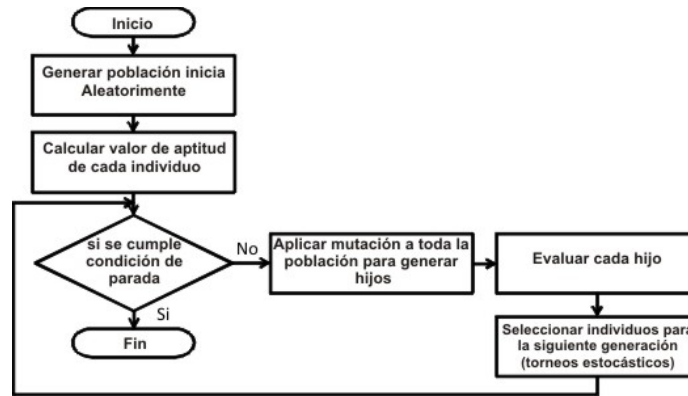

Fig. 5.1: Diagrama de flujo de EP

5.2. Representación de la población y función de aptitud

Inspirados en [9, 31, 8, 24] un individuo será representado por una cadena de adición válida $U = \{u_1, u_2, \ldots, u_i, \ldots, u_{length(U)}\}$, es decir, que un individuo será una lista de números enteros positivos de números que cumplan las propiedades de las cadenas de adición. Esto quiere decir que la solución estará representada a nivel fenotípico (no se requiere de proceso de decodificación de soluciones).

La aptitud de cada individuo es representada por las longitudes de las cadenas de adición $length(U)$ [9, 31, 8, 24]. El problema que se quiere resolver es minimizar cadenas de adición, así que los individuos con menos elementos (menor longitud) se consideran mejores. Dicha representación se ilustra en la Figura 5.2.

id	Individuo	longitud (Valor de aptitud)
0	1 2 · · · · · e	7
1	1 2 · · · · · · e	8
2	1 2 · · · e	5
3	1 2 · · · · e	6
	....	
	....	
	....	
TamPob	1 2 · · · · e	6

Fig. 5.2: Representación de la población

5.2.1. Población inicial

Al igual que en [9, 31, 8, 24] cada individuo de la población es construido mediante las siguientes tres estrategias:

1. Doblado del número previo $u_{i+1} = 2.u_i \equiv u_{i+1} = u_i + u_i$
2. Suma de dos números previos $u_{i+1} = u_i + u_{i-1}$
3. Suma de un número previo más un número aleatorio de la cadena $u_{i+1} = u_i + u_{rand}$

Donde $u_{i-1} < u_i \leq e$, es decir, la factibilidad de la solución siempre se mantiene. El proceso general se describe en el Algoritmo 19, donde los primeros dos elementos de la cadena de adición U siempre serán el número 1 seguido del número 2 y el tercer elemento puede ser 3 ó 4 este es elegido de forma aleatoria. Posteriormente la función Completar es invocada (propuesta en [31]). Esta función utiliza los parámetros f y g como probabilidad de que ocurra una de las tres estrategias enlistadas anteriormente.

El Algoritmo 20 detalla la función *Completar*. $Flip(prob)$ determina la probabilidad de que suceda o no suceda un evento determinado.

Algoritmo 19 Cadena_Factible(e)

Entrada: Exponente e
Salida: Una cadenas de adición factible $U = u_1, u_2, ..., u_l$, donde $u_{length(U)} = e$
1: Establecer $u_1 = 1$ y $u_2 = 2$
2: Establecer $u_3 = rnd(3, 4)$
3: $Completar(U, 4, e)$

El último proceso observado en el Algoritmo 20 es usado para generar cadenas de adición válidas. A diferencia de otras opciones propuestas en [8, 31], donde valores aleatorios son considerados, en este trabajo se hace una búsqueda determinista (renglones 12 - 16 en el Algoritmo 20), se comienza a sumar el último valor de la cadena u_i con u_{aux}. El valor resultante de la suma tiene que ser menor o igual al exponente e, de lo contrario aux se decrementa y se repite el proceso, hasta encontrar un número que no invalide la cadena de adición.

5.3. Operador de variación

Inspirados en [31] en esta investigación se propone hacer una búsqueda local en el operador de variación (mutación) del algoritmo EP, es decir, generar t mutantes para cada individuo y el de mejor aptitud es seleccionado como hijo. Esto con el objetivo de explorar más en el campo de búsqueda y acotar el número de generaciones necesarias para que el algoritmo propuesto converja en

Algoritmo 20 Completar(U, k, e)

Entrada: Una cadena de adición incompleta U. k es la siguiente posición de la cadena e es el exponente
Salida: Una cadena de adición completa y factible U

1: establecer $i = k - 1$
2: **mientras** $u_i \neq e$ **hacer**
3: **si** $Flip(f)$ **entonces**
4: $u_{i+1} = 2u_i$
5: **si no**
6: **si** $Flip(g)$ **entonces**
7: $u_{i+1} = u_i + u_{i-1}$
8: **si no**
9: $u_{i+1} = u_i + u_{rand}$
10: **fin si**
11: **fin si**
12: **mientras** $u_{i+1} > e$ **hacer**
13: $aux = i - 1$
14: $u_{i+1} = u_i + u_{i-aux}$
15: $aux = aux - 1$
16: **fin mientras**
17: **fin mientras**
18: **devolver** U

una solución factible.

En la Figura 5.3 se muestra un ejemplo donde $e = 39$ y el punto de mutación $pMut = 5$. Entonces a partir del número que está en la posición $pMut$ se generan t mutantes (en el ejemplo son 4 mutantes por individuo). En el ejemplo de la Figura 5.3, el mutante 2 ó 4 será elegido como hijo ya que son los que tienen mejor aptitud con respecto a los mutantes 1 y 3.

En el Algoritmo 21 se detalla el proceso de mutación en el algoritmo propuesto.

5.4. Mecanismo de reemplazo

El mecanismo de reemplazo que se utiliza en EP, consiste en torneos estocásticos, en los que se escogen a los individuos para las siguientes generaciones. Dichos torneos se implementan considerando lo siguiente:

1. La población actual y sus hijos (mutaciones) se fusionan en un conjunto
2. Un contador de número de victorias se añadirá a cada individuo en el conjunto

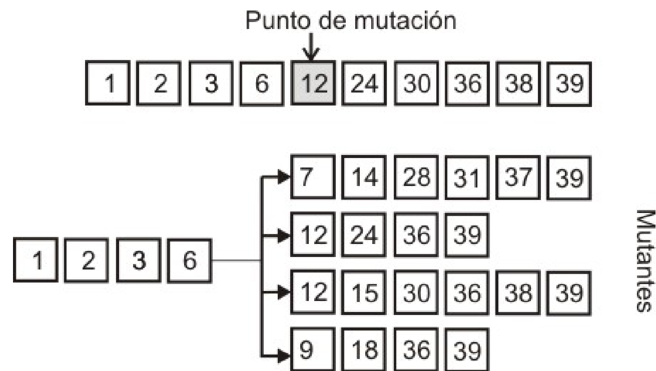

Fig. 5.3: Mutaciones en EP

Algoritmo 21 Mutación(U)

Entrada: Cadena de adición válida U
Salida: Su hijo correspondiente $U^{'}$
1: Generar (t) copias del individuo U, llamado $C_1, C_2, \ldots, C_t$
2: Generar un punto de mutación $i = rnd(3, U_l)$
3: **para** $k = 1$ to t **hacer**
4: Elementos 1 hasta $i - 1$ permanecen intactos en C_k
5: Completar(C_k, i, e)
6: **fin para**
7: Seleccionar de $C_1, C_2, \ldots, C_t$ aquel que tenga menor longitud $U^{'}$
8: **devolver** $U^{'}$

---

3. Cada individuo en el conjunto va a competir contra q individuos escogidos aleatoriamente del conjunto mediante enfrentamientos frente a frente

4. La longitud de cada individuo, es decir, su aptitud, será el criterio de comparación

5. El contador de número de victorias se incrementa cada vez que el individuo es mejor con respecto a alguno de los q individuos elegidos al azar.

