Universidad Nacional Autónoma de México

Facultad de Ciencias

Algoritmos Evolutivos Paralelos: Análisis de Diversidad

T E S I S

QUE PARA OBTENER EL TÍTULO DE:

FÍSICO

PRESENTA:

FRANCISCO MORALES MORILLÓN

DIRECTOR DE TESIS:

DRA. KATYA RODRÍGUEZ VÁZQUEZ

2014

UNAM – Dirección General de Bibliotecas
Tesis Digitales
Restricciones de uso

DERECHOS RESERVADOS ©
PROHIBIDA SU REPRODUCCIÓN TOTAL O PARCIAL

Todo el material contenido en esta tesis esta protegido por la Ley Federal del Derecho de Autor (LFDA) de los Estados Unidos Mexicanos (México).

El uso de imágenes, fragmentos de videos, y demás material que sea objeto de protección de los derechos de autor, será exclusivamente para fines educativos e informativos y deberá citar la fuente donde la obtuvo mencionando el autor o autores. Cualquier uso distinto como el lucro, reproducción, edición o modificación, será perseguido y sancionado por el respectivo titular de los Derechos de Autor.

1. Datos del alumno
Morales
Morillón
Francisco
56 55 26 56
Universidad Nacional Autónoma de México
Facultad de Ciencias
Física
306162790

2. Datos del tutor
Dra.
Katya
Rodríguez
Vázquez

3. Datos del sinodal 1
Dr.
David Philip
Sanders

4. Datos del sinodal 2
Dr.
Gustavo
de la Cruz
Martínez

5. Datos del sinodal 3
Dr.
Iván Vladimir
Meza
Ruiz

6. Datos del sinodal 4
Dr.
Carlos
Málaga
Iguiñiz

7. Datos del trabajo escrito
Algoritmos Evolutivos Paralelos
Análisis de Diversidad
104 p
2014

Índice

Prefacio **vi**

1 Introducción **1**
1.1 El modelo de Ising . . . 1
1.2 *Spin Glass* . . . 3
1.3 Minimizando la energía de un Spin Glass . . . 5
1.4 Introducción al cómputo paralelo . . . 6
1.4.1 Principales arquitecturas paralelas . . . 9
1.4.2 Principales modelos de programación paralela . . . 12
1.4.3 *Speedup* y Eficiencia . . . 13

2 Algoritmos Genéticos **14**
2.1 ¿Qué son los algoritmos genéticos? . . . 14
2.2 Elementos básicos de un algoritmo genético . . . 16
2.2.1 Codificación . . . 16
2.2.2 Función de aptitud . . . 17
2.2.3 Mecanismos de reproducción . . . 18
2.2.4 Métodos de selección . . . 19
2.2.5 Elitismo . . . 20
2.2.6 Convergencia . . . 21
2.3 Diversidad en los algoritmos genéticos . . . 22
2.3.1 Distancia Hamming . . . 24
2.3.2 Entropía de Shannon . . . 26
2.3.3 Correlación de Spearman . . . 28
2.4 Algoritmos genéticos paralelos . . . 30
2.4.1 Clasificación de los algoritmos genéticos paralelos . . . 31
2.4.2 Topologías y esquemas de migración . . . 34

3 Implementación de los algoritmos genéticos en *Spin Glasses* **38**
3.1 Implementación del problema . . . 38
3.2 Algoritmo genético serial . . . 44
3.3 Algoritmo genético paralelo . . . 52
3.4 Equipo utilizado . . . 60

4 Resultados **61**

4.1 Eficiencia computacional 61
4.2 Diversidad y resultados de los algoritmos 64

5 Conclusiones **76**
5.1 Conclusión 76
5.2 Trabajo futuro 77

A Código fuente **78**
A.1 Algoritmo genético serial 78
A.2 Algoritmo genético paralelo 85

Bibliografía **94**

Lista de Figuras

1.1 Configuración 2
1.2 Modelo Edwards-Anderson 4
1.3 Plaquetas 5
1.4 Ejecución síncrona de instrucciones 8
1.5 Ejecución asíncrona de instrucciones 8
1.6 Arquitectura de memoria compartida 10
1.7 Arquitectura de memoria distribuida 11

2.1 Ejemplo del operador de reproducción 19
2.2 Ejemplo del operador de mutación 19
2.3 Comportamientos monotónicos y no monotónicos 29
2.4 Estructura de un algoritmo genético paralelo del tipo maestro-esclavo con una sola población 32
2.5 Estructura de un algoritmo genético paralelo con mútlples poblaciones 32
2.6 Estructura de un algoritmo genético paralelo de *grano fino* 33
2.7 Estructura de un algoritmo genético paralelo *híbrido jerárquico* combinando múltiples poblaciones con maestro-esclavo 35
2.8 Estructura de un algoritmo genético paralelo *híbrido jerárquico* combinando múltiples poblaciones con grano fino 35
2.9 Topologías de comunicación entre subpoblaciones 37

3.1 Codificación de una configuración de spin glass 39
3.2 Interacciones en un spin glass 39
3.3 Matrices de codificación para un spin glass 40
3.4 Operador de reproducción para un spin glass 41
3.5 Operador de mutación para un spin glass 42
3.6 Método de selección por torneo 42
3.7 Distancia Hamming de un spin glass 43

4.1 Tiempos de ejecución de los algoritmos 62
4.2 *Speedup* del algoritmo genético paralelo 63
4.3 Eficiencia del algortimo genético paralelo 64
4.4 Energía promedio de las mejores soluciones 66
4.5 Histograma de las mejores soluciones 66
4.6 Distancia Hamming promedio de la poblaciones de los algoritmos genéticos serial y paralelo 67

4.7 Entropía promedio de la poblaciones de los algoritmos genéticos serial y paralelo 69
4.8 Dispersiones (Energía - Distancia Hamming y Energía - Entropía) de los algoritmos genéticos serial y paralelo 70
4.9 Dispersiones en el tiempo (Energía - Distancia Hamming y Energía - Entropía) de los algoritmos genéticos serial y paralelo 71
4.10 Correlación Spearman (Energía - Distancia Hamming y Energía - Entropía) en función del tiempo de los algoritmos genéticos serial y paralelo 73
4.11 Configuraciones encontradas para los spin glasses 74

Prefacio

Los algoritmos genéticos son una herramienta muy poderosa debido a su gran flexibilidad y adaptabilidad para resolver diversos problemas, aunque al mismo tiempo estas características los convierten en sujetos complejos para su estudio.

Al implementar un algoritmo genético para resolver un problema, dependiendo de la complejidad de éste, los tiempos de ejecución pueden llegar a ser considerables, y es por esta razón que se busca reducir los tiempos de ejecución mediante la implementación paralela del algoritmo genético.

Las versiones paralelas de los algoritmos genéticos, además de reducir los tiempos de ejecución, dan lugar a modificaciones en la estructura del algoritmo que repercuten en su desempeño.

En este trabajo se estudian a los Vidrios de Espín (*Spin Glasses* en Inglés), los cuales permiten modelar materiales con interacciones ferromagnéticas y anti-ferromagnéticas. Debido a la variedad en el tipo de interacciones que se tienen, el problema de encontrar la configuración de los espines que minimiza la energía de un spin glass se convierte en un problema de optimización combinatoria de alta complejidad, lo que impide encontrar una solución de forma analítica.

Por lo anterior, se propone utilizar algoritmos genéticos, en su versión serial y paralela, como método para minimizar la energía de los spin glasses. También se busca establecer la importancia de la diversidad en los algoritmos genéticos, caracterizando su comportamiento en ambas versiones por medio de medidas como la distancia Hamming y la entropía de Shannon.

La forma en la que se abordan todos estos temas dentro de este trabajo es mediante la siguiente estructura.

El primer capítulo contiene una introducción a los spin glasses en dos dimensiones a través del modelo de Edwards-Anderson como una extensión del modelo de Ising. Se explica la complejidad del sistema y se justifica el uso de los algoritmos genéticos como método para encontrar las configuraciones de mínima energía de los spin glasses. También se introducen los conceptos básicos del cómputo paralelo, necesarios para las versiones paralelas de los algoritmos genéticos.

A lo largo del segundo capítulo se presenta una extensa introducción sobre los algoritmos genéticos. Se introducen los principales tipos de algoritmos genéticos mediante la clasificación propuesta por Cantú Paz en su libro *Efficient and Accurate Parallel Genetic Algorithms* [1], y se remarcan las principales diferencias entre las implementaciones seriales y paralelas.

El tercer capítulo está dedicado a explicar cómo se realiza la implementación del problema de los spin glasses para ambas versiones de los algoritmos genéticos (serial y paralela).

Finalmente, en los capítulos 4 y 5 se presentan los resultados y conclusiones obtenidas en este trabajo. Se muestran las comparaciones observadas tanto en el aspecto de la eficiencia computacional, como en el desempeño para encontrar soluciones optimas entre la versión serial y paralela del algoritmo.

Capítulo 1

Introducción

1.1 El modelo de Ising

Una forma para describir a los materiales magnéticos y su comportamiento en presencia de campos magnéticos externos es mediante el modelo de Ising. Este modelo se enfoca en describir el comportamiento del momento magnético de los átomos del material magnético [2].

Para describir este comportamiento, el modelo de Ising considera cómo los momentos magnéticos responden a los campos magnéticos externos y también toma en cuenta las interacciones que se presentan entre los momentos magnéticos de los átomos vecinos dentro del material. Ahora, como los momentos magnéticos tienden a alinearse o a anti-alinearse debido a las interacciones antes mencionadas, una forma práctica para representarlos es mediante el *espín de Ising*, que se define como $\sigma \in \{+1, -1\}$.

Otra parte importante en la definición del sistema en el modelo de Ising son el tamaño y la dimensión del material que se busca describir, aunque para los propósitos de esta tesis, únicamente se consideran materiales en dos dimensiones (2-D), y en particular, arreglos con una estructura cristalina cuadrada simple. Dicho lo anterior, se tiene un tipo de estructura definida, y una forma para representarla es considerar el conjunto de puntos que forman parte de la misma, pero para poder determinarlos se requiere establecer su dimensión d, y el tamaño del lado L. Así que una red cristalina cuadrada simple y con lado L se puede representar por el conjunto de puntos $\mathbb{L} = \{1, 2, \ldots, L\}^d$.

Con estos dos elementos, el conjunto de puntos $\mathbb{L}$ que describe la estructura de la red y el espín de Ising, se tiene una configuración del sistema, pues si se identifica cada punto del conjunto $\mathbb{L}$ con cada átomo que compone al material y con su espín correspondiente, esto es, $\sigma_i \in \{+1, -1\} \quad \forall i \in \mathbb{L}$, se puede generar un nuevo conjunto $\underline{\sigma} = (\sigma_1, \sigma_2, \ldots, \sigma_N)$ con $N = L^d$, el cual representa una de las 2^N posibles configuraciones del sistema, pues como cada espín tiene sólo dos posibles estados, el espacio de configuraciones de cada espín es $\mathcal{X} = \{+1, -1\}$, por lo que el espacio de configuraciones de los N espines del sistema está dado por

$$\mathcal{X}_N = \underbrace{\mathcal{X} \times \cdots \times \mathcal{X}}_{N} = \{+1, -1\}^{\mathbb{L}}, \tag{1.1}$$

de donde se tiene que $|\mathcal{X}_N| = 2^N$.

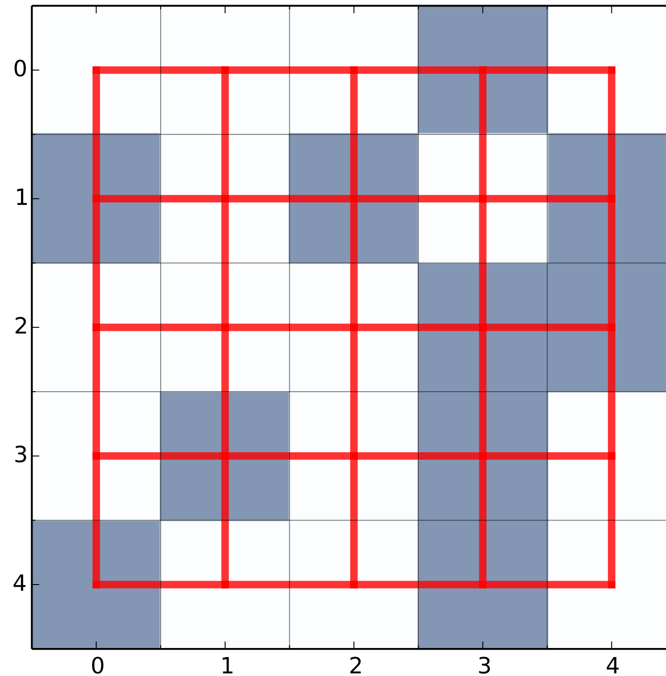


FIGURA 1.1: Una posible configuración de un sistema con $L = 5$. Los cuadrados blancos centrados en los vértices de la red (las líneas rojas) representan espines $\sigma_i = -1$, mientras que los cuadrados oscuros a espines $\sigma_i = +1$. Usando la ecuación (1.2) se tiene que la energía de esta configuración es $E(\underline{\sigma}) = 10 + 5B$.

Por último para poder determinar la energía de una configuración $\underline{\sigma}$ del sistema, se suman todas las contribuciones de energía debidas a las interacciones de los pares de espines vecinos en la red cuadrada, y también se suman las contribuciones de cada espín debida al campo magnético externo. De este modo se define la función para calcular la energía como

$$E(\underline{\sigma}) = -\sum_{(ij)} \sigma_i \sigma_j - B \sum_{i \in \mathbb{L}} \sigma_i, \tag{1.2}$$

donde la primera suma es sobre los índices (ij) con $i, j \in \mathbb{L}$, esto es, para cada sitio (vértice) de la red cuadrada simple se considera el espín del sitio i con los espines de los sitios que sean vecinos cercanos j (conectados por las líneas rojas de la figura 1.1). En la segunda suma, se tienen las contribuciones del campo magnético externo aplicado con magnitud B.

Es importante hacer notar que en el caso $B = 0$, se tiene que el estado base (el de menor energía) es un estado degenerado, pues hay dos configuraciones que tienen la misma energía: $\underline{\sigma}^{(+)}$ con $\sigma_i = +1 \quad \forall \sigma_i \in \underline{\sigma}^{(+)}$ y $\underline{\sigma}^{(-)}$ con $\sigma_i = -1 \quad \forall \sigma_i \in \underline{\sigma}^{(-)}$. Esta degeneración se rompe cuando $B \neq 0$.

1.2 *Spin Glass*

A diferencia del modelo de Ising, los spin glasses buscan modelar el comportamiento de materiales magnéticas más complejos, como las aleaciones, donde se tiene que las interacciones que hay entre los momentos magnéticos dependen de los tipos de materiales en la aleación, provocando que las interacciones puedan ser ferromagnéticas o anti-ferromagnéticas. Cuando se presentan las interacciones anti-ferromagnéticas, la forma de minimizar la energía entre los espines vecinos que interactúan es anti-alineando sus momentos magnéticos, mientras que se alinean cuando las interacciones son del tipo ferromagnético. Esta diferencia es la que hace a este tipo de sistemas complejos al momento de buscar las configuraciones de mínima energía.

Una forma ampliamente aceptada para modelar un sistema spin glass es mediante el *modelo de Edwards-Anderson* [2]. Este modelo tiene su base en el modelo de Ising de la sección anterior, manteniendo las restricciones en el tipo de estructura, por lo que se reduce a la red cuadrada simple en dos dimensiones. Pero en este modelo se necesita modificar la función para calcular la energía debido a las nuevas consideraciones, así que la nueva expresión para la energía es

$$E(\underline{\sigma}) = -\sum_{(ij)} J_{ij} \sigma_i \sigma_j - B \sum_{i \in \mathbb{L}} \sigma_i, \tag{1.3}$$

donde la única diferencia es el término J_{ij} en la primera suma, el cual se debe a las nuevas consideraciones acerca de los diferentes elementos usados en el material y sirve para representar el tipo de interacciones que cada espín tiene con sus vecinos, ya que estas pueden ser ferromagnéticas, a las cuales normalmente se les asigna el valor $J_{ij} = +1$, o anti-ferromagnéticas $J_{ij} = -1$ dependiendo el caso, como se puede ver en la figura 1.2.

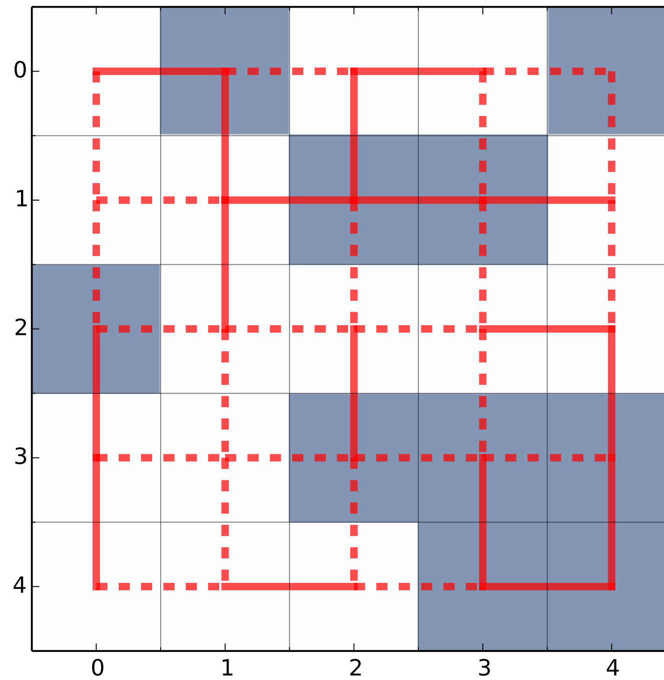


FIGURA 1.2: Una representación del modelo de *Edwards-Anderson* para $L = 5$. Las líneas rojas continuas representan interacciones ferromagnéticas ($J_{ij} = +1$) entre espines, mientras que las líneas rojas punteadas representan interacciones anti-ferromagnéticas ($J_{ij} = -1$). En este caso se tiene que la energía para la configuración es $E(\underline{\sigma}) = -2 + 5B$, la cual está dada por la ecuación (1.3).

A diferencia del modelo de Ising, el modelo de Edwards-Anderson presenta una complejidad mayor que se origina por los diferentes tipos de interacciones J_{ij} en el sistema que pueden provocar una nueva condición llamada *frustración*. Una *frustración* se presenta cuando hay restricciones impuestas por las constantes J_{ij} que impiden a los espines satisfacer todas las interacciones al mismo tiempo, como se muestra en la figura 1.3.

Dentro de este modelo se define como una *plaqueta* al conjunto de cuatro espines contiguos ordenados en forma de un cuadrado (figura 1.3). Se tiene que una plaqueta presenta frustración si y solo si el producto de las cuatro líneas (J_{ij}) que unen al cuadrado es negativo. En una plaqueta frustrada al no poder satisfacer

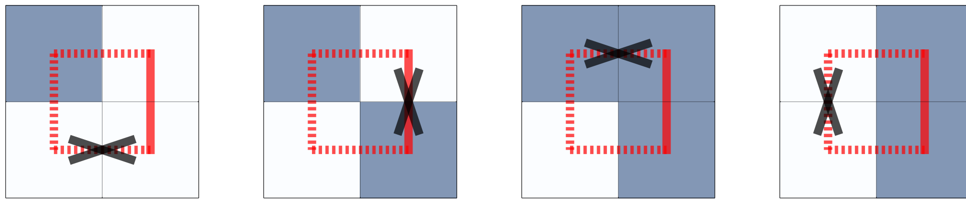


FIGURA 1.3: Cuatro plaquetas del modelo de *Edwards-Anderson* con $B = 0$ y frustraciones. Las líneas rojas continuas representan interacciones ferromagnéticas ($J_{ij} = +1$), mientras que las líneas rojas punteadas representan interacciones anti-ferromagnéticas ($J_{ij} = -1$). La cruz de color negro sobre las líneas rojas indica la interacción que no se pudo satisfacer en cada configuración. La energía de cada configuración de acuerdo con la ecuación (1.3) es $E(\underline{\sigma}) = -2$.

todas las interacciones, nunca se puede minimizar la energía de todos los espines en la plaqueta, y en el caso de no tener un campo magnético externo ($B = 0$), se tiene una degeneración en las configuraciones con la menor energía de la plaqueta.

De modo que las plaquetas frustradas dentro de un spin glass provocan que la búsqueda de la configuración con la energía mínima sea muy complicada, pues las interdependencias entre éstas y el resto del sistema no permiten que sean tratadas de forma aislada, lo que facilitaría la búsqueda de la configuración con la mínima energía. Finalmente hay que hacer notar que en cuanto se presenta una frustración en el sistema, la configuración del estado base es la configuración que satisface la mayor cantidad de interacciones.

Las características aquí descritas sobre los spin glasses y en la sección anterior para el modelo Ising, son suficientes para el objetivo de este trabajo, aunque en el capítulo 2 del libro de Marc Mezard y Andrea Montanari [2] se puede encontrar una descripción más detallada de ambos modelos. En el caso particular de los spin glasses, se puede encontrar una introducción más extensa a este tipo de sistemas y sus variantes en [3], [4] y [5].

1.3 Minimizando la energía de un Spin Glass

Uno de los puntos principales de este trabajo es poder encontrar las configuraciones de mínima energía para cualquier spin glass, pues la complejidad intrínseca de este tipo de sistemas hasta ahora ha impedido encontrar soluciones de forma analítica, por lo que se requieren métodos que permitan realizar búsquedas de

forma “inteligente” en el espacio de configuraciones, ya que este espacio crece de manera exponencial con el tamaño del sistema, como ya se mencionó previamente en el modelo de Ising, lo que al mismo tiempo vuelve impensable considerar el uso de fuerza bruta para explorar todas las posibles configuraciones. Por esta razón se propone implementar un algoritmo genético como método para buscar las soluciones óptimas del problema, pues un algoritmo genético es un método de búsqueda que se adapta conforme pasa el tiempo para poder encontrar posibles soluciones de un problema y usarlas para ir mejorando la calidad de estas de forma progresiva.

De modo que usar algoritmos genéticos en un spin glass es una forma “inteligente” de explorar el vasto espacio de configuraciones sin recurrir a la fuerza bruta (explorar todas las posibles configuraciones) y tratar de encontrar la configuración de mínima energía del sistema. Pues la propiedad que tienen los algoritmos genéticos para poder generar y mejorar (ó evolucionar) soluciones es lo que les permite explorar de una mejor forma el espacio de configuraciones, ya que dentro del algoritmo se establecen los parámetros y preferencias para que de esta manera se pueda “orientar” la búsqueda y evitar perder tiempo en regiones del espacio de configuraciones que no cumplen con el perfil de la búsqueda. Esta es la parte ‘inteligente” del método, aunque más adelante en el siguiente capítulo se discuten de forma más detallada que es un algoritmo genético, junto con las ventajas y desventajas de este método.

Además de la propuesta que se presenta aquí para encontrar las configuraciones de mínima energía de los spin glasses, cabe mencionar que hay otras propuestas que utilizan diferentes métodos, por ejemplo: Monte Carlo [6], Búsqueda Armónica y Cadenas de Markov [7] y Algoritmos de Optimización Jerárquicos Bayesianos (hBOA) [8].

1.4 Introducción al cómputo paralelo

La idea principal del cómputo paralelo es la de distribuir la carga computacional, de la manera más uniforme posible, sobre los recursos (núcleos de procesadores ó procesadores) disponibles; de esta forma se reduce el tiempo de ejecución del algoritmo implementado. Estas reducciones en los tiempos de ejecución, junto con el constante incremento del poder computacional, han hecho que el cómputo

paralelo se convierta en una opción muy atractiva para ayudar a resolver problemas complejos o que simplemente requieren gran poder computacional.

Una consecuencia de la paralelización de cualquier algoritmo es el aumento en la complejidad del mismo, ya que deben tenerse nuevas consideraciones por la ejecución simultánea de varios procesos y el flujo de datos entre ellos para que no haya conflictos con éstos y puedan afectar el resultado de la implementación.

Al diseñar un algoritmo paralelo se debe tener una particular atención en los datos que se manejan y se intercambian entre los diferentes procesos que se involucran, ya que pueden presentarse situaciones donde diferentes procesos necesiten acceder de forma simultánea a un mismo conjunto de datos, ya sea sólo para leerlos o para modificarlos, lo que puede causar problemas en los resultados del algoritmos. Este problema se conoce como *dependencia de datos*, el cual es de las principales fuentes de errores y restricciones cuando se intenta implementar una versión paralela de cualquier algoritmo [9].

La dependencia de datos, al ser una de las principales limitantes en la paralelización de cualquier algoritmo, influye de manera importante en el diseño y eficiencia de los mismos. Un ejemplo de la gran influencia que tiene la dependencia de datos se ve reflejada en la forma en la que se ejecutan las instrucciones en un algoritmo paralelo, ya que pueden ser ejecutadas de manera síncrona o asíncrona, como se muestra respectivamente en las figuras 1.4 y 1.5, pero al final la implementación de una u otra está sujeta al grado de dependencia de datos.

Aunque la libertad para implementar una ejecución síncrona o asíncrona en un algortimo paralelo está condicionada a la naturaleza del problema, las características de estos modos de ejecución son independientes.

Por ejemplo, un algoritmo con ejecución síncrona es más simple de programar. puesto que hay una sincronización de los procesos que ejecutan una instrucción, lo que significa que todos los procesadores ejecutan el mismo número de instrucciones en los mismos periodos de tiempo y también hay un mayor control en el manejo de datos. Pero a cambio de esta simplificación al programar se tiene una menor eficiencia computacional, pues como se puede ver en la figura 1.4, cuando un procesador termina una instrucción desperdicia mucho tiempo esperando a que el más lento termine. Una forma de reducir los tiempos de espera es mediante la repartición uniforme de cargas, para tratar de hacer que todos las instrucciones duren aproximadamente el mismo tiempo.

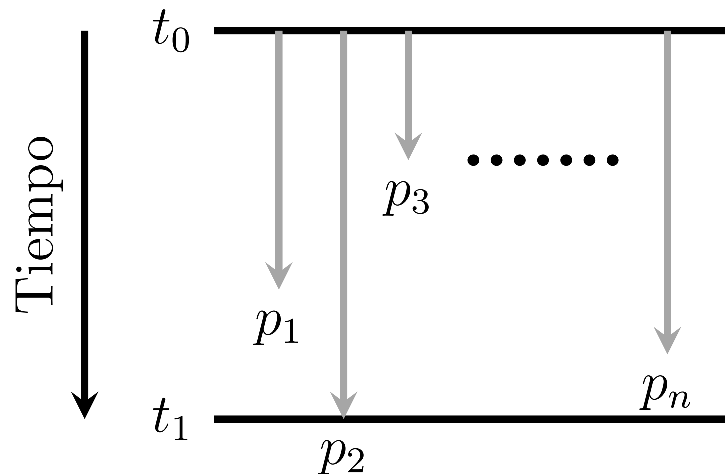


FIGURA 1.4: Representación de la ejecución síncrona de instrucciones en un algoritmo paralelo. La marca de t_0 representa el tiempo inicial en el que todas las instrucciones iniciaron su ejecución, mientras que t_1 representa una barrera de sincronización para ejecutar un nuevo conjunto de instrucciones. La barrera de sincronización se coloca cuando el proceso que consume mayor tiempo llega a su fin. Los procesos que terminan antes deben esperar al último para continuar.

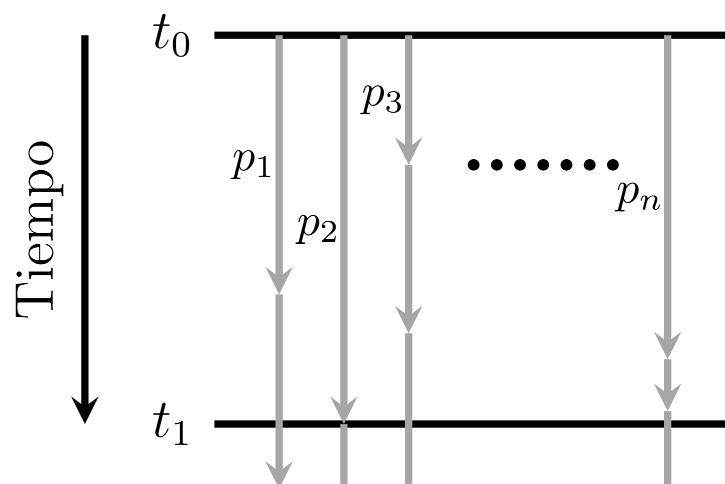


FIGURA 1.5: Representación de la ejecución asíncrona de instrucciones en un algoritmo paralelo. La marca de t_0 representa el tiempo inicial en el que todas las instrucciones iniciaron su ejecución, mientras que t_1 en este caso sólo representa una marca de tiempo ya que no hay barreras de sincronización en este tipo de implementaciones. En la ejecución asíncrona los procesos ejecutan la siguiente instrucción en cuanto terminan con la instrucción actual sin tener que esperar.

Ahora, si es posible implementar una ejecución asíncrona en un algoritmo paralelo, los recursos computacionales se aprovechan de una mejor forma, pues los procesadores están ocupados prácticamente todo el tiempo (figura 1.5) y esto se traduce como un menor tiempo de ejecución del algoritmo. Sin embargo el precio que se paga es una mayor complejidad al programar, ya que se debe tener un mayor control sobre el flujo de datos para no afectar los resultados.

Otra característica a considerarse en las implementaciones paralelas es que el algoritmo en cuestión sea escalable. Esto quiere decir que el algoritmo sea fácil de implementar sobre un mayor número de procesadores, sin tener que hacer cambios mayores en el mismo y que al mismo tiempo, siga siendo eficiente.

1.4.1 Principales arquitecturas paralelas

Dentro del cómputo paralelo existen diferentes formas para ser implementado, ya sea por las características físicas del equipo de cómputo (hardware) disponible o por el diseño del algoritmo que se busca paralelizar.

En cuanto al hardware del equipo de cómputo, se tiene que hay dos principales categorías ó arquitecturas computacionales: las de memoria compartida y las de memoria distribuida.

Los equipos basados en la arquitectura de memoria compartida se caracterizan por permitir acceso a todos los procesadores a una misma memoria (figura 1.6), lo que hace la programación paralela relativamente más fácil y simple que en el caso de la memoria distribuida. Otra ventaja de esta arquitectura es que las comunicaciones entre los procesadores y la memoria para acceder a los datos son muy rápidas, por lo que los tiempos de comunicación pueden despreciarse.

La principal desventaja de este tipo de arquitecturas es su limitación para escalar el número de procesadores y la cantidad de memoria por equipo, pues hay limitaciones físicas para el número de procesadores que se pueden tener en un equipo de este estilo [9]. De forma similar, existen limitaciones para la cantidad de memoria que se puede manejar, además de que el costo de estos equipos se incrementa con el número de procesadores y la cantidad de memoria en ellos.

En los equipos basados en la arquitectura de memoria distribuida (figura 1.7), a diferencia de los de memoria compartida, cada procesador tiene su unidad

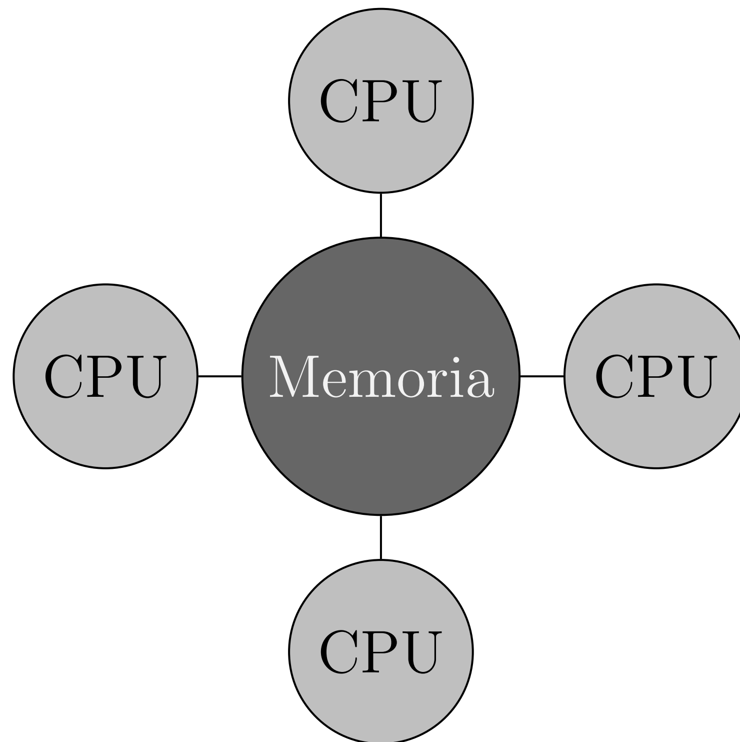


Figura 1.6: Esquema de una arquitectura de memoria compartida.

de memoria y la forma de pasar información entre los procesadores, ya que se encuentran aislados entre sí, es mediante mensajes a través de una red que los conecta. La forma más común de hacerlo es mediante el sistema estandarizado MPI (Message Passing Interface), que es un protocolo de comunicación para equipos de cómputo paralelo [10].

Los equipos basados en la arquitectura de memoria distribuida cuentan con dos grandes ventajas: se puede escalar el número de procesadores y cantidad de memoria, y la portabilidad, pues MPI se puede implementar prácticamente en cualquier equipo de cómputo paralelo. Estas dos ventajas se logran a cambio de una alta complejidad para programar, pues se deben dar instrucciones específicas a cada procesador sobre qué hacer, qué parte de los datos tienen y cómo se deben comunicar entre ellos para intercambiar información. Otra de sus grandes desventajas es el tiempo que consumen las comunicaciones entre los diferentes procesadores, ya que es por medio de una red y si ésta no está implementada de forma correcta, las afectaciones a los tiempos de comunicación son mayores. Una forma para tratar de reducir este problema es enviando la menor cantidad de mensajes posibles y que éstos pasen la mayor cantidad de información posible.

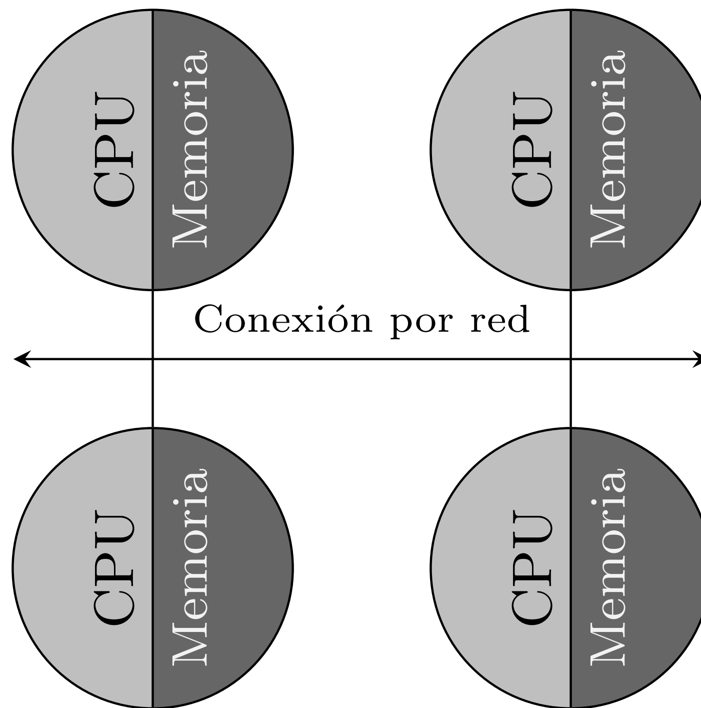


Figura 1.7: Esquema de una arquitectura de memoria distribuida.