6. Después de que todos los individuos han participado en los torneos estocásticos se ordenan según su número de victorias y la primera mitad permanecerá para la próxima generación, mientras que la segunda mitad se elimina del proceso.

Algoritmo 22 Reemplazo(Conjunto)

Entrada: Conjunto combinado de n individuos de la población actual y sus n hijos.
Salida: La población para la siguiente generación Pop$^{'}$ de tamaño n.
1: **para** $j = 1$ hasta $(2.n)$ **hacer**
2: $victorias_j = 0$
3: **para** $k = 1$ hasta q **hacer**
4: $i = rnd(1, 2.n)$
5: **si** Length(Conjunto$_j$) $<$ Length(Conjunto$_i$) **entonces**
6: victorias$_j$ $++$
7: **fin si**
8: **fin para**
9: **fin para**
10: Ordenar *Conjunto* basado en *victorias*
11: **para** $j = 1$ hasta n **hacer**
12: $Pop^{'}_j = Conjunto_j$
13: **fin para**
14: **devolver** $Pop^{'}$

El pseudocódigo del proceso de reemplazo se presenta en el Algoritmo 22, donde los torneos estocásticos se espera coadyuven a mantener la diversidad en la población y promover mejores resultados finales.

5.5. EP Modificado para exponentes pequeños

En el Algoritmo 23 se muestra el pseudocódigo del algoritmo EP propuesto.

A continuación se presenta un ejemplo que utiliza el algoritmo propuesto EP, para encontrar cadenas de adición de longitud mínima de un exponente

Algoritmo 23 EP_Modificado(e)

Entrada: Exponente e

Salida: Una cadena de adición cuasi-óptima U

1: Pop= $\emptyset$
2: **para** $j = 0$ hasta n **hacer**
3: Pop$_j$ = Cadena_factible(e)
4: **fin para**
5: **para** $k = 0$ hasta $MAXGEN$ **hacer**
6: Hijo = $\emptyset$
7: **para** $m = 0$ hasta n **hacer**
8: Hijo$_m$ = Mutación(Pop$_m$)
9: **fin para**
10: Pop$'$ = Reemplazo(Pop + Hijo)
11: Pop=Pop$'$
12: **fin para**

pequeño. Se considera un exponente pequeño cuando éste es menor de 128-bits.

Antes de empezar, debemos definir los valores de los siguientes parámetros:
Tamaño población (n) = 2
Máximo de generaciones MAXGEN = 1
Número de mutantes (t) = 2
Número de individuos por encuentros (q) = 3
Probabilidad de Doblado (f) = 0.7
Problabilidad de sumar números previos (g) = 0.2

Sea $e = 131$, el Algoritmo 23 tiene el siguiente comportamiento:

1. Generar población inicial aleatoriamente:
   $Padre[0] \rightarrow 1, 2, 3, 6, 9, 18, 27, 54, 108, 126, 129, 131$ $L = 11$
   $Padre[1] \rightarrow 1, 2, 3, 5, 10, 20, 40, 80, 120, 130, 131$ $L = 10$

2. Crear la descendencia con el operador de variación (mutación) para la generación 1
   Para $Padre[0]$
   $Mut[0] \rightarrow 1, 2, 3,$ **6, 12, 24, 48, 96, 120, 126, 128, 131** $L = 11$
   $Mut[1] \rightarrow 1, 2, 3,$ **5, 8, 16, 32, 64, 128, 131** $L = 9$

   Para $Padre[1]$
   $Mut[0] \rightarrow 1, 2, 3, 5, 10, 20,$ **30, 60, 120, 130, 131** $L = 10$
   $Mut[1] \rightarrow 1, 2, 3, 5, 10, 20,$ **40, 50, 100, 120, 130, 131** $L = 11$

   Población quedaría de la siguiente manera:
   $Pop[0] \rightarrow 1, 2, 3, 6, 9, 18, 27, 54, 108, 126, 129, 131$ $L = 11$

$Pop[1] \rightarrow 1, 2, 3, 5, 8, 16, 32, 64, 128, 131$ $L = 9$

$Pop[2] \rightarrow 1, 2, 3, 5, 10, 20, 40, 80, 120, 130, 131$ $L = 10$

$Pop[3] \rightarrow 1, 2, 3, 5, 10, 20, 30, 60, 120, 130, 131$ $length = 10$

3. Seleccionar a los mejores individuos mediante torneos estocásticos
   If $Pop[0] < Pop[2]$: No
   If $Pop[0] < Pop[3]$: No
   If $Pop[0] < Pop[1]$: No. Por lo tanto contPop[0] = 0

   If $Pop[1] < Pop[0]$: Si
   If $Pop[1] < Pop[2]$: Si
   If $Pop[1] < Pop[3]$: Si. Por lo tanto contPop[1] = 3

   If $Pop[2] < Pop[1]$: No
   If $Pop[2] < Pop[0]$: Si
   If $Pop[2] < Pop[3]$: No. Por lo tanto contPop[2] = 1

   If $Pop[3] < Pop[0]$: Si
   If $Pop[3] < Pop[2]$: No
   If $Pop[3] < Pop[1]$: No. Por lo tanto contPop[3] = 1

   Ordenar la población de acuerdo a su contador de victorias (contPop) en orden descendente
   $Pop[1]$
   $Pop[2]$
   $Pop[3]$
   $Pop[0]$

   Se escoge la primera mitad para la siguiente generación, el resto es eliminado.

En el ejemplo anterior, solo se ejecuto una generación, por lo tanto, la cadena de adición de longitud mínima es: $1 \rightarrow 2 \rightarrow 3 \rightarrow 5 \rightarrow 8 \rightarrow 16 \rightarrow 32 \rightarrow 64 \rightarrow 128 \rightarrow 131$ con longitud = 9. Para exponentes más grandes se necesita realizar mayor número de generaciones, pero el proceso es el mismo.

5.6. Modificación para resolver problemas con exponentes grandes

Derivado de experimentos preliminares, se observó que el algoritmo basado en EP se queda atrapado en óptimos locales al resolver problemas con exponentes grandes, es decir, el algoritmo propuesto no logra encontrar cadenas de adición de longitud menor que las generadas por algoritmos deterministas para exponentes mayores de 128 bits. Por ende, se propone un ajuste similar que en [8] para resolver problemas con exponentes grandes, el cual se describe a continuación.