También es común encontrar arquitecturas híbridas, donde se encuentre una combinación de memoria compartida con memoria distribuida, con el fin de extender de cierta forma las propiedades de la arquitectura de memoria compartida. Hay que mencionar que este tipo de arquitecturas híbridas son utilizadas por las supercomputadoras. En estos casos las ventajas y desventajas son una combinación de las dos arquitecturas.

En los últimos años ha surgido una opción alternativa a los equipos de cómputo paralelo basados en CPUs (las arquitecturas anteriores), la cual se basa en las tarjetas de procesamiento gráfico, conocidas como GPUs.

Los GPUs han sido diseñados bajo una arquitectura que se asemeja a la arquitectura de memoria compartida, siendo las principales diferencias el número y tipo de procesadores que los conforman. Este diseño permite utilizar sus procesadores gráficos para realizar cálculos de propósitos generales [11]. Los GPUs cuentan con un gran número de procesadores, aunque son inferiores en cuanto a la capacidad de cómputo comparados los de los CPUs , pero esto los convierte en opciones muy atractivas para problemas de “grano fino” [12].

Una introducción más extensa al cómputo paralelo es la que se encuentra en [13], mientras que para los GPUs se puede consultar [11].

1.4.2 Principales modelos de programación paralela

Así como existen diferentes arquitecturas para equipos de cómputo paralelo, también existen diferentes modelos para programar de forma paralela y aunque no se puede decir que uno es mejor que otro, sí se tiene que algunos modelos funcionan mejor que otros en ciertas implementaciones. Aquí se presentan dos modelos de los más comunes dentro de la programación paralela: el modelo de instrucción sencilla y múltiples datos o SIMD por sus siglas en inglés, y el modelo de múltiples instrucciones y múltiples datos o MIMD. Estos son modelos que pertenecen a la clasificación de Micheal Flynn [14].

En el modelo SIMD, como su nombre lo indica, los procesadores involucrados ejecutan una misma instrucción de forma simultánea sobre diferentes conjuntos de datos. Este modelo puede ser muy útil en algoritmos que tengan implementados ciclos sobre conjuntos de datos, ya que la misma operación que se repite en el ciclo puede ser ejecutada por los diferentes procesadores con la ventaja de que se puede repartir la carga de los conjuntos de datos y así reducir los tiempos de ejecución. Esta característica de ejecutar la misma instrucción en diferentes procesadores hace que este modelo sea fácil de implementar, ya que la única consideración que se debe tener al momento de programar es la repartición de los conjuntos de datos sobre los procesadores.

El modelo MIMD es más difícil de implementar, ya que, como su nombre lo indica, trabaja con múltiples instrucciones y múltiples datos, lo cual requiere una gran independencia en los datos que se utilicen y esto a veces es difícil de encontrar en los problemas que se quieren resolver. Además de esto, se debe tener en cuenta que al ejecutarse múltiples instrucciones en los procesadores, lo más probable es que éstas requieran diferentes cantidades de tiempo de ejecución, por lo que se utilizan ejecuciones asíncronas para tratar de mantener la eficiencia computacional. Pero de nuevo, al utilizar ejecuciones asíncronas la complejidad al programar aumenta.

1.4.3 *Speedup* y Eficiencia

La finalidad de implementar algoritmos de forma paralela es la de reducir sus tiempos de ejecución, y una forma de saber qué tan buena es la implementación paralela de un algoritmo es mediante dos cantidades: el *speedup* y la eficiencia, ya que usar el tiempo como escala de comparación no es muy útil, pues este cambia con el tamaño del problema, aunque sea el mismo algoritmo. Para este trabajo se utilizarán las definiciones de Ian Foster [15].

El speedup es el factor de reducción del tiempo de ejecución de la implementación paralela de un algoritmo respecto a la implementación serial. La forma en la que el speedup se define es la siguiente:

$$S_p = \frac{t_s}{t_p}, \tag{1.4}$$

donde S_p se refiere al speedup para p procesadores, t_s al tiempo de ejecución de la implementación serial del algoritmo y t_p es el tiempo de ejecución de la versión paralela ejecutándose sobre p procesadores.

Mientras que la eficiencia es una medida de qué tan bien son utilizados los recursos computacionales disponibles, esto lo hace considerando la fracción de tiempo que pasan realizando cálculos. La eficiencia se define como:

$$E_f = \frac{t_s}{pt_p}, \tag{1.5}$$

siendo E_f la eficiencia, t_s el tiempo de ejecución serial, t_p el tiempo de ejecución paralelo y p el número de procesadores utilizados.

En el caso de que la implementación paralela fuera implementada de forma ideal, el tiempo paralelo ideal sería $t_p = \frac{t_s}{p}$, lo que daría como resultado un speedup ideal de $S_p = p$ y una eficiencia $E_f = 1$. Esto quiere decir que cada procesador es utilizado todo el tiempo para realizar cálculos y que el tiempo de ejecución se reduce de forma lineal con el número de procesadores que se utilicen.

Capítulo 2

Algoritmos Genéticos

2.1 ¿Qué son los algoritmos genéticos?

Los algoritmos genéticos son métodos de búsqueda que se adaptan conforme avanzan en el tiempo y que tienen sus bases en los principios de la selección natural y de la genética [16]. Son estos principios los que rigen el comportamiento de los algoritmos y es lo que permite que las posibles soluciones encontradas puedan ir evolucionando de forma paulatina conforme surgen nuevas generaciones. Pero a pesar de estar basados en estos principios que son ampliamente conocidos y descritos de forma correcta, en el caso de los algoritmos genéticos, debido a la gran flexibilidad del método y al gran número de problemas en los cuales pueden ser implementados, no hay una clase de teoría general que pueda describir las propiedades que éstos tienen [16], aunque muchas de las bases fundamentales sobre las cuales trabajan fueron establecidas por primera vez por Holland [17].

Un algoritmo genético busca imitar los procesos biológicos que involucran a los principios de la selección natural y la supervivencia del más apto, que afectan a las poblaciones en la naturaleza para hacer que éstas evolucionen con el paso de muchas generaciones. Y son estos procesos los que originan competencias entre los individuos de una población por los recursos y para dejar su descendencia, siendo los individuos más aptos los que tienen una mayor probabilidad de tener éxito en dichos objetivos. Esto tiene como resultado que las características (genes) que permitieron el éxito de un individuo, se presenten en algunos de los individuos de una nueva generación, los cuales en algunas ocasiones pueden ser individuos con

una mayor adaptación comparada con la de sus padres (la generación anterior), lo que se convierte en una mayor competencia (calidad) en la población.

Para tratar de imitar lo anterior en un algoritmo genético, se deben establecer las analogías que permitan simular los comportamientos que se observan en la naturaleza. La primera de estas analogías es la que hay entre un individuo en una población, la cual se traduce como la de una solución en un conjunto de posibles soluciones.

Una analogía más que debe hacerse es la correspondiente a la parte genética, pues son los genes que pasan de una generación a otra los que permiten el surgimiento de individuos más aptos. La forma de transcribir este aspecto en un algoritmo genético es mediante la *codificación* de las posibles soluciones, teniendo en cuenta que al igual que los genes en los seres vivos, las soluciones codificadas deben poder combinarse y modificarse para generar nuevos individuos transmitiendo su información, los cuales posiblemente tengan nuevas y mejores propiedades. Se debe hacer notar que la codificación depende en su totalidad del problema que se esté tratando y en algunos casos la codificación no es trivial y se necesita buscar la forma más adecuada de hacerlo, pues gran parte del éxito o fracaso en la implementación de un algoritmo genético se debe a la codificación del problema que se busca resolver [16–18].

Ahora, de la misma forma que en una población hay individuos más aptos que otros, en un algoritmo genético se tiene que en el conjunto de posibles soluciones hay unas que son mejores que otras, y para poder decidir cual solución es mejor (más apta), se debe establecer una forma de clasificación para las soluciones. Esto se hace mediante una función, generalmente denominada *función de aptitud*, la cual tiene dependencias en la codificación usada.

Finalmente, para completar las analogías entre los procesos evolutivos naturales y los algoritmos genéticos, falta establecer cómo funcionan la reproducción y mutación de los individuos. Estos procesos se adaptan en los algoritmos genéticos usando funciones llamadas *operadores* que simulan los procesos de reproducción y mutación por separado, y que al igual que la función de aptitud, dependen de la codificación utilizada en el problema.

En la siguiente sección del capítulo se explican de forma más detallada cómo funcionan estos elementos y los procesos involucrados en la creación de nuevas generaciones en los algoritmos genéticos.

Es importante mencionar que en un algoritmo genético, dependiendo del tipo de problema y la dificultad del mismo, se pueden encontrar soluciones razonablemente buenas en un periodo de tiempo razonable, aunque esto no significa que se vaya a encontrar la mejor solución del problema, algo que no se puede garantizar.

Una introducción a los algoritmos genéticos más extensa, incluyendo algunos ejemplos prácticos, se presenta en el tercer capítulo del libro de John Koza [19].

2.2 Elementos básicos de un algoritmo genético

Habiendo establecido las analogías entre la naturaleza y los algoritmos genéticos, en esta sección se busca explicar cómo se presentan las implementaciones de estas analogías, junto con las propiedades que deben tener. También se busca explicar los procesos que son independientes del problema, como son el proceso de selección para la reproducción y el elitismo, pues estos elementos básicos son de gran utilidad para los algoritmos genéticos.

2.2.1 Codificación

La codificación se encarga de representar a las posibles soluciones, de manera que se puedan caracterizar convenientemente las propiedades que se buscan en las soluciones, es decir, parametrizar al problema. Para hacerlo se busca imitar la misma estructura que se presenta en la naturaleza con los genes y cromosomas, siendo los cromosomas estructuras que contienen a un conjunto de genes. Cuando se identifican los parámetros del problema que se quieren representar, éstos normalmente se agrupan en una cadena de números. Entonces, a cada elemento (número) de la cadena se convierte al equivalente a un gen, mientras que el conjunto de todos ellos (la cadena completa) equivale a un cromosoma. Es por esto que casi siempre se utilizan representaciones con cadenas de longitud fija.

Por ejemplo, se tiene que una configuración $\underline{\sigma}$ del modelo de Ising en una dimensión ($d = 1$) y con una longitud $L = 10$, se ve como: $\underline{\sigma} = (+1, -1, +1, -1, -1, -1, +1, +1, -1, +1)$, y es lo que se busca codificar. Pero por lo simple de este caso sólo hace falta identificar los elementos, así pues los spines son los parámetros del problema y como únicamente toman valores $+1$ y -1, la

representación actual ya puede ser considerada como la representación codificada de los genes. El conjunto de todos los spines puede ser considerado ya como un cromosoma codificado, pues tiene la propiedad de ser una cadena numérica de longitud fija, por lo que en este caso cada configuración $\underline{\sigma}$ ya es una representación codificada válida para un algoritmo genético.

De nuevo se debe mencionar que debido a la gran flexibilidad del método, existen diferentes formas para realizar la codificación, por lo que la forma utilizada en el ejemplo anterior no es única. Dentro de las variaciones más comunes se encuentran el utilizar una codificación en diferentes bases, como la binaria o la decimal, y usar cromosomas compuestos por varias cadenas en lugar de una sola.

2.2.2 Función de aptitud

La forma para poder clasificar a un conjunto de soluciones es mediante la función de aptitud, la cual requiere saber cuáles son los parámetros del problema, cómo se codificaron y la estructura en la que están agrupados. La función de aptitud debe determinar el peso y valor de cada gen dentro del cromosoma, dependiendo del problema que se busca resolver. Esta función debe poder asignar un valor numérico que sea capaz de reflejar la "aptitud" de cada una de las soluciones propuestas, para que de esta forma puedan clasificarse en base a esta característica. La clasificación de las posibles soluciones de acuerdo a qué tan buenas son resolviendo el problema es un aspecto muy importante en el algoritmo, pues es utilizado posteriormente en los procesos para crear la nueva generación de soluciones, ya que se busca dar preferencia a las soluciones más "aptas".

Siguiendo con el ejemplo del modelo de Ising en una dimensión, la función de aptitud que se necesita es la misma función que determina la energía de las configuraciones (ecuación (1.2)), pues cada spin tiene el mismo peso dentro de una configuración y lo que se busca encontrar es la configuración que tenga la menor energía. Ahora las configuraciones tienen un valor de "aptitud" asociado que es proporcional a su energía, el cual es usado para clasificarlas.

2.2.3 Mecanismos de reproducción

La creación de nuevas posibles soluciones (individuos) a partir del conjunto actual se realiza principalmente mediante dos operadores: el operador de reproducción y el operador de mutación. Ambos operadores trabajan sobre las soluciones ya codificadas y buscan tener comportamientos análogos a los que se presentan en la naturaleza. A continuación se presentan las principales características de ambos.

- **Operador de reproducción**: Usualmente se requieren sólo a dos individuos para este operador, que simula el cruzamiento de los cromosomas. Esto se hace cortando las cadenas de los cromosomas e intercambiándolas entre si, logrando así una "recombinación" del material genético en los cromosomas, que puede dar lugar a nuevos individuos con nuevas propiedades. Dependiendo de cómo se defina este operador, se pueden usar a más de 2 individuos y de igual forma los cromosomas se pueden dividir en 2 o más partes.

- **Operador de mutación**: Este operador se aplica de forma individual a los nuevos cromosomas (individuos) generados a partir del operador de reproducción y generalmente modifica a un solo gen (una posición) del cromosoma de forma aleatoria con una probabilidad muy pequeña de que esto pueda ocurrir, por lo general menor al 5%. La mutación es un elemento muy importante dentro de los algoritmos genéticos, pues es la única fuente de nuevo material genético, lo que se traduce como nuevos puntos de búsqueda en el espacio de configuraciones. También es importante hacer notar que el bajo porcentaje establecido para el operador de mutación se debe a que es un operador que puede tener un alto poder destructivo sobre los individuos de la población, además de que un alto porcentaje de mutación convertiría al algoritmo genético en una búsqueda aleatoria.

  La modificación que realiza este operador está regida por el tipo de codificación; por ejemplo, en el caso de una codificación binaria solo se cambia un 1 por un 0 y viceversa, mientras que para una codificación decimal se puede cambiar un número por cualquier otro de la base. De nuevo, existen más formas de definir el operador de mutación, las cuales dependen de la codificación y el problema elegidos [18].

Regresando al ejemplo del modelo de Ising en una dimensión y con longitud $L = 10$, se tiene que una forma de definir ambos operadores para que funcionen de forma correcta con el modelo, es como se muestra en las figuras 2.1 y 2.2, pues como se puede ver en los dos ejemplos, los individuos iniciales después de ser modificados por los operadores siguen teniendo todas las propiedades de una configuración válida para el modelo.

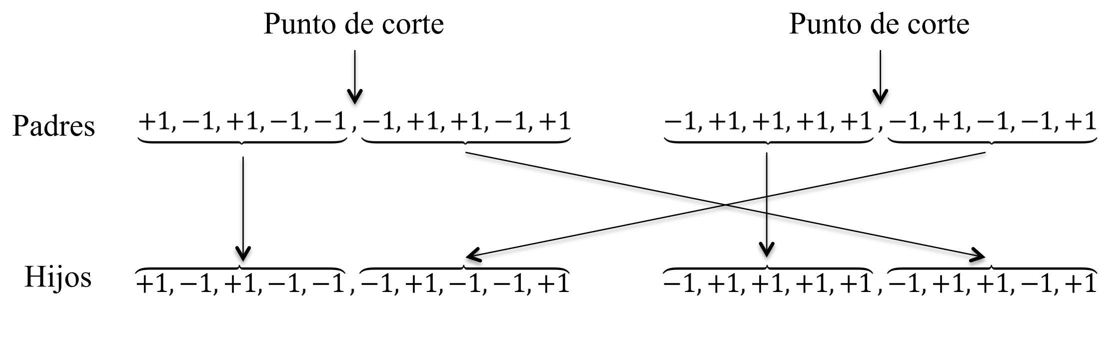


FIGURA 2.1: Operador de reproducción de un sólo punto de corte sobre 2 configuraciones.

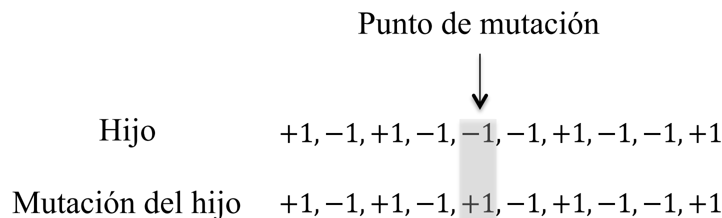


FIGURA 2.2: Operador de mutación simple sobre una configuración.

2.2.4 Métodos de selección

Son procedimientos que actúan sobre los individuos de una población, seleccionándolos de acuerdo a su aptitud para establecer prioridades en el proceso evolutivo del algoritmo. Estos métodos de selección pueden ser deterministas o estocásticos. Un método determinista es cuando cada individuo de la población tiene asignado un número fijo de veces que va a ser usado en los procesos reproductivos, mientras que en un método estocástico se asigna una probabilidad de selección a cada individuo [20].

Las principales diferencias entre ambos métodos se presentan en la homogeneización (o convergencia) de la población. La convergencia de una

población ocurre cuando todos los individuos de la población son iguales, esto es, todos tienen la misma representación genética. De acuerdo con [20], en los métodos de selección que involucran procesos deterministas, el individuo más apto domina la población en periodos de tiempo más cortos que en los casos donde se utilizan procesos estocásticos para la selección, lo que provoca una convergencia prematura de la población. Esta convergencia prematura conlleva una mayor probabilidad de encontrar una solución subóptima, lo cual se busca evitar en cualquier implementación.

Con el fin de evitar convergencias prematuras, se utilizan métodos estocásticos en los procesos de selección; dentro de esta clase de métodos hay dos técnicas muy comunes: el torneo y el proporcional a la aptitud.

El método de torneo involucra seleccionar un número k de individuos de una población de forma aleatoria con una distribución de probabilidad uniforme, para después seleccionar al individuo con las mejores características de acuerdo al problema de los k seleccionados en un principio. El proceso se repite hasta obtener el número de individuos necesarios para los procesos reproductivos.

El método de selección proporcional a la aptitud asigna una probabilidad de selección a cada individuo de acuerdo a su aptitud de la forma,

$$p_i = \frac{f_i}{\sum_i f_i}, \tag{2.1}$$

donde p_i es la probabilidad asignada al i-ésimo individuo de la población, f_i es el valor de la función de aptitud del i-ésimo individuo, y $\sum_i f_i$ es la suma del valor de la función de aptitud de cada individuo de la población. Este segundo método es más dinámico, pues las probabilidades que se asignan a los individuos de la población cambian de forma significativa conforme la población evoluciona, pues la distribución de probabilidad en la población depende de la aptitud de los individuos.

2.2.5 Elitismo

El elitismo es una medida que se sobrepone a los métodos de selección y tiene el fin de preservar a los mejores individuos de la población, es decir, los individuos

con los valores de aptitud más convenientes de acuerdo al problema [21]. La forma en la que esto se logra es pasando copias de los individuos más aptos de forma directa y sin modificación alguna a la siguiente generación de individuos (la nueva población), ya que sus características se pueden perder cuando se implementan los operadores de reproducción y mutación. De esta forma, con el elitismo se puede garantizar que la solución final encontrada por el algoritmo es la mejor solución encontrada a lo largo de todo el proceso evolutivo.

Usualmente sólo se preserva al individuo más apto de la población o se selecciona de forma elitista un porcentaje muy bajo de la población, siendo los individuos más aptos los elegidos. La razón de preservar un número bajo de individuos se debe a la forma en la que trabajan los algoritmos genéticos, pues si se preservara una gran parte de la población, el número de nuevos individuos sería menor, lo que se traduce como menos opciones nuevas para explorar el espacio de búsqueda. De modo que seleccionar a un gran número de individuos de forma elitista afecta el desempeño de los algoritmos genéticos, mientras que preservar un número pequeño, comparado con el tamaño de la población, ayuda a mantener pistas sobre regiones en el espacio de búsqueda con mejores características que sirvan al problema y al mismo tiempo permite explorar más regiones nuevas.

2.2.6 Convergencia

La convergencia en los algoritmos genéticos hace referencia principalmente al grado de homogeneización de la población, como ya se mencionó dentro de los métodos de selección. Una forma para poder darse cuenta de que la población ha convergido a alguna solución es observar los valores de aptitud promedio de la población, pues una característica de la convergencia en un población es que no haya cambios significativos en estos valores con el paso de las generaciones, lo que es resultado a la similitud entre los individuos de la población. Así que entre mayor sea la homogeneización de la población, se tiene una convergencia más marcada y una menor variación en los valores de aptitud de la población. Aunque se debe hacer notar que dependiendo del problema no siempre se puede asegurar el tipo de la solución a la cual se converge, ya que muchas veces no se conoce a la mejor solución, por lo que ésta puede ser una solución óptima o una subóptima en el caso de que haya múltiples soluciones.

Otro punto importante que se presenta al haber convergencia en una implementación es que el método de búsqueda del algoritmo deja de ser eficiente, ya que los métodos de reproducción dejan de ser efectivos. Por ejemplo, el operador de reproducción deja de tener efecto, ya que al tener una población con individuos iguales, las partes que se intercambian son las mismas. De forma similar, el efecto del operador de mutación deja de ser relevante, por su baja probabilidad de que ocurra y porque generalmente tiende a destruir soluciones, lo que significa que los individuos con mutaciones, por lo general, tienen una menor aptitud y por lo tanto una menor probabilidad de ser seleccionados por el operador de reproducción en la siguiente generación.

Debido a estas características que se presentan en un algoritmo cuando hay convergencia, es común que ésta sea utilizada como un criterio para terminar la ejecución del algoritmo.

2.3 Diversidad en los algoritmos genéticos

La diversidad en los algoritmos genéticos es una propiedad intrínseca de las poblaciones que se refiere a las diferencias que existen entre los individuos de una población. De este modo, entre más diferentes sean los individuos de una población, mayor es su diversidad.

Las principales repercusiones que tiene la diversidad en las implementaciones de los algoritmos genéticos están relacionadas con convergencias prematuras que impiden encontrar soluciones óptimas de los problemas, por lo que muchos trabajos han reconocido este problema [19, 20] y algunos más sugieren métodos y estrategias para tratar de evitar este tipo de problemas [22, 23].

La tendencia de los algoritmos genéticos a converger de forma prematura a óptimos locales (subóptimos) se debe al flujo de material genético entre la población. Pues en una población finita donde no haya fuentes de nuevo material genético, es decir, que se encuentre aislada y que no haya mutaciones, el flujo de material genético está sujeto a los procesos de selección. De modo que si los procesos de selección realizan un muestreo estocástico uniforme, a lo más, se puede mantener la diversidad genética de la población, pero usualmente esta se reduce por fallas en el muestreo provocando su homogeneización, lo que es conocido en el caso extremo como *deriva genética* [20, 24].

Con una baja diversidad de la población se incrementan las posibilidades de quedar atrapado en un subóptimo, pues la baja diversidad está asociada con problemas de convergencia prematura. Si a la baja diversidad en una población se le agrega un espacio de búsqueda complejo con muchos subóptimos, la probabilidad de encontrar la solución óptima global disminuye aún más, pues la convergencia prematura evita que se lleve a cabo una amplia exploración del espacio de búsqueda. Por estas razones se busca tener una diversidad alta dentro de la población.

Tener una alta diversidad dentro de una población no significa que ésta vaya a converger a la solución óptima global del problema, pero definitivamente ayuda a evadir subóptimos del espacio de búsqueda, por lo que mantener una diversidad relativamente alta a lo largo de la ejecución de un algoritmo genético tiende a mejorar la calidad de los resultados obtenidos, puesto que se realiza una mejor exploración del espacio de búsqueda. Un punto importante a considerar cuando se tiene una alta diversidad en un algoritmo genético es el incremento en el tiempo de convergencia, lo cual puede no ser favorable en términos computacionales.

Es importante hacer notar que para lograr una implementación exitosa de un algoritmo genético, se necesitan tener diferencias en la diversidad de la población en diferentes momentos, ya que se necesita una diversidad alta en las partes iniciales del algoritmo para evitar subóptimos, pero también se necesita de una disminución de la diversidad para poder converger a una solución óptima en la parte final. Esto convierte a la diversidad en un elemento dinámico de los algoritmos genéticos, pues varios factores interfieren entre sí, dificultando el poder implementar de forma eficiente un algoritmo genético [20]. Gran parte de esta complejidad es inherente al problema que se quiera tratar, pues cada problema presenta un espacio de búsqueda diferente, los parámetros de tamaño de población y número de generaciones necesarias para obtener un resultado aceptable también son diferentes, así como los operadores y en algunos casos los métodos de selección, según sean necesarios, y una mala elección en cualquiera de estos elementos se puede ver reflejada en una rápida pérdida de diversidad y un desempeño pobre del algoritmo.

Debido a la importancia que tiene la diversidad en el desempeño de los algoritmos genéticos, se pueden implementar criterios auxiliares para poder medir la diversidad de una población y usar la información que se obtiene para lograr

ajustes en los parámetros y elementos de un algoritmo genético, de modo que se pueda mejorar su eficacia.

Así como hay una gran variedad de problemas en los que se pueden utilizar a los algoritmos genéticos, las formas en las que se puede medir la diversidad de una población también son variadas, dependiendo de la información o ajustes que se quieran hacer, siendo la *Distancia Hamming* y la *Entropía de Shannon* unas de las medidas más utilizadas [2, 20, 23, 25]. Una descripción más detallada sobre estas dos medidas de diversidad se presenta en las siguientes secciones.

Las opciones para medir la diversidad de una población en los algoritmos genéticos se apoyan principalmente en tres elementos: *genotipo*, *fenotipo* y *aptitud*.

Para poder explicar los elementos anteriores es necesario recordar que un cromosoma es una estructura conformada por un conjunto de genes, que puede ser interpretada como un vector de longitud fija donde cada una de sus entradas corresponde a un gen. Un concepto más que debe introducirse es el de *alelo*; los alelos son las posibles formas alternativas que puede tener un gen particular, lo que sería el equivalente a los valores que pueden tomar las entradas de un vector. De modo que las variaciones de los cromosomas están determinadas por los alelos de los genes que lo conforman.

Entonces, el genotipo es el nombre que recibe una configuración específica de un cromosoma, es decir, el conjunto de cada alelo en los genes del cromosoma, por lo que si dos individuos tienen diferentes alelos en alguno de los genes que los componen, entonces el genotipo de ambos individuos es diferente. El fenotipo hace referencia a los valores decodificados de los genes del cromosoma de un individuo. Mientras que la aptitud, como ya se ha visto, es el valor que regresa la función de aptitud y que sirve para clasificar la calidad del individuo como solución en base a su genotipo.

2.3.1 Distancia Hamming

La distancia Hamming es una medida de diversidad diseñada para codificaciones binarias, por lo que es muy común en las implementaciones de algoritmos genéticos, ya que muchos utilizan este tipo de codificación en las representaciones de los problemas. Debido a esto, se puede considerar que la distancia Hamming es una medida de diversidad del genotipo de la población.

Ésta se define sobre pares de cadenas con la misma longitud l, siendo la distancia entre las dos cadenas que se comparan el número de elementos en los que ambas difieren. Por ejemplo, si se considera el caso de las cadenas con longitud $l = 5$: $x = 10111$ y $y = 10001$, la distancia Hamming entre estas cadenas es $d_H(x, y) = 2$, ya que las cadenas x y y sólo difieren entre sí en 2 elementos (el tercero y cuarto).

De modo que la distancia máxima entre dos cadenas se presenta cuando éstas son completamente diferentes y el valor de la distancia Hamming es igual a la longitud l de las cadenas, mientras que la distancia mínima es cuando las cadenas son iguales y entonces se tiene una distancia igual a 0.

Por la forma en la que se define la distancia Hamming, se tiene que para todo elemento x, y y $z \in \mathbb{X}$, siendo $\mathbb{X}$ el conjunto de cadenas binarias con longitud l, la distancia cumple con las propiedades:

$$d_H(x, y) \geq 0, \tag{2.2}$$

$$d_H(x, y) = 0 \quad \Leftrightarrow \quad x = y, \tag{2.3}$$

$$d_H(x, y) = d_H(y, x), \tag{2.4}$$

$$d_H(x, z) \leq d_H(x, y) + d_H(y, z), \tag{2.5}$$

donde d_H se refiere a la distancia Hamming. Al cumplir con las propiedades anteriores, se tiene que la distancia Hamming induce una métrica sobre el conjunto de cadenas $\mathbb{X}$.

Ahora, para tener una noción más intuitiva sobre las diferencias registradas por la distancia Hamming puede ser más conveniente normalizar los valores, lo único que se debe hacer es dividir los valores de distancia obtenidos entre la longitud l de las cadenas. De este modo se tiene que la mayor distancia que se puede obtener ahora es 1, mientras que la menor sigue siendo 0.

Una vez que se tiene claro la forma en la que trabaja la distancia Hamming y sus propiedades, se debe resaltar el hecho de que opera sobre pares, por lo que si se busca obtener un número que pueda mostrar la distancia promedio que hay en una población con m individuos, se deben calcular las distancias de cada uno de los individuos con el resto de los individuos de la población. Una forma para poder visualizar las distancias que se deben calcular es viendo cada par como elementos de una matriz, por lo que se tiene una matriz cuadrada de lado m, donde la diagonal es 0 por la propiedad (2.3). Ahora tanto por simetría, como

por ahorrar tiempo de cálculo, la propiedad (2.4) dice que los elementos arriba de la diagonal son iguales a los de la parte inferior, por lo que sólo es necesario calcular una de estas partes. Entonces se reduce al caso de una matriz triangular inferior (ó superior), sin contar la diagonal. De este modo se tiene que se deben calcular un total de $\frac{m(m-1)}{2}$ distancias para poder obtener la distancia promedio de una población con m individuos. La expresión para la distancia promedio de una población es la siguiente:

$$\overline{d_H} = \frac{2}{(m^2-m)}\frac{1}{l}\left[\sum_{i=1}^{m-1}\sum_{j=i+1}^{m} d_H(x_i,x_j)\right] \qquad x_i,x_j \in \mathbb{X}, \tag{2.6}$$

donde $\overline{d_H}$ es la distancia Hamming promedio de una población con m individuos. Los coeficientes $\frac{2}{(m^2-m)}$ y $\frac{1}{l}$ son usados para normalizar, el primero respecto al numero de individuos de la población y el segundo respecto a la longitud de los mismos. Finalmente las sumas recorren a la población calculando las distancias $d_H(x_i,x_j)$ necesarias para obtener el promedio.

El valor $\overline{d_H}$ de la distancia Hamming promedio de una población es lo que se utiliza como medida de diversidad de una población.

2.3.2 Entropía de Shannon

Una forma de medir la diversidad de una población es mediante las diferencias en las estructuras genéticas, o genotipos, como lo hace la distancia Hamming, pero los genotipos no reflejan completamente la información que tienen los individuos de la población y es por esto que se hace uso de otras propiedades como el fenotipo o la aptitud de los individuos para tratar de tener una imagen más completa de lo que es la diversidad en una población. Recurrir a propiedades diferentes al genotipo para medir la diversidad de una población se vuelve importante en problemas donde se tengan individuos con genotipos diferentes que presentan un fenotipo o un valor de aptitud igual a pesar de la diferencia en la estructura genética. Este tipo de situaciones son las que se buscan tratar con la introducción de la entropía de Shannon como medida de diversidad de una población.

Para definir la entropía de Shanon se necesita una variable aleatoria discreta X, definida por el conjunto finito de valores que puede tomar X, y con una distribución de probabilidad $p(x)$, donde $p(x) : \mathsf{X} \rightarrow [0,1]$ y satisface

$$\sum_{x \in \mathsf{X}} p(x) = 1, \tag{2.7}$$

Entonces, usando la definición propuesta en [2], se tiene que la entropía de Shannon S es,

$$S = -\sum_{x \in \mathsf{X}} p(x) \log_2 p(x), \tag{2.8}$$

donde se define $0 \log_2 0 = 0$. Con esta definición, se tiene que la entropía de Shannon tiene su valor máximo cuando todos los posibles estados que conforman a la distribución de probabilidad $p(x)$ son equiprobables, y es 0 cuando el valor de la variable aleatoria se sabe con certeza, es decir, X sólo puede tomar un valor.

Para establecer la conexión entre la entropía de Shannon y los algoritmos genéticos, se tienen que identificar a los elementos, así pues, X corresponde a la población, x a los individuos que la conforman, mientras que $p(x)$ a la porción de los individuos en la población que tienen en común un mismo valor de alguna propiedad, como el fenotipo o su valor de aptitud.

Así que, de acuerdo a la ecuación (2.8), la entropía de Shannon del sistema considera el peso de cada contribución de las porciones en las que se divide la población de acuerdo a la propiedad que se seleccione, teniendo valores altos cuando la población cuenta con un gran número de valores diferentes, lo que equivale a tener una alta diversidad en la población, es decir, una población heterogénea en cuanto a la propiedad seleccionada.

Para los casos en los que la entropía de Shannon es baja, lo que se puede inferir es que la población se divide en pocas porciones con diferentes valores. La condición de baja entropía de Shannon es el equivalente a la convergencia de una población.

A partir de ahora, se hará referencia en este trabajo a la entropía de Shannon como sólo *entropía*.