El método de desplazamiento de ventana (SWM) descrito en la Sección 4.1.3.1 es usado para encontrar cadenas de adición de longitud mínima para exponentes grandes (de 128-bits hasta 1024-bits). Se propone utilizar EP dentro del proceso de dicho método, específicamente en la generación de secuencias de adición las cuales son usadas en una parte del funcionamiento de SWM.

El algoritmo 24 describe el proceso para encontrar secuencias de adición mínimas para generar cadenas de adición de exponentes grandes donde en el renglón 5 se incorpora el algoritmo de EP. max_MSW es el tamaño máximo de la primera ventana (la ventana más significativa). q es el número máximo de ceros consecutivos permitidos para generar ventanas de no-cero (NZ).

Algoritmo 24 EP_Generador_Secuencias_Adición

Entrada: Enteros $e = (e_{m-1} \ldots\ e_1\ e_0)$ $, max_MSW,\ q$
Salida: Secuencia de adición $Seq = (1, 2, \ldots, m)$

1: Descomponer e en h ventanas Z y NZ W_i
2: Representar en decimal las ventanas NZ
3: Establecer $MSW = W_0$
4: Ordenar en forma ascendente las ventanas NZ $List = \{W_0, W_1, \ldots, W_{NZ-1}\}$
5: Establecer $U = EP_Modificado(MSW)$
6: Selecciona un elemento $a\ \in\ U$ tal que
$a\ >\ W_{NZ-2}$
7: Establecer $List = \{W_0, W_1, W_{NZ-1}, a\}$
8: Establecer $List = Generador_Secuencias_Adición$(List)
9: Establecer $Seq = List\ +\ U$

El siguiente es un ejemplo utilizando el algoritmo de EP propuesto para encontrar secuencias de adición mínimas para exponentes grandes. Siguiendo el proceso mencionado en el Algoritmo 24.

Sea $e = 166666400466134139879501284668646024975$ donde e es un número de 128-bits, $max_MSW = 14$ y $q = 2$.

1. Dividir la representación binaria de e ventanas Z y NZ.

$$\underbrace{11111010110001}_{16049}\ 0\ \underbrace{11}_{3}\ 000\ \underbrace{111111}_{63}\ \underbrace{11}_{3}\ 000\ \underbrace{1010101}_{85}\ 0$$

$$\underbrace{1}_{1}\ 00\ \underbrace{1}_{1}\ 000\ \underbrace{1}_{1}\ 0000\ \underbrace{111}_{7}\ 0000\ \underbrace{1011}_{11}\ 00\ \underbrace{1}_{1}\ 0\ \underbrace{110101}_{53}$$

$$00\ \underbrace{1011}_{11}\ 000000\ \underbrace{110101}_{53}\ \underbrace{101}_{5}\ 000\ \underbrace{1}_{1}\ 000\ \underbrace{101}_{5}\ 00\ \underbrace{11101}_{29}$$

$$0\ \underbrace{111}_{7}\ 00\ \underbrace{111}_{7}\ 0000\ \underbrace{1111}_{15}$$

2. Establecer $MSW = 16049$
3. Ordenar $W \rightarrow List = \{1, 3, 5, 7, 11, 15, 29, 53, 63, 85\}$
4. $U =EP_Modificado(16049)$ ver Algoritmo 23
$U = 1 \rightarrow 2 \rightarrow 4 \rightarrow 8 \rightarrow 16 \rightarrow 32 \rightarrow 48 \rightarrow 80 \rightarrow 160 \rightarrow 320 \rightarrow 640 \rightarrow 1280 \rightarrow 2560 \rightarrow 5120 \rightarrow 7680 \rightarrow 15360 \rightarrow 16000 \rightarrow 16048 \rightarrow 16049$

5. Establecer $a = 160$, porque $160 \in U$ y $160 > 85$

6. Añadir a a $List = \{1, 3, 5, 7, 11, 15, 29, 53, 63, 85, \underline{160}\}$

7. Establcer $List =$ *Generador_Secuencias_Adición(List)* ver Algoritmo 16
   $List = \{1, 2, 3, 5, 7, 10, 11, 12, 15, 24, 29, 53, 63, 75, 85, 160\}$

8. Establecer $Seq = List + U$ Donde los elemento se U son tomados de a en adelante
   $Seq = \{1, 2, 3, 5, 7, 10, 11, 12, 15, 24, 29, 53, 63, 75, 85, 160, 320, 640, 1280, 2560, 5120, 7680, 15360, 16000, 16048, 16049\}$
   con longitud = 25

El resto del proceso es realizado por SWM el cual se ilustra en la Figura 5.4 y se describe a detalle en la Sección 4.1.3.1.

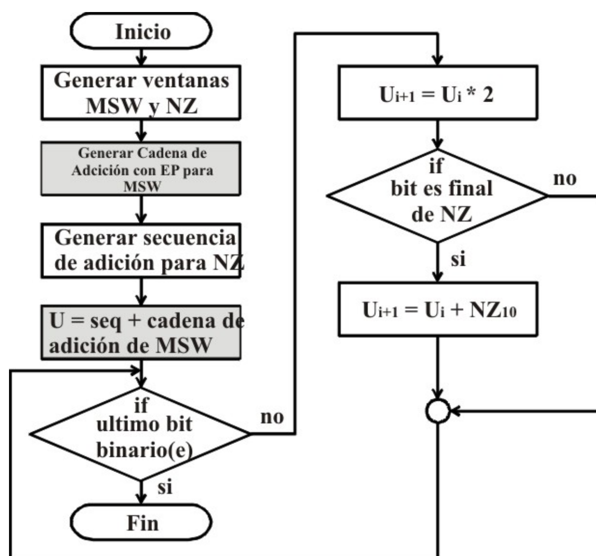

Fig. 5.4: Diagrama de Flujo del EP_SWM

Dado el conjunto {1, 3, 5, 7, 11, 15, 29, 53, 63, 85, 16049} con el Algoritmo 16 (versión original para calcular secuencias de adición) se obtiene una secuencia de adición de longitud = 118, mientras que con la propuesta descrita en el Algoritmo 24 (donde se usa el algoritmo de EP para generar secuencias de adición) se obtiene una secuencia de adición de longitud = 25.

Capítulo 6

Experimentos y Resultados

En este capítulo se describe la manera en que se probó la calidad y eficiencia del algoritmo de programación evolutiva (EP) para la búsqueda de cadenas de adición de longitud mínima. Así mismo se hace una descripción de los conjuntos de experimentos realizados y los resultados obtenidos.

Cinco experimentos, tomando como base las formas de prueba encontradas en la literatura especializada, fueron diseñados. En cada uno, se hizo una comparación contra los resultados de los algoritmos estocásticos del estado-del-arte descritos en el Capítulo 4, excepto el GA [9] pues sus resultados son claramente superados por el resto de los algoritmos estocásticos.

1. Pruebas con cadenas de adición acumuladas de conjuntos de exponentes "pequeños".
2. Pruebas con exponentes "difíciles" de optimizar
3. Pruebas con un conjunto de exponentes "diversos"
4. Pruebas con exponentes "grandes"
5. Pruebas con cadenas de adición Euclidianas.

Exponentes "pequeños", "difíciles", "diversos" y "grandes" (hasta de 1024-bits) fueron usados en los experimentos con el objetivo de probar el algoritmo EP en diferentes espacios de búsqueda. La calidad (la mejor solución alcanzada) y la consistencia (mejores valores en la mediana) se obtuvieron en un conjunto de corridas independientes que fueron consideradas como criterio de desempeño. Así mismo, se realizaron pruebas con cadenas de adición Euclidianas. Además, los resultados obtenidos fueron evaluados con pruebas estadísticas, tales como Wilcoxon y Prueba T.

Para el experimento 4 se ocuparon números de 128-bits, 256-bits, 512-bits y 1024-bits. Dichos resultados fueron comparados con los publicados en [8, 18].

Los parámetros usados en el algoritmo de EP fueron los siguientes:

- Tamaño de población $n = 100$
- Número de generaciones MAXGEN = 230
- Número de mutantes $t = 4$
- Número de individuos por encuentros $q = 10$
- Probabilidad de Doblado de números $f = 0.7$
- Probabilidad de suma de números previos $g = 0.2$

Con base en los valores de parámetros antes mencionados el algoritmo EP realiza $92,000$ evaluaciones por corrida. Ver Tabla 6.1 donde se incluyen también los números de evaluaciones reportados por los algoritmos usados en el comparativo.

Técnica	Evaluaciones		
EP	92,000	100x230x4	Indiv x gene x muta
REPLS-GA [31]	240,000	200x300x4	Indiv x gene x muta
PSO [24]	300,000	30x10000	Partículas x iteraciones

Tabla 6.1: Evaluaciones realizadas por cada meta-heurística del estado del arte

6.1. Experimento 1: cadenas de adición acumuladas

El primer experimento consistió en calcular el total de cadenas de adición acumuladas para un conjunto dado de exponentes "pequeños". Una cadena de adición acumulada (T) para un valor máximo Z, representa la suma de todas las longitudes de las cadenas de adición obtenidas de los exponentes [1, 2, ..., Z], se representa en la Ecuación 6.1

$$T(Z) = \sum_{i=1}^{Z} EP_Modificado(i) \tag{6.1}$$

donde EP_Modificado(i) es la longitud de la cadena de adición para el exponente i generada por el algoritmo correspondiente (en este caso EP). Por lo tanto un menor valor $T(Z)$ representa un mejor desempeño del algoritmo.

Como se observa en [8, 24, 31], los siguientes intervalos de exponentes e fueron probados: $e \in [1, 512]$, $e \in [1, 1000]$, $e \in [1, 2000]$, $e \in [1, 2048]$, y $e \in [1, 4096]$. 30 corridas independientes para cada conjunto de exponentes fueron ejecutadas con los valores de los parámetros mencionados al inicio de este capítulo. Los resultados estadísticos son resumidos en la Tabla 6.2, donde se puede notar que EP tiene un desempeño robusto, es decir, el mejor valor obtenido, el promedio y la mediana están muy cerca y la desviación estándar es relativamente pequeña.

$e \in$	Mejor	Promedio	Mediana	Peor	Desv.Est
[1,512]	4924	4924.10	4924	4925	0.305
[1,1000]	10808	10810.21	10810	10813	0.857
[1,1024]	11115	11118.03	11118	11120	1.402
[1,2000]	24076	24080.73	24080	24084	2.282
[1,2048]	24745	24750.47	24750	24754	3.067
[1,4096]	54497	54505.93	54507	54515	5.444

Tabla 6.2: Resultados estadísticos del primer experimento con EP.

En la Tabla 6.3 se presenta una comparación de los mejores resultados obtenidos con EP contra los resultados reportados en AIS [8], REPLS-GA [31], y por PSO [24].

Se puede observar que el EP fue capaz de encontrar la longitud óptima en el primer conjunto de exponentes (los números en negrita en la Tabla 6.3). Además, se obtuvieron mejores resultados en el resto de cadenas de adición acumuladas con respectos a los obtenidos por AIS y REPLS-GA.

Cabe mencionar que se aplicó la prueba "T" a los resultados obtenidos por cada algoritmo. Dicha comparación se hizo tomando la desviación estándar y el número de corridas para cada familia de exponentes publicados en [8, 31, 24]. En la Tabla 6.3, (+) significa que existe una diferencia significativa entre EP y el algoritmo comparado. (-) indica una diferencia no significativa. Se uso en este caso la prueba "T" porque no se contó con las muestras de corridas de todos los algoritmos comparados.

Los resultados generales en la Tabla 6.3 muestran que EP tiene un rendimiento competitivo con respecto a los algoritmos del estado del arte. Otra ventaja de EP con respecto a REPLS-GA [31] y PSO [24] es que el algoritmo propuesto requiere $92,000$ evaluaciones, mientras REPLS-GA y PSO ejecutan $240,000$ y $300,000$, respectivamente. El AIS no reporta número de evaluaciones.

Para el mismo experimento, se incrementó el número de evaluaciones para el algoritmo de EP. Para ello se aumentó el número de generaciones $MAXGEN$ y el número de individuos n a 300 y 200, respectivamente. Los resultados obtenidos se reportan en la Tabla 6.4, donde se observa una mejora en los resultados en los

$e \in$	Opt.	AIS [8]	REPLS-GA [31]	PSO [24]	EP
[1,512]	4924	4924 (+)	4924 (-)	...	**4924**
[1,1000]	10808	10813 (+)	10809 (+)	...	**10808**
[1,1024]	11115	11120 (+)	...	11120 (+)	**11115**
[1,2000]	24063	24108 (+)	24076 (+)	...	24076
[1,2048]	24731	24778 (+)	24748 (+)	...	24745
[1,4096]	54425	54617 (+)	54487 (+)	...	54497

Tabla 6.3: Resultados obtenidos de cadenas de adición acumuladas y comparación contra AIS [8], REPLS-GA [31] y PSO [24]

conjuntos [1,2000], [1,2048] y [1,4096] (columna 6 en itálicas) en comparación con los reportados en [8, 31, 24].

$e \in$	Opt.	AIS [8]	REPLS-GA [31]	PSO [24]	EP
[1,512]	4924	4924 (+)	4924 (-)	...	**4924**
[1,1000]	10808	10813 (+)	10809 (+)	...	**10808**
[1,1024]	11115	11120 (+)	...	11120 (+)	**11115**
[1,2000]	24063	24108 (+)	24076 (+)	...	*24070*
[1,2048]	24731	24778 (+)	24748 (+)	...	*24737*
[1,4096]	54425	54617 (+)	54487 (+)	...	*54454*

Tabla 6.4: Resultados obtenidos por EP con 240000 evaluaciones para cadenas de adición acumuladas y comparación con AIS [8], GA [31] y PSO [24]. En negritas se remarcan los resultados en el óptimo. En itálicas se indican los mejores resultados sin llegar al óptimo.

6.2. Experimento 2: exponentes difíciles

El segundo experimento incluye un conjunto de exponentes conocidos como "difíciles" de optimizar. Esto es porque con métodos tradicionales (deterministas) no se logran cadenas de adición mínimas [8, 24, 31]. Las comparaciones se basan en los mejores resultados obtenidos de 30 corridas independientes por exponente de EP contra los resultados reportados por REPLS-GA en [31] utilizando los mismo parámetros que el primer experimento. AIS [8] y PSO [24] no se reportan soluciones para tales exponentes. Los mejores resultados se presentan en las Tablas 8.1 y 8.2 del Anexo 8.1.