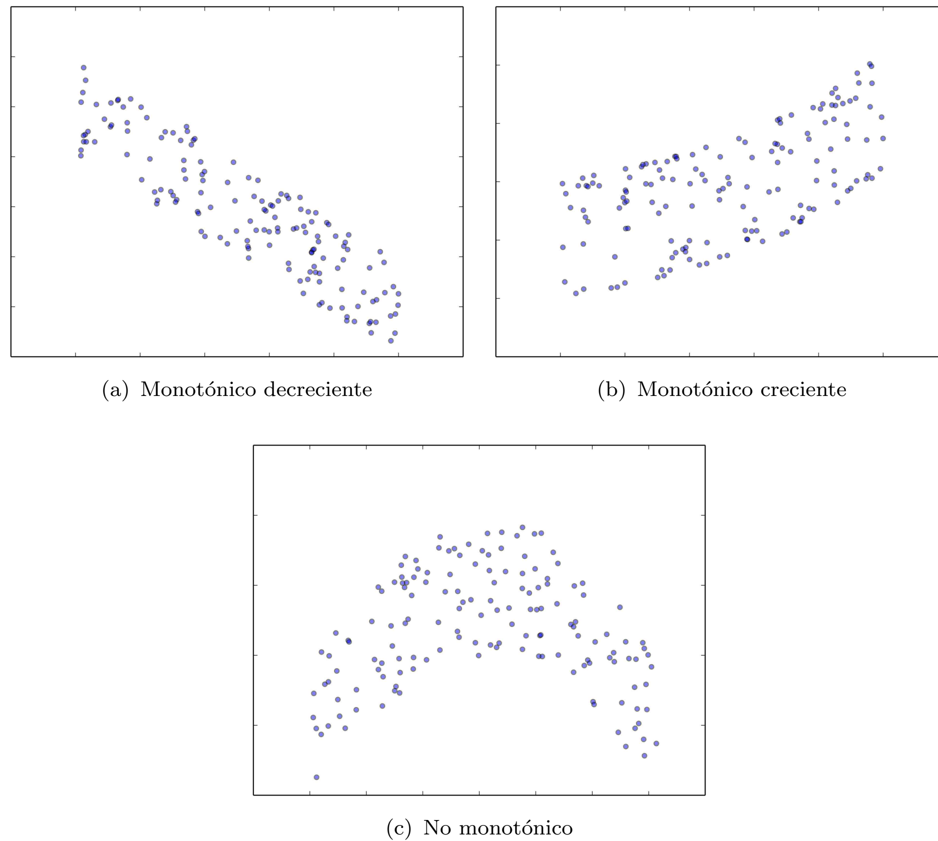
En [25] se presenta un trabajo más detallado sobre la entropía en los algoritmos evolutivos, incluyendo más interpretaciones e implementaciones de la misma.

2.3.3 Correlación de Spearman

Una forma para poder obtener más información sobre la influencia que tiene la diversidad de la población en el desempeño de un algoritmo genético es mediante la correlación de la información obtenida de las medidas de diversidad y algún otro parámetro importante del algoritmo, como el valor de aptitud del mejor individuo de la población. Lo que se busca con la correlación es medir de cierta forma la dependencia que hay entre los datos que se comparan, para utilizar posteriormente la información conseguida y tratar de determinar el grado de influencia que la diversidad, en cualquiera de sus medidas, tiene sobre el principal objetivo de encontrar soluciones óptimas con algoritmos genéticos.

Al haber diferentes formas de medir la correlación que hay entre conjuntos de datos, el tener conocimiento sobre las características de los mismos se vuelve de gran importancia para determinar el tipo de correlación que se busca. Por ejemplo, en el caso que aquí se muestra, la correlación de Spearman (una variante de la correlación de Pearson), asigna un valor de rango de acuerdo al orden de los elementos en los conjuntos de datos, ya sea en orden creciente o decreciente, según se requiera, y utiliza los rangos asignados para establecer la correlación. La diferencia de utilizar los conjuntos de datos o los rangos asociados a estos es lo que hace la diferencia entre estas dos correlaciones, pues en el caso de Pearson es una correlación lineal lo que se evalúa, mientras que en el caso de Spearman se busca una correlación monotónica (figura 2.3). Y es por esto que tener un conocimiento o noción previa del comportamiento que se espera es útil para seleccionar la herramienta más conveniente.

La forma en la que se asigna el valor del rango a los datos de un conjunto es la siguiente: si se tienen los conjuntos de datos ordenados $X = (3.3, 3.6, 4.0, 4.1, 4.2)$, $Y = (10, 12, 13, 13, 14)$, $Z = (6, 7, 7, 7, 8)$, los rangos se asignan de forma creciente en este ejemplo, por lo que para el conjunto X se tiene que sus rangos son $R_X = (1, 2, 3, 4, 5)$. En los casos de Y y Z hay valores repetidos y la forma de asignar el rango de los valores repetidos es calculando el promedio de los lugares que ocupan en el conjunto de datos ordenados. Para el conjunto Y se tiene que los datos que se repiten ocupan el lugar 3 y 4 en el conjunto, teniendo que el promedio de estos lugares es 3.5, por lo que los rangos para Y quedan como $R_Y = (1, 2, 3.5, 3.5, 5)$. Finalmente para Z se repiten los lugares 2, 3 y 4, por lo que el promedio de los lugares es 3 y los rangos del conjunto son $R_Z = (1, 3, 3, 3, 5)$.



(a) Monotónico decreciente

(b) Monotónico creciente

(c) No monotónico

FIGURA 2.3: Las gráficas (a) y (b) representan comportamientos monotónicos, mientras que (c) muestra un comportamiento no monotónico.

La correlación de Spearman, como ya se mencionó, es una variante de la correlación de Pearson que trabaja con los rangos de los conjuntos de datos, por lo que la fórmula que se utiliza para calcular ambas correlaciones es la misma. La forma en la que se define la correlación de Pearson es

$$\rho = \frac{\sum_i (x_i - \overline{x})(y_i - \overline{y})}{\sqrt{\sum_i (x_i - \overline{x})^2 \sum_i (y_i - \overline{y})^2}} \qquad x_i \in X, y_i \in Y, \tag{2.9}$$

donde ρ es el coeficiente de correlación, $\overline{x}$ es el promedio de X y de forma análoga $\overline{y}$ el promedio de Y. Ahora, para medir la correlación de Spearman en lugar de la de Pearson sólo hay que cambiar los conjuntos de datos X y Y por sus respectivos conjuntos de rangos R_X y R_Y. De este modo se mide una correlación monotónica en lugar de una lineal.

Una descripción más extensa acerca de la correlación de Spearman, incluyendo diferentes formas de calcularla para diferentes casos es la que se presenta en [26].

En este trabajo se busca establecer una correlación entre los conjuntos de datos de la diversidad de una población y el valor de la mejor aptitud dentro de la misma población. Elegir el valor de la mejor aptitud para establecer la correlación, se debe a qué éste presenta un comportamiento monotónico en función del tiempo (las generaciones) causado por la implementación del elitismo en el algoritmo, además de que el valor de la mejor aptitud representa el objetivo principal del trabajo, la configuración de espines con la menor energía. El comportamiento monotónico que presenta el valor de mejor aptitud sugiere utilizar una correlación que explote esta característica, y es por esta razón que se elige a la correlación de Spearman para analizar la posible dependencia que tenga con la diversidad de la población.

2.4 Algoritmos genéticos paralelos

La efectividad de los algoritmos genéticos los ha llevado a ser implementados en una gran variedad de problemas, los cuales presentan diferentes niveles de complejidad que se reflejan en las demandas de recursos computacionales y en el tiempo de ejecución del algoritmo.

El trabajar con problemas de alta complejidad que demandan una mayor cantidad de recursos computacionales y que requieren de mayores tiempos de ejecución, puede representar un problema en relación a la eficiencia del algoritmo para encontrar una buena solución en un periodo de tiempo aceptable. Una forma de compensar la gran demanda computacional para mantener tiempos relativamente cortos, o aceptables, es mediante el cómputo paralelo, donde se busca repartir de forma uniforme la carga computacional con el fin de reducir los tiempos de ejecución.

En el caso de los algoritmos genéticos, el cómputo paralelo, además de ayudar a mantener tiempos de ejecución aceptables, permite implementar nuevas variaciones en los algoritmos que pueden modificar de forma significativa su comportamiento. En las siguientes secciones se explicará más acerca de estas variaciones y sus consecuencias.

2.4.1 Clasificación de los algoritmos genéticos paralelos

La gran diversidad de formas para implementar un algoritmo de forma paralela tiene como resultado una nueva variedad de opciones en el caso de los algoritmos genéticos, por lo que es necesario establecer una clasificación general de los mismos. La clasificación que aquí se muestra está basada en el trabajo de Erick Cantú Paz [1] y [27].

Dentro de la clasificación de Cantú Paz se reconocen a cuatro tipos principales de algoritmos genéticos paralelos:

- **Maestro-esclavo con una sola población**
- **Múltiples poblaciones**
- ***Grano fino***
- **Híbridos jerárquicos**

El primer caso, los algoritmos genéticos del tipo **maestro-esclavo con una sola población** tienen la principal característica de concentrar a toda la población en un nodo maestro. Este nodo maestro se encarga de realizar los procesos de selección, y aplicar los operadores de reproducción y mutación, ya que generalmente estos procesos no requieren una gran cantidad de recursos computacionales. Mientras que en los nodos esclavos se ejecutan las partes más demandantes en cuestión de recursos computacionales, como lo es la función de aptitud, siendo el nodo maestro quien decide cómo repartir el trabajo sobre los nodos esclavos. De esta forma no se modifica para nada el comportamiento del algoritmo, por lo que el comportamiento del algoritmo paralelo debería ser análogo al de las implementaciones seriales, con la única diferencia en la reducción del tiempo de ejecución. En la figura 2.4 se muestra un esquema de la estructura para este tipo de implementaciones.

En el caso de los algoritmos genéticos con **múltiples poblaciones**, se presenta un nuevo elemento que los hace diferentes de todas las demás implementaciones. La gran diferencia, como su nombre lo dice, es que consideran múltiples poblaciones (figura 2.5), con la característica de que estas poblaciones. ó subpoblaciones si se piensa al conjunto total de éstas como una sola población, intercambian

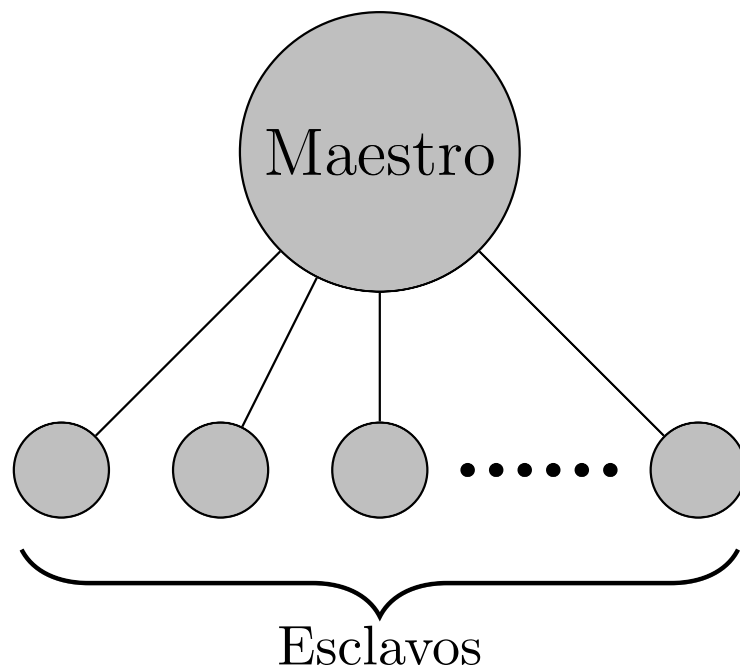


Figura 2.4: Estructura de un algoritmo genético paralelo del tipo maestro-esclavo con una sola población. El nodo más grande representa al nodo maestro, mientras que los nodos pequeños a los nodos esclavos.

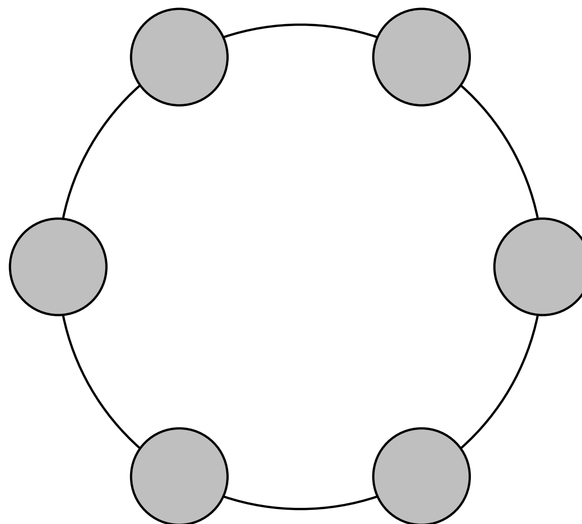


Figura 2.5: Estructura de un algoritmo genético paralelo con múltiples poblaciones (ó subpoblaciones). Cada nodo representa a una subpoblación en principio aislada, ya que hay intercambios de individuos entre éstas de forma ocasional.

individuos entre sí de forma ocasional y controlada. Este intercambio de individuos es conocido como *migración*, de la cual se hablará más adelante.

La migración dentro de los algoritmos genéticos paralelos incrementa la complejidad de éstos, ya que la migración tiene grandes efectos sobre el desempeño de los algoritmos y por esta razón se deben establecer parámetros para el tamaño y número de las subpoblaciones, el número de individuos que se van a intercambiar entre éstas, la frecuencia con la que esto se hace y los destinos de los individuos que migran, así como los criterios de selección para estos individuos.

Un factor a considerarse en este tipo de implementaciones es que normalmente el tamaño de las subpoblaciones es más pequeño que el de una población en un algortimo genético serial, y esto puede traer consigo problemas de convergencia prematura, ya que las poblaciones pequeñas tienden a converger más rápido, lo que aumenta la probabilidad de encontrar soluciones subóptimas.

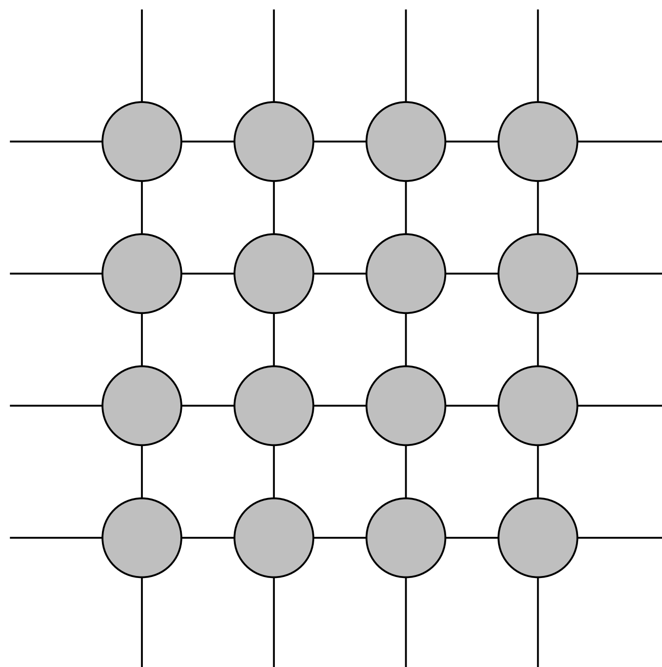


FIGURA 2.6: Estructura de un algoritmo genético paralelo de *grano fino*. Cada nodo en la red representa a un individuo de la población. Para este tipo de implementaciones se busca tener un procesador por individuo.

Para el caso de los algoritmos genéticos del tipo ***grano fino***, de nuevo sólo se considera una población (global), con la diferencia de que estos algoritmos se ejecutan usualmente sobre una estructura de red cúbica simple en 2-D (figura 2.6). La forma ideal para implementar un algoritmo de este tipo es teniendo un

procesador por cada individuo de la población, para que de este modo la evaluación de la función de aptitud sea ejecutada de forma simultánea sobre toda la población.

Los métodos de selección y reproducción son diferentes en este tipo de implementaciones, ya que hay restricciones para seleccionar únicamente a individuos dentro de una vecindad en la red. Estas restricciones provocan que las características dominantes se propaguen de forma análoga a partículas en líquido y por esta razón este tipo de implementaciones son conocidas como "modelo de difusión". La estructura de este tipo de algoritmos genéticos los hace amplios candidatos al modelo de programación SIMD.

Debido a las diferencias y restricciones en las dinámicas de los algoritmos genéticos de múltiples poblaciones y de grano fino, su comportamiento es diferente al de los algoritmos genéticos seriales, y como resultado, diferentes fenómenos pueden ser observados en estas implementaciones.

Finalmente los algoritmos genéticos del tipo **híbrido jerárquico** son una combinación del tipo de múltiples poblaciones con los del tipo de maestro-esclavo o de grano fino. La idea es tener en un nivel exterior el esquema de múltiples poblaciones y en nivel más interior la estructura de maestro-esclavo o de grano fino, dependiendo del caso. De este modo, se tiene una combinación de las mejores características de cada modelo para lograr uno más robusto con mejores posibilidades de tener un mejor desempeño. Las figuras 2.7 y 2.8 muestran las ideas de estas implementaciones.

2.4.2 Topologías y esquemas de migración

Como se acaba de ver, las implementaciones más robustas de los algoritmos genéticos paralelos involucran al esquema de múltiples poblaciones (subpoblaciones), de modo que es importante establecer algunos puntos sobre la migración entre las subpoblaciones como los criterios de selección de los individuos que migran y la forma en la que lo hacen hacia las demás subpoblaciones (las topologías).