Los resultados de ambas heurísticas (REPLS-GA [31] y EP) mostrados en las Tablas 8.1 y 8.2 en el Anexo 8.1 son similares. Sin embargo el algoritmo de EP requiere $92,000$ evaluaciones mientras que el REPLS-GA utiliza $240,000$ evaluaciones para llegar a tales resultados. Aunado a ello, el algoritmo de EP se

probó con sólo $25,000$ evaluaciones y los resultados no variaron en significativamente, obteniendo 11 de 20 exponentes con longitud de 27 y para los restantes 9 exponentes, la longitud de las cadenas de adición encontradas fue de 28.

Se aplicó el Test de Wilcoxon para comprobar si existían diferencias significativas en las muestras de los experimentos de ambos algoritmos (REPLS-GA [31] y EP) pues para ambos algoritmos se contaron con sus muestras de corridas. Los resultados de la prueba estadística mencionada muestran diferencias significativas entre EP y REPLS-GA en 11 de los 20 exponentes de este experimento.

6.3. Experimento 3: exponentes diversos

En el tercer experimento, un conjunto de 28 exponentes "diversos" fueron resueltos con el algoritmo EP y los resultados fueron comparados contra los obtenidos por el AIS [8] y PSO [24]. El REPLS-GA [31] no es considerado en esta comparación porque no reporta resultados para este conjunto de exponentes. Los mejores resultados de los tres enfoques son presentados en las Tablas 8.3 y 8.4 del Anexo 8.2. Los parámetros son los mencionados al inicio del Capítulo. Se ejecutaron 30 corridas independientes por cada exponente.

Los resultados en las Tablas 8.3 y 8.4 (véase Anexo 8.2) indican que el algoritmo de EP iguala a los generados por AIS y PSO en 27 de 28 exponentes y para el exponente 3585 obtiene una cadena de adición más corta. Para este experimento EP resultó competitivo en comparación con el AIS [8] y PSO [24]. Vale la pena recordar que el PSO reporta $300,000$ evaluaciones por corrida.

6.4. Experimento 4: exponentes grandes

En el cuarto experimento se comparan los mejores resultados obtenidos por el método de deslizamiento de ventana con EP (EP_SWM) contra el SWM tradicional reportado en [18] y la adaptación de AIS publicado en [8]. Para este experimento se usaron 10 exponentes grandes aleatorios con $m = 128$, 256, 512 and 1024 bits, donde m es la representación binaria del exponente e. La comparación se basa en el mejor resultado de 30 corridas independientes por exponente.

En la Tabla 6.5 k es el tamaño de la ventana usado por SWM [18], MSW significa el valor máximo de la ventana más significativa y q es el máximo valor de ceros consecutivos para formar una ventana, usado en AIS_SWM [8] y EP_SWM.

En la Tabla 6.5 se puede notar que EP_SWM genera mejores cadenas de adición que el SWM tradicional publicado en [18]. Esto prueba que EP ayuda

m	SWM [18]		AIS_SWM [8]			EP_SWM		
	length	k	length	MSW	q	length	MSW	q
128	156	4	153	17	2	154	8	2
256	308	4	304	13	2	*304*	6	2
512	607	5	604	11	2	*604*	17	2
1024	1195	5	1196	6	5	**1190**	14	2

Tabla 6.5: Resultados de EP_SWM para exponentes grandes y comparación contra SWM [18] y AIS_SWM [8]. En negritas se marcan los mejores resultados. En itálicas se muestran resultados similares a los previamente reportados.

significativamente a SWM en la creación de cadenas de adición para exponentes grandes.

Por otro lado, comparando AIS_SWM contra EP_SWM, el algoritmo propuesto genera resultados similares en exponentes con $m = 256$ y 512, pero en exponentes con $m = 1024$ EP_SWM obtiene mejores resultados. Sólo en exponentes con $m = 128$ AIS_SWM es mejor. Sin embargo se puede concluir que EP_SWM genera resultados competitivos para exponentes grandes.

6.5. Experimento 5: cadenas de adición Euclidianas

En el quinto experimento se generaron cadenas de adición Euclidianas (EACs) con el algoritmo de EP para "diversos" exponentes y mostrar su rendimiento. En las Tablas 8.5 y 8.6 del Anexo 8.3 se presentan los resultados de las EACs.

Para este experimento se usaron los parámetros descritos al inicio del Capítulo con excepción de los parámetros f y g, porque un EAC no permite el doblado de números, por lo tanto $f = 0.0$ y $g = 0.7$

En las Tablas 8.5 y 8.6 del Anexo 8.3 se puede notar que los resultados no varían mucho con respecto al experimento 3, a pesar de no haber permitido doblado de números en la generación de las cadenas de adición.

No se reportan comparativos en este experimento pues ninguno de los métodos estocásticos ha reportado resultados en cadenas de adición Euclidianas.

Capítulo 7

Conclusiones y Trabajo Futuro

El problema presentado en esta investigación es el uso de un algoritmo de programación evolutiva para resolver el problema de reducción de longitud de las cadenas de adición. Con la intensión de disminuir el número de multiplicaciones y divisiones que se ejecutan en la exponenciación modular, usada en los algoritmos de cifrado asimétrico en Criptografía. El algoritmo de EP se basa únicamente en un operador de mutación y un proceso de sustitución estocástico para sesgo de la búsqueda de soluciones competitivas. En general EP requirió de menos evaluaciones con respecto a algunos algoritmos inspirados en la naturaleza mencionados en la literatura especializada. El algoritmo está diseñado para tratar sólo con soluciones factibles a partir de su población inicial. Además el algoritmo propuesto es rápido de implementar, debido a que no se utiliza el operador de cruza tal como el GA, por lo tanto se generan pocas líneas de código y se ahorra tiempo en codificación.

El algoritmo de EP fue probado en cinco experimentos con diferentes tipos de exponentes "pequeños", "diversos", "difíciles" y "grandes". En cuanto a las cadenas de adición acumuladas para exponentes " pequeños", EP fue capaz de proporcionar resultados competitivos y mejores en algunos casos con sólo 30 % de las evaluaciones requeridas por dos de los algoritmo del estado del arte. Este comportamiento también se observó en dos grupos, uno de los "difíciles" y otro de "diversos" exponentes, donde el algoritmo EP obtuvo resultados similares con respecto al REPLS-GA [31] en el primer conjunto y con respecto al AIS [8] y PSO [24] en el segundo conjunto de exponentes, todos ellos con un menor número de evaluaciones.

Además, EP fue adaptado para encontrar un mínimo de cadenas de longitud para exponentes grandes (128, 256, 512 y 1024 bits) obteniendo mejores resultados en exponentes de 1024-bits.

Por último, se generaron cadenas de adición Euclidianas con el algoritmo de EP para "diversos" exponentes, obteniendo un buen rendimiento con respecto a los resultados del experimento tres.

El trabajo futuro de esta tesis consiste en implementar el algoritmo de EP en algún sistema de cifrado de llave-pública, tal como el RSA o DSA, con el motivo de medir el desempeño del criptosistema usando EP como generador de cadenas de adición de longitud mínima, durante el proceso de cifrado y descifrado de datos. Finalmente, se planea probar el algoritmo para generar cadenas de adición Euclidianas con exponentes grandes.

Capítulo 8

Anexos

8.1. Tablas del experimento 2

e	Cadena de adición	REPLS-GA [31]	EP
3243679	1-2-3-5-10-20-40-80-83-166-206 412-824-1648-3296-6592-9888 19776-39552-79104-158208-316416 632832-632915-1265830-2531660 3164575-3243679	27	27
3493799	1-2-3-4-8-16-32-64-128-256 512-515-1030-2060-4120-8240 16480-32960-65920-131840 263680-263683-527366-1054732 2109464-3164196-3427879-3493799	27	27
3459835	1-2-4-5-10-20-25-35-70-140-280-560 1120-2240-4480-8960-17920-35840 71680-71705-143410-286820-573640 1147280-1147305-2294610-3441915 3459835	27	27
3235007	1-2-3-6-12-24-27-54-108-162-324 648-1296-1944-3888-7776-15552 31104-62208-124416-248832-248859 497691-995382-1990764-2986146 3235005-3235007	27	27
3230591	1-2-3-4-8-16-32-64-128-131-262 524-1048-2096-4192-8384-16768 33536-67072-134144-268288-536576 538672-538803-1077606-2155212 2694015-3230591	27	27
3182555	1-2-3-6-9-18-36-72-144-288-576 1152-1728-2880-5760-11520-23040 46080-92160-184320-187200-187209 374418-748836-1497672-2995344 3182553-3182555	27	27
3440623	1-2-4-8-16-32-48-96-192-193 386-772-773-1546-3092-6184 12368-24736-49472-50245-99717 199434-398868-797736-1595472 3190944-3390378-3440623	27	27
3926651	1-2-4-5-9-18-36-72-144-288 576-1152-2304-4608-9216-18432 18437-36869-73738-92175-184350 368700-737400-811138-1548538 3097076-3908214-3926651	27	27
3234263	1-2-3-6-12-24-48-96-192-194 388-776-1552-3104-3107-3155 6310-12620-25240-50480-100960 201920-403840-807680-1615360 3230720-3233875-3234263	27	27
3352927	1-2-4-8-16-17-34-68-136-272 544-1088-2176-4352-8704-17408 34816-69632-139264-278528 557056-1114112-1114656-1114724 1114741-2229482-3344223-3352927	27	27

Tabla 8.1: Resultados del segundo experimento: mejores resultados obtenidos por REPLS-GA[31] y EP para exponentes “difíciles” (1 de 2)

e	Cadena de adición	REPLS-GA [31]	EP
3704431	1-2-4-8-9-18-36-72-108-180-360 720-1440-2880-5760-5761-11522 17283-28805-57610-115220-230440 460880-921760-1843520-3687040 3704323-3704431	27	27
3922763	1-2-3-6-12-24-26-52-104-208 416-832-1664-3328-3331-6659 9990-19980-39960-79920-159840 163171-326342-652684-1305368 1958052-3916104-3922763	27	27
2948207	1-2-4-8-12-24-48-96-192-384 768-1536-3072-6144-12288-12289 24578-49156-98312-98696-197392 394784-407073-814146-1628292 2442438-2849511-2948207	27	27
3093839	1-2-3-5-10-15-30-60-75-150 151-302-604-1208-2416-4832-9664 19328-38656 77312-154624-309248 618496-1236992-2473984-3092480 3093688-3093839	27	27
3243931	1-2-4-8-16-32-64-128-256-512 513-1026-2052-4104-8208-16416 16480-32960-65920-66433-132353 264706-529412-1058824-2117648 3176472-3242905-3243931	27	27
3325439	1-2-3-6-12-24-48-96-98-196-392 784-1568-3136-6272-12544-25088 50176-50182-100364-100367-101151 201518-403036-806072-1612144 3224288-3325439	27	27
3190511	1-2-4-8-9-17-34-68-136-272-340 680-1360-2720-5440-10880-10889 21778-43556-87112-174224-348448 359337-707785-1415570-2831140 3190477-3190511	27	27
3287999	1-2-3-5-10-15-25-50-100-200-400 800-1600-3200-6400-6403-12806 25612-51224-102448-204896-409792 819584-819599-1639198-1645601 3284799-3287999	27	27
3266239	1-2-4-8-16-32-64-128-256-512 1024-2048-4096-4098-8196-16392 16393-32785-49178-98356-196712 393424-786848-1573696-3147392 3245748-3262141-3266239	27	27
3167711	1-2-3-5-10-15-30-60-120-240-480 960-1920-1922-3844-7688-15376 23064-46128-92256-184512-184527 369054-372898-745796-1491592 2983184-3167711	27	27

Tabla 8.2: Resultados del segundo experimento: mejores resultados obtenidos por REPLS-GA[31] y EP para exponentes “difíciles” (2 de 2)

8.2. Tablas del experimento 3

e	Cadena de adición	AIS [8]	PSO [24]	EP
5	1-2-3-5	3	3	3
7	1-2-4-5-7	4	4	4
11	1-2-4-8-10-11	5	5	5
19	1-2-4-8-16-18-19	6	6	6
29	1-2-4-8-12-20-28-29	7	7	7
47	1-2-3-5-10-20-40-45-47	8	8	8
71	1-2-4-8-16-32-64-68-70-71	9	9	9
127	1-2-3-6-12-24-48-72-120-126-127	10	10	10
191	1-2-4-5-10-20-30-60-120-180-181-191	11	11	11
379	1-2-3-6-9-18-36-72-144-288-360-378-379	12	12	12
607	1-2-3-6-12-24-48-96-192-384 576-600-606-607	13	13	13
1087	1-2-3-6-12-24-27-54-108-216-432-864 1080-1086-1087	14	14	14
1903	1-2-3-5-10-13-26-52-104-208-416-832 1664-1669-1877-1903	15	15	15
3585	1-2-4-8-16-32-64-128-256-512-1024 2048-3072-3584-3585	16	16	**14**
6271	1-2-3-6-12-24-48-96-192-384-768-1536 3072-6144-6150-6246-6270-6271	17	17	17
11231	1-2-3-6-7-14-28-56-112-224-448-896 1792-3584-7168-10752-11200-11228 11231	18	18	18
18287	1-2-3-6-12-14-28-56-84-140-280-560 1120-2240-4480-8960-17920-18200 18284-18287	19	19	19
34303	1-2-3-6-12-15-30-60-63-126-252-504 1008-2016-4032-8064-16128-32256 34272-34302-34303	20	20	20