Primero se debe hacer notar que existen dos casos extremos de la migración, el primero es cuando no hay migración alguna (es la frecuencia más baja de migración) más que al final para comparar los resultados obtenidos y seleccionar a la mejor solución. El segundo es cuando la migración se hace cada generación (esta

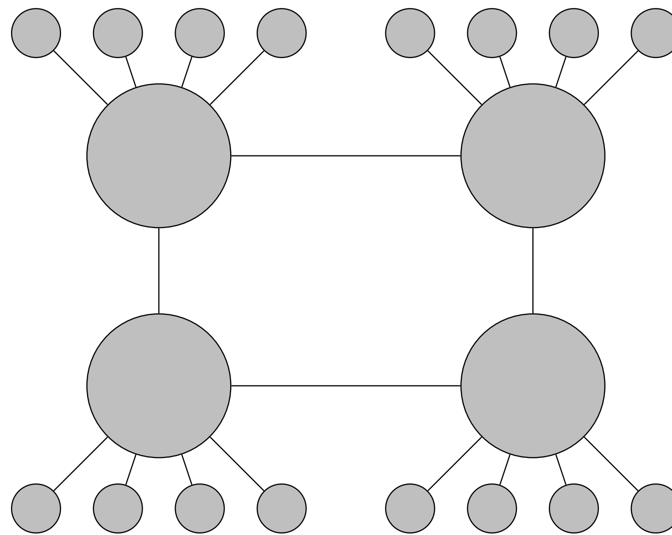


Figura 2.7: Estructura de un algoritmo genético paralelo *híbrido jerárquico* combinando múltiples poblaciones con maestro-esclavo. Los nodos principales representan a las poblaciones, mientras que los nodos pequeños a los esclavos correspondientes de estas.

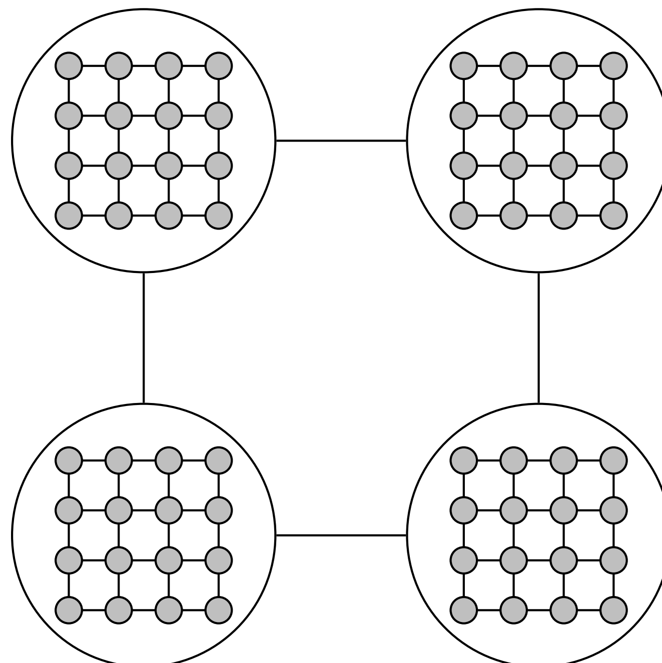


Figura 2.8: Estructura de un algoritmo genético paralelo *híbrido jerárquico* combinando múltiples poblaciones con grano fino. Los nodos principales representan a las poblaciones, mientras que los nodos pequeños a los individuos de cada población en una implementación de grano fino.

es la frecuencia más alta). En el caso de migración nula el comportamiento del algoritmo se reduce al de uno con una sola población, que muchas veces es el mismo que el de la implementación serial. Si la migración se realiza cada generación, la comunicación entre las subpoblaciones hace qué estas simulen el comportamiento de una sola población global, lo cual se vuelve más notorio cuando la migración involucra un número considerable de individuos, lo que puede ocasionar que cualquier ventaja de mantener una subpoblación aislada desaparezca.

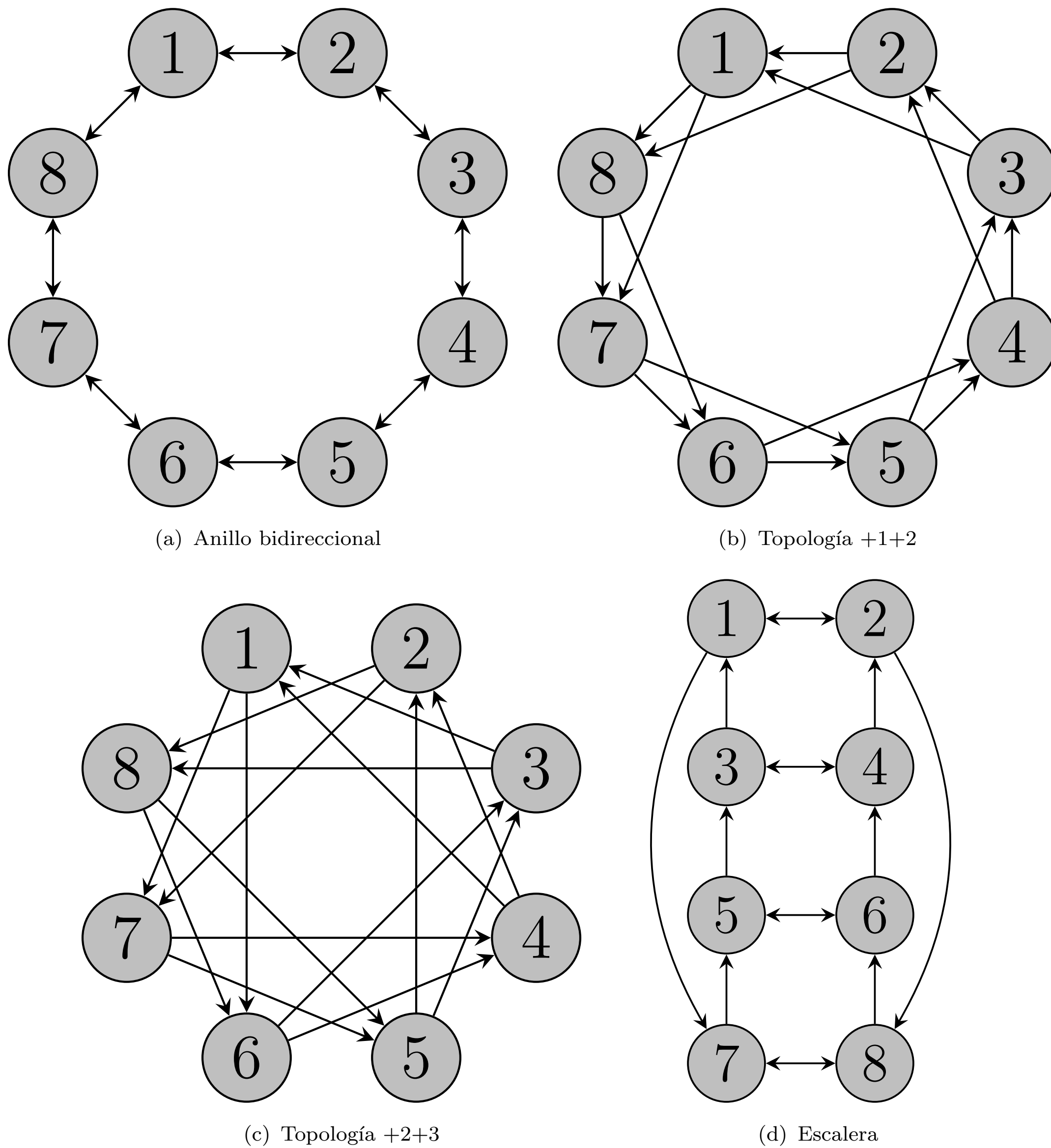
Otro punto importante asociado a la migración y la frecuencia con la que ésta sucede, son las comunicaciones entre los procesadores involucrados, ya que las comunicaciones son proporcionales a la frecuencia de la migración y el número de subpoblaciones, además de que dependiendo de la arquitectura de la computadora esto puede llegar a ser muy costoso en cuanto al tiempo de ejecución del algoritmo.

La migración entre poblaciones se puede presentar de forma síncrona o asíncrona, aunque usualmente es de forma síncrona, esto es, la migración sólo ocurre en intervalos de tiempo (generaciones) determinados, mientras que la migración asíncrona está determinada por eventos.

En cuanto a la selección de individuos para la migración, usualmente se selecciona al individuo más apto o a un grupo de los individuos con mejor aptitud de cada subpoblación y se mandan copias de estos a las subpoblaciones correspondientes. Para mantener el tamaño de las subpoblaciones constantes debido al incremento por las migraciones, usualmente se eliminan a los individuos menos aptos hasta llegar al tamaño original de la subpoblación.

Ahora, en cuanto a la forma de establecer las comunicaciones entre las subpoblaciones, se tiene que hay diferentes formas ó topologías que se pueden implementar, siendo unas mejores que otras en ciertos casos. Algunos de los ejemplos más comunes son los que se presentan en la figura 2.9.

Se debe hacer notar que algunas topologías tienen un mayor grado de conectividad entre las subpoblaciones que otras, lo que influye en la velocidad de propagación de una solución, y esto debe tomarse en cuenta en el problema que se quiera resolver, ya que puede, o no, ser conveniente que una solución se propague rápido entre las subpoblaciones provocando una convergencia. Mientras que en el caso contrario, si una solución no se propaga de forma adecuada, las posibilidades de generar nuevas y potencialmente mejores soluciones disminuye.



(a) Anillo bidireccional

(b) Topología +1+2

(c) Topología +2+3

(d) Escalera

Figura 2.9: La gráfica presenta algunos ejemplos comunes de topologías de comunicación entre subpoblaciones. La característica partícular de estos ejemplos es que todas las topologías tienen únicamente 2 vecinos.

Un estudio más detallado sobre las migraciones y las topologías de comunicación entre las subpoblaciones se puede consultar en el capítulo 6 del libro de Cantú Paz [1].

Capítulo 3

Implementación de los algoritmos genéticos en *Spin Glasses*

3.1 Implementación del problema

A lo largo del capítulo 2 se presentaron los principales elementos de un algoritmo genético, por lo que ahora sólo queda establecer las implementaciones de cada uno de estos elementos para el caso particular de los spin glasses.

En cuanto a la **codificación**, ésta es muy directa, ya que como el modelo de Edwards-Anderson trabaja con los spines de Ising $\sigma \in \{+1, -1\}$, los cuales pueden considerarse como ya codificados, así que lo único que hace falta es elegir una estructura de datos que permita manejar de forma adecuada la información de los spines de un spin glass en 2-D.

El trabajar con spin glasses en 2-D implica que hay una estructura de datos idéntica a la de una matriz cuadrada en el problema, que usualmente es convertida a una estructura en forma de cadena (arreglo), ya que este tipo de estructura de datos puede manejarse relativamente fácil en cualquier lenguaje de programación y es equivalente a la matriz, aunque para los fines de este trabajo resulta más útil preservar esta estructura de matriz. La figura 3.1 muestra la codificación de una configuración $\underline{\sigma}$ para un spin glass en 2-D con lado $L = 3$ para el caso de ambas estructuras: cadena y matriz.

Otro elemento importante que se necesita codificar e implementar sobre una estructura de datos es el que corresponde al tipo de interacciones J_{ij} entre los

$$(+1, -1, -1, +1, -1, +1, +1, +1, -1)$$

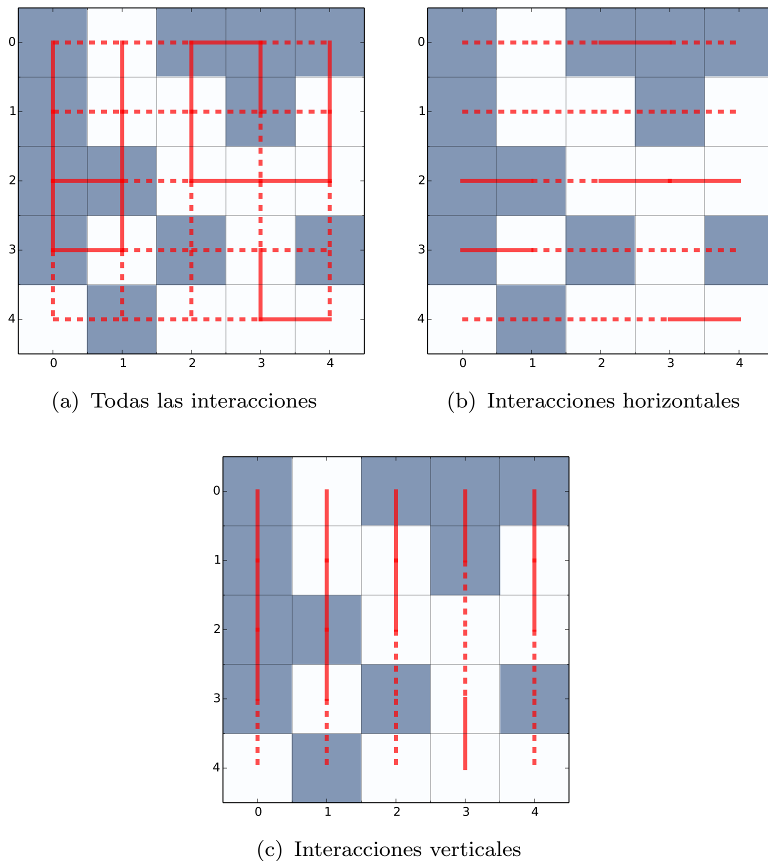
(a) Cadena (arreglo)

$$\begin{bmatrix} +1 & -1 & -1 \\ +1 & -1 & +1 \\ +1 & +1 & -1 \end{bmatrix}$$

(b) Matriz

FIGURA 3.1: Se utiliza como ejemplo una configuración de spin glass en 2-D con lado $L = 3$ para mostrar la equivalencia de las dos representaciones. Los elementos de la cadena (a) son agrupados en grupos de longitud L y son acomodados en orden dentro la matriz (b).

elementos que conforman al spin glass, ya que hay una por cada par de spines de Ising i y j. Como estas interacciones sólo pueden ser de dos tipos, ferromagnéticas ($J_{ij} > 0$) ó anti-ferromagnéticas ($J_{ij} < 0$), una codificación binaria igual a la de los spines de Ising es adecuada para el problema, de modo que ahora $J_{ij} \in \{+1, -1\}$.



(a) Todas las interacciones

(b) Interacciones horizontales

(c) Interacciones verticales

FIGURA 3.2: En la figura (a) se muestran las $2(L^2 - L)$ interacciones que hay en un spin glass en 2-D con lado $L = 5$, mientras que en (b) y (c) se muestran por separado las interacciones horizontales y verticales del spin glass.

Ahora, como se puede ver en la figura 3.2(a), hay más interacciones J_{ij} que spines de Ising, de hecho hay $2(L^2 - L)$ interacciones y L^2 spines de Ising en un spin glass en 2-D con lado L. Una forma práctica de abordar el problema de la representación para estas interacciones es considerarlas como dos conjuntos separados: el de las interacciones horizontales 3.2(b) y el de las verticales 3.2(c). De este modo, y utilizando la codificación binaria $J_{ij} \in \{+1, -1\}$, se puede guardar la información de estos dos conjuntos en dos matrices: una de tamaño $L - 1$ por L y otra de L por $L - 1$, y lo más importante es que de esta forma se preserva la posición de cada interacción J_{ij}, lo que facilita su identificación para usarlas posteriormente cuando se calculan las energías de las configuraciones.

Para ejemplificar lo anterior, usando las configuraciones de la figura 3.2 se muestran a continuación las matrices que codifican a la configuración de los spines de Ising $\underline{\sigma}$ y a las matrices que codifican a las interacciones J_{ij}.

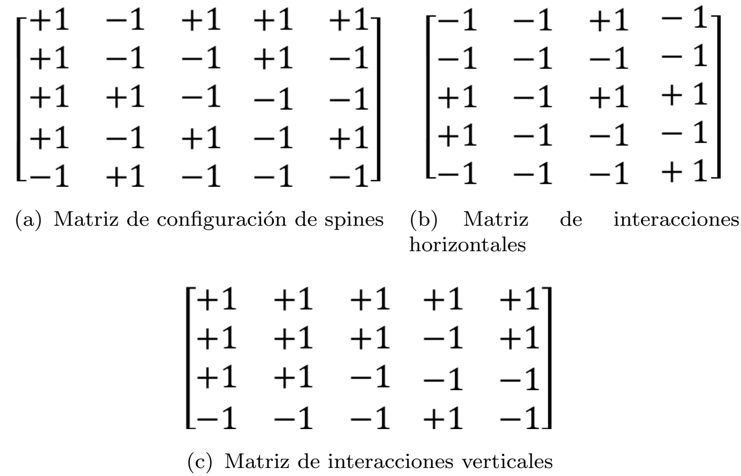


FIGURA 3.3: En la figura (a) se muestra la matríz de configuración de los spines del spin glass de la figura 3.2, mientras que en (b) y (c) se muestran las matrices de interacciones horizontales y verticales respectivamente del mismo spin glass.

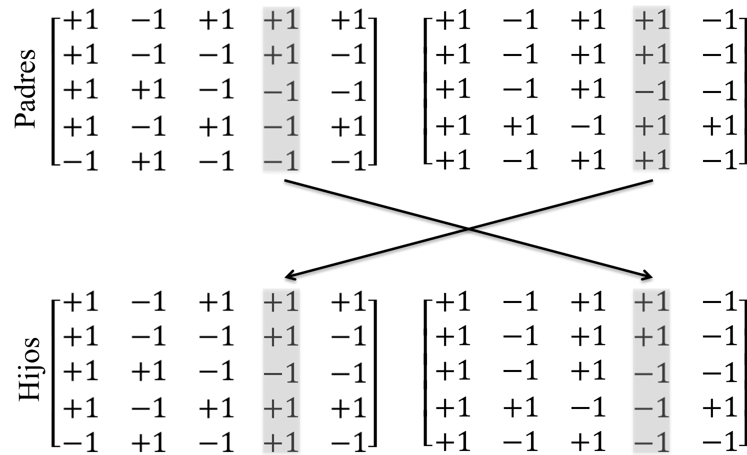
La **función de aptitud** para los spin glasses debe ser una función que permita medir el parámetro que se busca optimizar. En este caso se busca minimizar la energía de las configuraciones de espines $\underline{\sigma}$, por lo que es directo el utilizar la ecuación (1.3) como función de aptitud y ya que para este trabajo se busca conservar las fuentes de degeneración en los spin glasses, no debe considerarse campo magnético alguno, esto es $B = 0$, entonces la ecuación (1.3) se reduce a

$$E(\underline{\sigma}) = -\sum_{(ij)} J_{ij}\sigma_i\sigma_j, \qquad (3.1)$$

donde la suma es sobre todos los pares de vecinos i, j con $i \neq j$ del spin glass. La ecuación (3.1) de energía para spin glasses sin campo magnético exterior es entonces la función de aptitud para las dos implementaciones de los algoritmos genéticos, pues se tiene el objetivo de encontrar las configuraciones de espines que minimicen la energía bajo éstas condiciones.