Tabla 8.3: Resultados del tercer experimento: mejores resultados obtenidos por AIS[8], PSO [24] y EP en un conjunto de “diversos” exponentes(1 de 2)

e	Cadenas de adición	AIS [8]	PSO [24]	EP
65131	1-2-3-6-8-14-28-56-112-224-448-896 1792-3584-3612-7224-10836-21672 43344-65016-65128-65131	21	21	21
110591	1-2-3-5-10-20-40-80-90-170-340-680 1020-1700-3400-6800-13600-27200 54400-108800-110500-110590-110591	22	22	22
196591	1-2-3-6-12-24-30-60-120-240-360-720 1440-2880-5760-11520-23040-46080 92160-184320-195840-196560-196590 196591	23	23	23
357887	1-2-3-5-10-20-40-43-86-172-344-688 1376-2752-5504-11008-22016-44032 88064-176128-352256-357760-357846 357886-357887	24	24	24
685951	1-2-3-6-12-24-48-96-102-204-408-510 1020-2040-4080-8160-16320-32640-48960 97920-146880-293760-587520-685440 685950-685951	25	25	25
1176431	1-2-3-5-10-20-40-80-160-320-640-1280 2560-5120-10240-20480-21760-21762 43524-65286-130572-261144-522288 1044576-1175148-1176428-1176431	26	26	26
2211837	1-2-3-6-12-13-25-28-53-106-212-424 848-1696-3392-6784-13568-27136-40704 67840-135680-271360-271385-542770 1085540-2171080-2211784-2211837	27	27	27
4169527	1-2-3-6-9-18-36-42-84-168-336-672 1344-1345-2690-5380-5422-10844-21688 43376-86752-173504-347008-694016 1388032-2776064-4164096-4169518 4169527	28	28	28
7924319	1-2-3-6-12-24-36-42-84-168-336-672 1344-2688-5376-10752-21504-43008 86016-86017-172034-344068-688136 1032204-1720340-3440680-6881360 7913564-7924316-7924319	29	29	29
14143037	1-2-3-6-9-18-36-54-108-216-432-864 1728-3456-3564-7128-14256-28512-57024 114048-228096-456192-912384-1368576 1368684-2737368-2737369-5474738 10949476-13686845-14143037	30	30	30

Tabla 8.4: Resultados del tercer experimento: mejores resultados obtenidos por AIS[8], PSO [24] y EP en un conjunto de "diversos" exponentes(2 de 2)

8.3. Tablas del experimento 5

e	Cadena de adición Euclidiana	EP
5	1-2-3-5	3
7	1-2-4-5-7	4
11	1-2-3-4-7-11	5
19	1-2-3-5-7-12-19	6
29	1-2-3-5-8-13-21-29	7
47	1-2-3-5-8-13-21-34-47	8
71	1-2-3-5-7-10-17-27-44-71	9
127	1-2-4-5-9-13-22-35-57-92-127	10
191	1-2-3-5-6-11-17-28-45-73-118-191	11
379	1-2-3-5-8-13-21-34-55-89-144-233-377-379	13
607	1-2-4-5-9-14-23-37-51-88- 139-227-366-593-607	14
1087	1-2-4-6-10-16-17-33-50-83- 84-167-251-418-669-1087	15
1903	1-2-3-5-8-13-21-22-43-64- 107-171-278-449-727-1176-1903	16
3585	1-2-3-5-8-13-21-29-50-71-121- 192-313-505-818-1323-2141-3464-3585	18
6271	1-2-4-6-10-16-17-33-50-83- 133-216-349-565-914-1479-2393 3872-6265-6271	19
11231	1-2-3-5-8-13-21-26-47-73-120- 193-313-506-699-1205-1904-3109- 5013-8122-11231	20
18287	1-2-4-6-10-16-20-36-56-92-148- 240-241-481-722-1203-1925-3128- 5053-8181-13234-18287	21
34303	1-2-4-5-6-11-17-28-45-73-118- 191-309-500-691-1191-1882-3073- 4955-8028-12983-21011-33994-34303	23

Tabla 8.5: Cadena de adición Euclidiana usando el algoritmo EP en un conjunto de “diversos” exponentes (1 de 2)

e	Cadena de adición Euclidiana	EP
65131	1-2-4-6-10-11-21-32-53-85-138- 223-361-584-945-1529-2474-4003-6477- 8951-15428-24379-39807-64186-65131	24
110591	1-2-3-5-8-13-21-34-55-89-144, 233-377-610-987-1597-2584-4181- 6765-10946-17711-28657-46368-64079- 110447-110591	25
196591	1-2-3-5-8-13-21-34-55-89-144-233-377- 610-987-988-1975-2963-4938-7901-12839- 20740-33579-54319-87898-142217-196536- 196591	27
357887	1-2-4-6-10-16-26-42-68-94-162- 256-418-674-1092-1766-2858-4624- 7482-12106-19588-31694-43800-75494- 119294-119298-238592-357886-357887	28
685951	1-2-3-5-8-13-21-34-55-89-144-233-377- 610-987-1597-2584-4181-6765-10946- 17711-28657-28890-46601-75491-122092- 197583-244184-441767-685951	29
1176431	1-2-3-5-8-13-21-34-55-89-144-233-377- 380-757-1137-1894-3031-4925-7956-12881 20837-33718-54555-88273-142828-231101 373929-428484-802413-1176342-1176431	31
2211837	1-2-4-6-10-16-26-42-68-110-178-288-466 754-932-1686-2618-4304-6922-11226- 18148-29374-47522-76896-124418-201314- 230688-432002-662690-673916-1105918- 1105919-2211837	32
4169527	1-2-4-6-7-13-20-33-53-86-139-225-364- 589-953-1542-2495-4037-6532-10569- 17101-27670-44771-72441-117212-189653- 306865-496518-498060-994578-1492638- 2487216-3979854-4169507-4169527	34
7924319	1-2-3-5-8-13-21-34-55-89-144-233-377- 610-987-1597-2584-4181-6765-6909-13674 20583-34257-54840-89097-143937-233034 233089-466123-699212-1165335-1864547- 3029882-4894429-7924311-7924319	35
14143037	1-2-3-4-7-11-18-29-47-76-123-199-322- 521-843-1364-1371-2735-4106-6841-10947 17788-28735-46523-75258-121781-197039- 318820-515859-834679-1350538-2185217- 3535755-3535758-7071513-7071524- 14143037	36

Tabla 8.6: Cadena de adición Euclidiana usando el algoritmo EP en un conjunto de "diversos" exponentes (2 de 2)

Bibliografía

[1] Varios Autores. Congruencias. Technical report, Enero 2004. Depto. de Matemática Aplicada, Fac. de Informática, U. Politécnica de Madrid. http://www.dma.fi.upm.es/java/matematicadiscreta/-aritmeticamodular/congruencias.html#4 Fecha de ingreso: 12/Oct/2010.

[2] Thomas Bäck. *Evolutionary Algorithms in Theory and Practice*. Oxford University Press, New York, 1996.

[3] Hans-Georg Beyer. *The theory of Evolution Strategies*. Springer, Berlin, 2001.

[4] Cristian Borghello. Seguridad informática. criptología. Technical report, Diciembre 2009. http://www.segu-info.com.ar/criptologia/criptologia.htm, Última fecha de ingreso: 03/Dic/2010.

[5] Jurjen Bos and Matthijs Coster. Addition chains heuristics. *Centrum voor Wiskunde en Informatica, Kruislaan 413, 1098 SJ Amsterdan, The Netherlands.*, Springer-Verlag:1–5, 1998.