En cuanto a los mecanismos reproductivos, se debe tomar en cuenta que los individuos sobre los cuales se va trabajar tienen una estructura de matriz (figura 3.3(a)).

Para el caso del **operador de reproducción** es conveniente establecer el operador con 2 puntos de corte para que de esta forma se puedan seleccionar y cortar un renglón o columna de forma aleatoria y después intercambiarlos con otro individuo, como se muestra en la figura 3.4.



$$\begin{bmatrix} +1 & -1 & +1 & +1 & +1 \\ +1 & -1 & -1 & +1 & -1 \\ +1 & +1 & -1 & -1 & -1 \\ +1 & -1 & +1 & -1 & +1 \\ -1 & +1 & -1 & -1 & -1 \end{bmatrix} \quad \begin{bmatrix} +1 & -1 & +1 & +1 & -1 \\ +1 & -1 & +1 & +1 & -1 \\ +1 & -1 & +1 & -1 & -1 \\ +1 & +1 & -1 & +1 & +1 \\ +1 & -1 & +1 & +1 & -1 \end{bmatrix}$$

$$\begin{bmatrix} +1 & -1 & +1 & +1 & +1 \\ +1 & -1 & -1 & +1 & -1 \\ +1 & +1 & -1 & -1 & -1 \\ +1 & -1 & +1 & +1 & +1 \\ -1 & +1 & -1 & +1 & -1 \end{bmatrix} \quad \begin{bmatrix} +1 & -1 & +1 & +1 & -1 \\ +1 & -1 & +1 & +1 & -1 \\ +1 & -1 & +1 & -1 & -1 \\ +1 & +1 & -1 & -1 & +1 \\ +1 & -1 & +1 & -1 & -1 \end{bmatrix}$$

FIGURA 3.4: Ejemplo del operador de reproducción para spin glasses. El operador selecciona un renglón o columna (el mismo para los dos individuos) de forma aleatoria y los intercambia. En esta figura se muestra el caso de una columna, mientras que para los renglones el proceso es análogo.

El **operador de mutación** se puede implementar de forma similar al operador de reproducción: se selecciona de forma aleatoria un renglón o una columna del individuo y al mutarla, ésta se reemplaza por una nueva (figura 3.5). Hay que remarcar que en este caso la mutación no es hacer el cambio de los +1 por -1 y viceversa, sino que es generar un nuevo conjunto de spines para el renglón o columna que se vaya a mutar.

Debido a que la mutación puede ser muy destructiva, en el sentido de que tiende a empeorar la calidad de las soluciones encontradas, ésta se implementa con

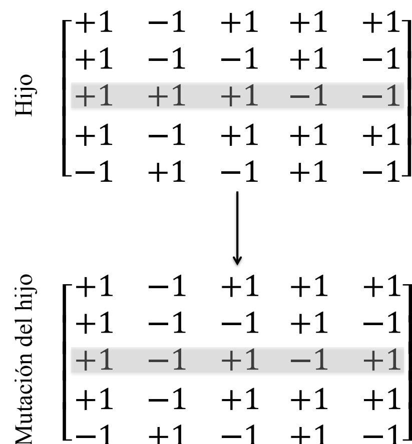


Figura 3.5: Ejemplo del operador de mutación para spin glasses. El operador selecciona un renglón o columna de forma aleatoria y la muta remplazándola con una nueva. En esta figura se muestra el caso de un renglón, mientras que para las columnas el proceso es análogo.

un porcentaje de probabilidad bajo, que para este trabajo es del 5%. Pero es importante mencionar que esta es necesaria, ya que sirve como fuente de diversidad para la población y de este modo poder explorar nuevas regiones del espacio de búsqueda.

Ahora, como **método de selección** se eligió implementar el *método de torneo*, ya que es de los métodos más comunes dentro de los algoritmos genéticos. Como ya se mencionó antes, este método consiste en seleccionar k individuos de la población de forma aleatoria con una distribución de probabilidad uniforme y después usar al más apto de esos k individuos como uno de los padres en el proceso de reproducción.

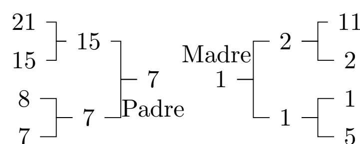


Figura 3.6: Ejemplo del método de selección por torneo. En este caso $k = 4$ en los 2 torneos, con la característica de que los 8 individuos deben ser diferentes.

Para este problema cada vez que se realiza el proceso de reproducción se implementan dos torneos con $k = 2$ y reemplazo en cada caso, con la característica adicional de que ninguno de los 4 individuos seleccionados puede ser igual al resto, ya que se busca evitar que haya padres repetidos, pues debido a que hay una distribución de probabilidad uniforme sobre la población, puede presentarse el

caso de que la selección junto con el torneo terminen con los mismos padres en una instancia. La figura 3.6 ejemplifica el proceso de selección.

Para la implementación de la **distancia Hamming** se hace un pequeño cambio en la representación de los individuos, cambiando los -1 de la matriz por 0, mientras los 1 se mantienen igual, ya que de este modo se puede calcular la distancia Hamming por medio del valor absoluto de la resta de las matrices de dos individuos diferentes (figura 3.7), que es análogo al caso de las cadenas en el capítulo 2.

$$\begin{bmatrix} +1 & -1 & +1 & +1 & +1 \\ +1 & -1 & -1 & +1 & -1 \\ +1 & +1 & +1 & -1 & -1 \\ +1 & -1 & +1 & +1 & +1 \\ -1 & +1 & -1 & +1 & -1 \end{bmatrix} \quad \begin{bmatrix} +1 & -1 & +1 & +1 & +1 \\ +1 & -1 & -1 & +1 & -1 \\ +1 & -1 & +1 & -1 & +1 \\ +1 & -1 & +1 & +1 & +1 \\ -1 & +1 & -1 & +1 & -1 \end{bmatrix}$$

$$\downarrow \qquad\qquad \downarrow$$

$$\left\| \begin{bmatrix} 1 & 0 & 1 & 1 & 1 \\ 1 & 0 & 0 & 1 & 0 \\ 1 & 1 & 1 & 0 & 0 \\ 1 & 0 & 1 & 1 & 1 \\ 0 & 1 & 0 & 1 & 0 \end{bmatrix} - \begin{bmatrix} 1 & 0 & 1 & 1 & 1 \\ 1 & 0 & 0 & 1 & 0 \\ 1 & 0 & 1 & 0 & 1 \\ 1 & 0 & 1 & 1 & 1 \\ 0 & 1 & 0 & 1 & 0 \end{bmatrix} \right\| = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

FIGURA 3.7: Ejemplo de como calcular la distancia Hamming para un spin glass. La matriz resultante de la resta contiene el número de elementos diferentes entre los dos individuos, los cuales al sumarse resultan en la distiancia Hamming entre los individuos.

El valor de la distancia Hamming se obtiene a partir de la matriz resultante de la resta; se tienen que sumar todos los 1 de esta matriz y dividir el valor obtenido entre L^2 para normalizarlo, ya que L es el tamaño del lado de la matriz.

Los casos de la **entropía** y de la **correlación de Spearman**; al ser independientes de la representación del problema, no necesitan una mayor explicación a la que se encuentra en el capítulo 2 para ambos casos. La independencia de la representación se debe a que sólo requieren de los valores que regresan las funciones de aptitud y de la distancia Hamming, así como de la distribución de los valores de aptitud de la población en el caso de la entropía, ya que usando estos elementos y las definiciones correspondientes se obtienen los resultados necesarios.

El **elitismo** es importante para asegurar que al final del algoritmo se tiene a la mejor solución encontrada, aunque como ya se discutió, éste tiene ventajas y desventajas, por lo que debe ser manejado con cuidado, al igual que la mutación. Una forma de implementar el elitismo es conservando un conjunto pequeño de individuos con los mejores valores de aptitud dentro de la población. Esta idea es la que se aplica en este trabajo y se conserva a un grupo de alrededor del 5% del

tamaño de la población, que en casos de poblaciones muy pequeñas este grupo se reduce al individuo más apto.

La **convergencia** para los objetivos de este trabajo pasa a un segundo plano, ya que lo que se busca es establecer una comparación de los efectos que tienen las diferencias en la diversidad de la población cuando se implementan los algoritmos genéticos en una versión serial y una paralela. Por esta razón no se emplea a la convergencia como un criterio de terminación para el algoritmo, y se utiliza en su lugar un número máximo de generaciones.

Todo lo anterior corresponde a lo necesario para poder implementar un algoritmo genético que pueda tratar con los spin glasses, ahora sólo queda explicar de forma detallada el algoritmo en el caso serial y hacer los ajustes correspondientes para la versión paralela.

3.2 Algoritmo genético serial

En esta sección se busca dar información detallada acerca de la estructura del algoritmo genético serial implementado para encontrar las configuraciones de mínima energía en los spin glasses, cómo se unen todos los componentes anteriores y en que orden se ejecutan.

A continuación se presenta la estructura principal del algoritmo. Las variables con una línea debajo son arreglos de una dimensión (cadenas), mientras que las que están subrayadas por dos líneas representan arreglos de dos dimensiones (matrices).

1: generaciones ← numero de generaciones
2: poblacion ← tamano de poblacion
3: repeticiones ← numero de repeticiones
4: tamano ← tamano del lado de la matriz
5: retencion pob ← 0.05
6: prob mutacion ← 0.05
7: leer matrices ← True/False
8: **if** leer matrices = True **then**
9:     inter ver ← ARCHIVO MATRIZ VERTICAL

```
10:     inter hor ← ARCHIVO MATRIZ HORIZONTAL
11: else
12:     inter ver ← GENERAR MATRIZ DE INTERACCIONES(tamano)
13:     inter hor ← GENERAR MATRIZ DE INTERACCIONES(tamano)
14: end if
15: for repeticiones ← 1, rep do
16:     pob ← GENERAR POBLACION(poblacion, tamano)
17:     for i ← 1, poblacion do
18:         aptitud poblacion[i] ← APTITUD(pob[i], tamano)
19:     end for
20:     aptitud promedio ← AGREGAR(PROMEDIO(aptitud poblacion))
21:     mejor aptitud ← AGREGAR(ORDENAR(aptitud poblacion)[0])
22:     pob ← pob[ORDENAR(aptitud poblacion)]
23:     ref reproduccion ← pob
24:     distancia ← AGREGAR(DISTANCIA HAMMING(pob))
25:     entropia ← AGREGAR(ENTROPIA(aptitud poblacion))
26:     gen ← 0
27:     while gen < generaciones do
28:         retener ← ENTERO(len(pob)·retencion pob)
29:         if retener < 1 then
30:             retener ← 1
31:         end if
32:         nueva pob ← COPIA(pob[:retener])
33:         longitud ← LONGITUD(pob)-retener
34:         evol ← OPERADOR DE REPRODUCCION(pob, tamano, longitud, prob
    mutacion)
35:         pob ← nueva pob+evol[0]
36:         for i ← 1, poblacion do
37:             aptitud poblacion[i] ← APTITUD(pob[i], tamano)
38:             if i < LONGITUD(evol[1]) then
39:                 tmp reproduccion ← AGREGAR(APTITUD(evol[1][i]))
40:                 if i < LONGITUD(evol[4]) then
41:                     tmp mutacion ← AGREGAR(APTITUD(evol[3][i]))
42:                 end if
43:             end if
44:         end for
```

```
45:         efecto reproduccion ← AGREGAR(SUMA(tmp reproduccion -
    ref reproduccion[evol[2]]) / LONGITUD(tmp reproduccion))
46:         if LONGITUD(tmp mutacion) = 0 then
47:             len mutacion ← 1
48:         else
49:             len mutacion ← LONGITUD(tmp mutacion)
50:         end if
51:         efecto mutacion ← AGREGAR(SUMA(tmp mutacion -
    tmp reproduccion[evol[4]])/ len mutacion)
52:         aptitud promedio ← AGREGAR(PROMEDIO(aptitud poblacion))
53:         mejor aptitud ← AGREGAR(ORDENAR(aptitud poblacion[0]))
54:         pob ← pob[ORDENAR(aptitud poblacion)]
55:         ref reproduccion ← pob
56:         mejor individuo ← pob[0]
57:         distancia ← AGREGAR(DISTANCIA HAMMING(pob))
58:         entropia ← AGREGAR(ENTROPIA(aptitud poblacion))
59:         gen ← gen+1
60:     end while
61:     GUARDAR SOLUCION ENCONTRADA
62:     histograma mejor aptitud ← AGREGAR(ordenar(aptitud poblacion)[0])
63:     aux aptitud ← AGREGAR(mejor aptitud)
64:     aux distancia ← AGREGAR(distancia)
65:     aux entropia ← AGREGAR(entropia)
66:     aux reproduccion ← AGREGAR(efecto reproduccion)
67:     aux mutacion ← AGREGAR(efecto mutacion)
68: end for
69: if rep > 1 then
70:     prom aptitud ← PROMEDIO(aux aptitud)
71:     prom distancia ← PROMEDIO(aux distancia)
72:     prom entropia ← PROMEDIO(aux entropia)
73:     prom reproduccion ← PROMEDIO(aux reproduccion)
74:     prom mutacion ← PROMEDIO(aux mutacion)
75: end if
76: spearman aptitud distancia ← CORRELACION SPEARMAN(aux aptitud,
    aux distancia)
77: spearman aptitud entropia ← CORRELACION SPEARMAN(aux aptitud,
```

aux entropia)

78: GUARDAR ARREGLO DE HISTOGRAMA DE MEJOR APTITUD

79: GUARDAR ARREGLOS DE PROMEDIOS Y CORRELACIONES

La implementación de este algoritmo fue realizada por completo en Python, por lo que algunas de las funciones utilizadas en el pseudocódigo anterior son funciones de módulos de Python.

El algoritmo empieza estableciendo los parámetros necesarios para definir las dimensiones del problema y el tiempo de ejecución (en número de generaciones).

En el diseño del algoritmo se tiene una opción que permite trabajar con alguna estructura específica de un spin glass por medio de dos archivos que contengan las matrices de interacciones separadas como se muestran en las figuras 3.2(b) y 3.2(c), aunque también cuenta con la opción de generarlas de forma aleatoria usando la función GENERAR MATRICES DE INTERACCIONES, que genera una matriz de forma aleatoria con $+1$ y -1 del tamaño requerido.

Con el fin de poder obtener comportamientos promedio del algoritmo genético, éste fue puesto dentro de un ciclo que permite guardar la información de cada corrida para que al final de todas las corridas se obtengan los valores promedio de toda la información recopilada.

El algoritmo genético comienza con la generación de una población de forma aleatoria usando la función GENERAR POBLACION, que es una función que a su vez llama a una función auxiliar para crear a los individuos de la población de forma similar a la función GENERAR MATRICES DE INTERACCIONES y después agruparlos en un arreglo.

Una vez que se tiene a la población, lo que sigue en el algoritmo es calcular la aptitud de los individuos en ésta para poder ordenar a la población para el proceso evolutivo y para poder calcular las medidas de diversidad implementadas. La función APTITUD trabaja de la siguiente forma:

1: **function** APTITUD(individuo, tamano)

2: total $\leftarrow 0$

3: **for** $i \leftarrow 1, tamano$ **do**

4: **for** $j \leftarrow 1, tamano - 1$ **do**

5: sig $\leftarrow j$+1

```
6:          total ← total-(inter hor[i][j]·individuo[i][j]·individuo[i][sig])
7:          total ← total-(inter ver[j][i]·individuo[j][i]·individuo[sig][i])
8:       end for
9:    end for
10:   return total
11: end function
```

En este caso las variables *inter hor* y *inter ver* corresponden a las matrices de interacciones horizontal y vertical respectivamente. Estas variables son usadas como variables globales, por lo que no es necesario listarlas como argumentos para la función.

Una vez que se tienen los valores de aptitud de la población se utilizan funciones propias de Python para ordenar a los individuos de acuerdo a su valor de aptitud. Cuando esto se termina, entonces se procede a calcular las medidas de diversidad implementadas (Entropía y distancia Hamming).

Para la distancia Hamming se utiliza la función del mismo nombre y algunas operaciones adicionales a la función que resultan más convenientes si se implementan fuera de la función que dentro. La forma de calcular la distancia Hamming es la siguiente:

```
1: temp ← 0
2: for i ← 1, poblacion-1 do
3:    temp ← temp+DISTANCIA HAMMING(pob, i)
4: end for
5: distancia ← 2·temp/(tamano·tamano·poblacion·(poblacion-1))
```

Donde la función para la distancia Hamming se define como:

```
1: function DISTANCIA HAMMING(pob, i)
2:    suma ← 0
3:    suma ← suma+SUMA(SUMA(ABS(pob[(i+1):]-pob[i])).RESHAPE(tamano·
   tamano))
4:    return suma
5: end function
```

Esta forma de calcular la distancia está diseñada para obtener la distancia promedio de la población, que es lo que se busca en este trabajo, y pensando en esto se construye una función que calcule la distancia Hamming, aprovechando la vectorización que puede ser implementada en Python. La vectorización en Python permite ejecutar instrucciones equivalentes a ciclos sobre los índices de arreglos de forma más rápida y eficiente que los mismos ciclos.