[6] Nareli Cruz-Cortés. Handling constraints in global optimization using artificial immune systems: A survey. In Efrén Mezura-Montes, editor, *Constraint-Handling in Evolutionary Optimization*, volume 198 of *Studies in Computational Intelligence*, pages 237–262. Springer Berlin / Heidelberg.

[7] Nareli Cruz-Cortés and Carlos A. Coello-Coello. Un sistema inmune artificial para solucionar problemas de optimización multiobjetivo. Technical report, Departamento de Ingeniería Eléctrica Sección de Computación, Instituto Politécnico Nacional, México DF., 2003.

[8] Nareli Cruz-Cortés, Francisco Rodríguez-Henríquez, and Carlos A. Coello-Coello. An artificial immune system heuristic for generating short addition chains. *IEEE Transactions on Evolutionary Computation*, 12(1):1–24, February 2008.

[9] Nareli Cruz-Cortés, Francisco Rodríguez-Henríquez, Raúl Juárez-Morales, and Carlos A. Coello-Coello. Finding optimal addition chains using a genetic algorithm approach. *Lecture Notes in Computer Science, Comput-*

er Science Section, Electrical Engineering Department, CINVESTAV IPN, 2:1–8, 2005.

[10] Mentes Curiosas. 8 ejemplos de inventos inspirados en la naturaleza. http://mentescuriosas.es/8-ejemplos-de-inventos-inspirados-en-la-naturaleza/, Diciembre 2009. Última fecha de ingreso: 03/Dic/2010.

[11] M. Dorigo, V. Maniezzo, and A. Colorni. The Ant System: Optimization by a Colony of Cooperating Agents. *IEEE Transactions of Systems, Man and Cybernetics-Part B*, 26(1):29–41, 1996.

[12] Taher ElGamal. A public key cryptosystem and a signature scheme based on discrete logarithms. Technical report, Hewlett-Packard Labs 1501 Page Mill Rd Palo Alto CA 94301, 1997.

[13] Andries P. Engelbrecht. *Fundamentals of Computational Swarm Intelligence.* Wiley, 2005.

[14] Lawrence J. Fogel. *Intelligence Through Simulated Evolution. Forty years of Evolutionary Programming.* John Wiley & Sons, New York, 1999.

[15] David E. Goldberg. *Genetic Algorithms in Search, Optimization and Machine Learning.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1st edition, 1989.

[16] Jaime Gutierrez. *Protocolos criptográficos y seguridad en redes.* University of Cantabria, 1st edition, 2003.

[17] Dervis Karaboga and Bahriye Basturk. A powerful and efficient algorithm for numerical function optimization: artificial bee colony (abc) algorithm. *Journal of Global Optimization*, 39(3):459–471, 2007.

[18] Cetin Kaya-Koc. High-speed rsa implementation. Technical report, RSA Laboratories, Redwood City, CA, 1994.

[19] Donald E. Knuth. *The art of computer programing.* Addision-Wesley, second edition, 1981.

[20] John R. Koza. *Genetic Programming II: Automatic Discovery of Reusable Programs.* MIT Press, 1994.

[21] Sander Van-Der Kruijssen. Addition chains, efficient computing of powers. *Bachelor Proyect, Amsterdam*, 1:13–50, 2007.

[22] Noboru Kunihiro and Hirosuke Yamamoto. Window and extended window methods for addition chain and addition-subtraction chain. *Special Section on Cryptography and Information Security*, 8:60 – 61, Enero 1998. IEICE Trans Fundamentals.

[23] LaReserva.com. Innovaciones inspiradas en la naturaleza. http://www.lareserva.com/home/innovaciones_inspiradas_naturaleza-_modelo_inventos, Octubre 2010. Última fecha de ingreso: 03/Dic/2010.

[24] Alejandro León-Javier, Nareli Cruz-Cortés, Marco A. Moreno-Armendáriz, and Sandra Orantes-Jiménez. Finding minimal addition chains with a particle swarm optimization algorithm. *Lecture Notes in Computer Science*, 5845/2009:680–691, 2009.

[25] Eduard Llull. Seguridad informática. criptología. Technical report, Octubre 2005. http://aleph.llull.net/2005/10/15/exponenciacion-modular/, Última fecha de ingreso: 03/Dic/2010.

[26] Jorge M. López. Criptografía. Technical report, http://www.matematicaparatodos.com/varios/criptografia.pdf, 2007.

[27] Manuel José Lucena-López. Criptografía y seguridad en computadores. Technical report, Departamento de informática. Escuela Politécnica Superior, Universidad de Jaen, España., 1999.

[28] Mario Merino-Martínez. Una introducción a la criptografía. el criptosistema r.s.a. Technical report, Bachillerato Internacional, I.E.S Cardenal López de Mendoza, 2004.

[29] Efrén Mezura-Montes and Betania Hernández-Ocaña. Modified bacterial foraging optimization for engineering design. In in Cihan H. Dagli et al. (editors), editor, *Proceedings of the Artificial Neural Networks in Enginnering Conference (ANNIE'2009)*, volume 19 of *ASME Press Series*, pages 357–364. Intelligent Engineering Systems Through Artificial Neural Networks, Noviembre 2009.

[30] Nadia Nedjah and Luiza de Macedo-Mourelle. Finding minimal addition chains using ant colony. In *Proceedings of the international conference on intelligent data engineering and automated learning*, pages 1–6. Springer, 2004.

[31] Luis G. Osorio-Hernández, Efrén Mezura-Montes, Nareli Cruz-Cortés, and Francisco Rodríguez-Henríquez. An improved genetic algorithm able to find minimal lenght addition chains for small exponents. In *Proceedings of the IEEE Congress on Evolutionary Computation*, pages 1–6. IEEE Press, 2009.

[32] Kevin M. Passino. Biomimicry of bacterial foraging for distributed optimization and control. *IEEE Control Systems Magazine*, 22(3):52–67, 2002.

[33] K. Price, R. Storn, and J. Lampinen. *Differential Evolution: A Practical Approach to Global Optimization.* Natural Computing Series. Springer-Verlag, 2005.

[34] Michael O. Rabin. Digitalized signatures and public key functions as intractable as factorization. *Massachusetts Institute of Technology. Laboratory for Computer Science*, pages 2–16, 1979.

[35] R. L. Rivest, A. Shamir, and L. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*, pages 1–15, 1978.

[36] Rafael Sandoval Rodríguez. Programación genética. Technical report, División de Estudios de Posgrado e Investigación del Instituto Tecnológico Superior de Chihuahua, México, 2002.

[37] Boris Ryabko and Andrey Fionov. *Basics of Contemporary Cryptography for it Practitioners*. World Scientific, series on coding theory and cryptology - vol.1 edition, 2005.

[38] Hernando Manuel Quintana Ávila. Congruencias. Technical report, Febrero 2010. Instituto Tecnológico Metropolitano, Medellin-Colombia. http://latekhne.itm.edu.co/index.php?option=com_content&task=view&id=1273&Itemid=42, Ultima Fecha de ingreso: 03/Dic/10.

[39] Chia-Long Wu, Der-Chyuan Lou, and Te-Jen Chang. Investigations on fast exponentiation algorithms for rsa cryptographic appplications. In *2006 ITRI Innovation and Technology Management Conference*. Feng Chia University, Taichung, Taiwan, China, 2006.