Así, para calcular la distancia Hamming promedio de la población se necesitan dos ciclos, lo cual se puede ver si se consideran a las sumas de las distancias que se calculan en la ecuación (2.6):

$$\sum_{i=1}^{m-1} \sum_{j=i+1}^{m} d_H(x_i, x_j) \qquad x_i, x_j \in \mathbb{X}, \tag{3.2}$$

Lo que se busca con la vectorización de la distancia Hamming es quitar el ciclo correspondiente al índice j, de modo que el cálculo sea más rápido a cambio de perder la información de las distancias de los pares, que no es relevante para los fines del trabajo. Al vectorizar la segunda suma se logra quitar el primer ciclo, para que sólo sea necesario uno más, lo cual permite definir la función DISTANCIA HAMMING como se muestra en el pseudocódigo anterior.

Las operaciones dentro de la función DISTANCIA HAMMING están anidadas de esa forma ya que primero se requiere obtener el número de diferencias entre el individuo actual i y los demás, de aquí que se utilice la parte ABS(*pob*[(i+1) :] − *pob*[i]); hay que recordar que cada individuo tiene la estructura de una matriz. En esta parte se encuentra la vectorización. Una vez que se tienen todas las diferencias, éstas se suman para tener una sola matriz con todas las diferencias. Ahora, esta matriz es convertida a una estructura de una cadena con la función RESHAPE, para que posteriormente sean sumadas todas sus entradas y así obtener un valor de la distancia del individuo i con los demás. Esto se repite mientras se cumpla el ciclo de i; al final, todo se suma y se normaliza para obtener la distancia Hamming promedio de la población.

El caso de la entropía es más simple, ya que hay funciones dentro de Python que ayudan a simplificar el trabajo. La rutina para la entropía es la siguiente:

```
1: function ENTROPIA(aptitud_poblacion)
2:     entropia ← 0
```

3: ent_temp ← HISTOGRAMA(aptitud poblacion, normalizado = 1)
4: **for all** $i \in ent_temp$ **do**
5: **if** $i \neq 0$ **then**
6: entropia ← entropia-$i \cdot \log_2 i$
7: **end if**
8: **end for**
9: **return** entropia
10: **end function**

La función HISTOGRAMA de Python hace todo el trabajo de clasificar los valores de aptitud de la población y de determinar el porcentaje de los mismos dentro del conjunto de valores, por lo que sólo queda utilizar de forma directa la ecuación (2.8).

En cuanto a la función OPERADOR DE REPRODUCCION, ésta lleva a cabo gran parte del proceso evolutivo, incluyendo el método de selección por torneo y al mismo tiempo se encarga de ordenar información para poder determinar posteriormente el efecto de este mismo operador junto con el operador de mutación, el cual es utilizado desde esta función. La estructura de esta función es la siguiente:

1: **function** OPERADOR DE REPRUDCCION(pob, tamano, longitud, prob mutacion)
2: hijos ← [ ]
3: **while** LONGITUD(hijos) < longitud **do**
4: indice ← TORNEO PARA SELECCIONAR UN PADRE
5: p ← pob[indice]
6: indice ← TORNEO PARA SELECCIONAR UN PADRE
7: m ← pob[indice]
8: COLECTAR INDICES E INDIVIDUOS DE LOS CASOS SELECCIONADOS
9: hijo ← [p, m]
10: corte ← RANDINT(0, tamano-1)
11: **if** RANDOM < 0.5 **then**
12: hijo[0][corte] ← m[corte]
13: hijo[1][corte] ← p[corte]
14: **else**
15: hijo[0].T[corte] ← m.T[corte]
16: hijo[1].T[corte] ← p.T[corte]

```
17:         end if
18:         if RANDOM < prob mutacion then
19:             hijo[0] ← MUTACION(hijo[0])
20:         end if
21:         hijos ← AGREGAR(hijo[0])
22:         if longitud-LONGITUD(hijos) > 0 then
23:             if RANDOM < prob mutacion then
24:                 hijo[1] ← MUTACION(hijo[1])
25:             end if
26:             hijos ← AGREGAR(hijo[1])
27:         end if
28:         COLECTAR INDICES E INDIVIDUOS DE LOS CASOS MUTADOS
29:     end while
30:     return hijos, individuos seleccionados, indices seleccionados,
    individuos mutados, indices mutados
31: end function
```

Como ya se mencionó, esta función es de gran importancia por la gran cantidad de tareas que realiza, siendo el proceso de reproducción la principal. Este proceso se lleva a cabo implementando el método de selección por torneo para cada padre con las características mencionadas en la sección anterior, mientras que la reproducción transcurre entre las lineas 9 a 17, donde las notaciones con T como $\underline{\underline{m}}.T[corte]$ en Python se refieren a la transposición de las matrices, ya que de esta forma se pueden intercambiar los renglones de manera análoga a las columnas que son los casos sin T, siendo $[corte]$ la notación para el renglón o columna en cuestión.

Posteriormente se tiene a la mutación, la cual se implementa de la siguiente forma:

```
 1: function OPERADOR DE MUTACION(individuo, tamano)
 2:     if RANDOM < 0.5 then
 3:         individuo[RANDINT(0, tamano-1)] ←GENERAR RENGLON ALEATORIO
 4:     else
 5:         individuo.T[RANDINT(0, tamano-1)] ←GENERAR RENGLON
    ALEATORIO
 6:     end if
 7:     return individuo
 8: end function
```

Esta rutina es más simple y de nuevo se utiliza la transposición de las matrices para poder efectuar de forma análoga la mutación de un renglón ó una columna según sea el caso, con un 50% de probabilidad para cada caso, ya sean los renglones ó columnas que se mutan son seleccionados de forma aleatoria.

Ahora, considerando de nuevo a la función OPERADOR DE REPRODUCCIÓN, se tiene que los elementos que regresa la función son varios. Se regresan los nuevos individuos de la población (*hijos*) y también se regresan índices sobre los individuos seleccionados para la reproducción (*indices seleccionados*) y copias de estos (*individuos seleccionados*), lo mismo se hace con los individuos sujetos a mutación (*individuos mutados* e *indices mutados*). Todo esta información adicional es requerida para después poder calcular los efectos de cada operador sobre la aptitud de los individuos que fueron sujetos a sus efectos. Estos valores son asignados a otro arreglo de una dimensión llamado *evol*.

La parte restante del pseudocódigo del algoritmo genético serial no requiere de explicaciones mayores, ya que sólo son copias y almacenamiento de variables, mientras que para los cálculos de la correlación de Spearman se utiliza una función predeterminada de Python que calcula esta correlación. Finalmente la información necesaria es almacenada en archivos de texto.

3.3 Algoritmo genético paralelo

De manera análoga a la sección anterior, esta sección tiene el propósito de describir de forma detallada la versión paralela del algoritmo genético implementado para el problema de los spin glasses.

Es importante destacar que la versión paralela del algoritmo está diseñada para una arquitectura de memoria compartida y utiliza el modelo SIMD en la paralelización.

Utilizando la misma notación de variables subrayadas por una línea para cadenas y por dos líneas para matrices. Se presenta a continuación la estructura principal del algoritmo genético paralelo:

1: generaciones ← numero de generaciones
2: poblacion ← tamano de poblacion
3: subp ← numero de subpoblaciones

```
4: repeticiones ← numero de repeticiones
5: tamano ← tamano del lado de la matriz
6: retencion pob ← 0.05
7: prob mutacion ← 0.05
8: migracion ← frecuencia de migracion en generaciones
9: procesadores ← numero de procesadores a usar
10: leer matrices ← True/False
11: if leer matrices = True then
12:     inter ver ← ARCHIVO MATRIZ VERTICAL
13:     inter hor ← ARCHIVO MATRIZ HORIZONTAL
14: else
15:     inter ver ← GENERAR MATRIZ DE INTERACCIONES(tamano)
16:     inter hor ← GENERAR MATRIZ DE INTERACCIONES(tamano)
17: end if
18: for repeticiones ← 1, rep do
19:     mejor aptitud ← [ ]
20:     temp reproduccion ← [ ]
21:     temp mutacion ← [ ]
22:     efecto reproduccion ← 0
23:     efecto mutacion ← 0
24:     pob ← GENERAR SUBPOBLACIONES EQUILIBRADAS
25:     for i ← 0, subp-1 do
26:         for ii ← 0, procesadores-1 do
27:             var tmp ← AGREGAR(PROCESADOR ii(APTITUD POB(pob[i], ii,
    tamano)))
28:         end for
29:         pob global ← AGREGAR(pob[i])
30:         aptitud global ← AGREGAR(ORDENAR(var tmp[0]))
31:         subp mejor aptitud[i] ← ORDENAR(var tmp[0])[0]
32:         pob[i] ← pob[i][ORDENAR(var tmp[0])]
33:         ref reproduccion[i] ← ORDENAR(var tmp[0])
34:         var tmp ← [ ]
35:     end for
36:     for ii ← 0, procesadores-1 do
37:         var tmp ← AGREGAR(PROCESADOR ii(DISTANCIA HAMMING(
    pob global, ii, tamano)))
```

```
38:     end for
39:     var tmp ← 2·SUMA(var tmp)/(tamano·tamano·poblacion·(poblacion-1))
40:     distancia ← AGREGAR(var tmp)
41:     var tmp ← [ ]
42:     entropia ← AGREGAR(ENTROPIA(aptitud global))
43:     mejor individuo ← AGREGAR(SELECCIONAR MEJOR INDIVIDUO)
44:     mejor aptitud ← AGREGAR(SELECCIONAR MEJOR APTITUD)
45:     gen ← 0
46:     while gen < generaciones do
47:         for i ← 0, subp-1 do
48:             retener ← ENTERO(len(pob[i])·retencion pob)
49:             if retener < 1 then
50:                 retener ← 1
51:             end if
52:             nueva pob ← COPIA(pob[i][:retener])
53:             longitud[i] ← LONGITUD(pob[i])-retener
54:             evol[i] ← OPERADOR DE REPRODUCCION(pob[i], tamano,
    longitud[i], prob mutacion)
55:             pob[i] ← nueva pob+evol[i][0]
56:         end for
57:         pob global ← [ ]
58:         for i ← 0, subp-1 do
59:             for ii ← 0, procesadores-1 do
60:                 var tmp ← AGREGAR(PROCESADOR ii(APTITUD POB(pob[i], ii,
    tamano, evol[i][1], evol[i][3])))
61:             end for
62:             pob global ← AGREGAR(pob[i])
63:             aptitud global ← AGREGAR(ORDENAR(var tmp[0]))
64:             subp mejor aptitud[i] ← ORDENAR(var tmp[0])[0]
65:             pob[i] ← pob[i][ORDENAR(var tmp[0])]
66:             efecto reproduccion[gen] ← efecto reproduccion[gen]+(SUMA(
    var tmp[1] - ref reproduccion[i][evol[i][2]]) / LONGITUD(var tmp[1]))
67:             if LONGITUD(evol[i][4]) ≠ 0 then
68:                 efecto mutacion[gen] ← efecto mutacion[gen]+(SUMA(
    var tmp[2] - var tmp[1][evol[i][4]])/ LONGITUD(evol[i][4]))
69:             else
```

```
70:             efecto mutacion tmp[gen] ← 0
71:          end if
72:          ref reproduccion[i] ← ORDENAR(var tmp[0])
73:          var tmp ← [ ]
74:       end for
75:       efecto reproduccion[gen] ← efecto reproduccion[gen]/subp
76:       efecto mutacion[gen] ← efecto mutacion[gen]/subp
77:       for ii ← 0, procesadores-1 do
78:          var tmp ← AGREGAR(PROCESADOR ii(DISTANCIA HAMMING(
   pob global, ii, tamano)))
79:       end for
80:       var tmp ← 2·SUMA(var tmp)/(tamano·tamano·poblacion·(poblacion-
   1))
81:       distancia ← AGREGAR(var tmp)
82:       var tmp ← [ ]
83:       entropia ← AGREGAR(ENTROPIA(aptitud global))
84:       mejor individuo ← AGREGAR(SELECCIONAR MEJOR INDIVIDUO)
85:       mejor aptitud ← AGREGAR(SELECCIONAR MEJOR APTITUD)
86:       if MOD(gen + 1, migracion) = 0 then
87:          MIGRACION ENTRE SUBPOBLACIONES
88:       end if
89:       gen ← gen+1
90:    end while
91:    GUARDAR SOLUCION ENCONTRADA
92:    histograma mejor aptitud ← AGREGAR(ordenar(aptitud global)[0])
93:    aux aptitud ← AGREGAR(mejor aptitud)
94:    aux distancia ← AGREGAR(distancia)
95:    aux entropia ← AGREGAR(entropia)
96:    aux reproduccion ← AGREGAR(efecto reproduccion)
97:    aux mutacion ← AGREGAR(efecto mutacion)
98: end for
99: if rep > 1 then
100:    prom aptitud ← PROMEDIO(aux aptitud)
101:    prom distancia ← PROMEDIO(aux distancia)
102:    prom entropia ← PROMEDIO(aux entropia)
103:    prom reproduccion ← PROMEDIO(aux reproduccion)
```

104: prom_mutacion ← PROMEDIO(aux_mutacion)
105: **end if**
106: spearman_aptitud_distancia ← CORRELACION_SPEARMAN(aux_aptitud, aux_distancia)
107: spearman_aptitud_entropia ← CORRELACION_SPEARMAN(aux_aptitud, aux_entropia)
108: GUARDAR ARREGLO DE HISTOGRAMA DE MEJOR APTITUD
109: GUARDAR ARREGLOS DE PROMEDIOS Y CORRELACIONES

La estructura de este algoritmo es muy similar a la del algoritmo serial, por lo que sólo se hace énfasis en las diferencias originadas por la paralelización del algoritmo.

Una de las principales diferencias se debe a las múltiples poblaciones que se tienen en la versión paralela, ya que éstas permiten implementar los esquemas de migración, que son de gran importancia en el desempeño y comportamiento del algoritmo.

Se hace notar que la paralelización de este algoritmo, al estar diseñada para una arquitectura de memoria compartida, se enfoca en las partes que son más costosas en cuanto a recursos computacionales, ya que de esta forma los procesadores encuentran disponible la información que necesitan para cuando sean requeridos utilizando el modelo SIMD, lo que significa un mayor poder computacional para las partes del algoritmo más exigentes.

Un ejemplo de cómo se emplea esta idea de paralelización es con la función de aptitud, ya que es la misma función que en el caso serial, pero se implementa de diferente forma para poder obtener un mayor beneficio de la paralelización, como sigue:

1: **for** $ii \leftarrow 0$, procesadores-1 **do**
2: var_tmp ← AGREGAR(PROCESADOR-ii(APTITUD_POB(sub_pob[i], ii, tamano, pob_rep(*opcional*), pob_mut(*opcional*))))
3: **end for**

Las instrucciones anteriores son utilizadas para crear la región paralela, ya que cada llamada de la función PROCESADOR-ii hace que el ii-ésimo procesador se encargue de una parte del trabajo de calcular la aptitud de una subpoblación. En la siguiente parte se muestra como es que se divide de forma automática la carga

de trabajo sin haber comunicaciones entre los procesadores, sólo se comunican hasta el final de la rutina cuando se regresan los valores calculados.

```
1: function APTITUD POB(sub pob, ii, tamano, pob rep = [ ], pob mut = [ ])
2:     apt tmp ← [ ]
3:     rep tmp ← [ ]
4:     mut tmp ← [ ]
5:     for i ← ii, LONGITUD(sub pob), i += procesadores do
6:         apt tmp ← AGREGAR(APTITUD(sub pob[i], tamano, [i]))
7:         if LONGITUD(pob rep) > 0 then
8:             rep tmp ← AGREGAR(APTITUD(pob rep[i], tamano, [i]))
9:             if LONGITUD(pob mut) > 0 then
10:                mut tmp ← AGREGAR(APTITUD(pob mut[i], tamano, [i]))
11:            end if
12:        end if
13:    end for
14:    return apt tmp, rep tmp, mut tmp
15: end function
```

La función anterior (auxiliar) para distribuir los cálculos de aptitud de una subpoblación está diseñada para utilizar todos los recursos disponibles sólo para este trabajo, incluyendo los casos en los que se necesiten los cálculos de aptitud de los individuos que fueron sujetos a los operadores de reproducción y mutación para poder medir los efectos de los operadores.

Un punto importante que se debe tener en cuenta es que la paralelización es asíncrona; esto significa que se reducen los tiempos de espera, pero se tiene la desventaja de que la información se almacena de forma desordenada y si no se tiene el cuidado necesario los resultados pueden ser afectados. Para evitar este problema, se utiliza un índice auxiliar que mantiene el control sobre el orden de la información, sin tener que trabajar de forma síncrona. Estos índices auxiliares son los correspondientes al argumento $[i]$ (el tercero) de la función APTITUD.

El diseño de una función auxiliar con estas características cobra mayor importancia cuando se trabaja con poblaciones muy grandes, ó cuando el tamaño establecido para los individuos es grande, pues a mayor tamaño hay un mayor número de interacciones que se deben calcular.

Para el caso de la distancia Hamming era necesario tener a todos los individuos en un sólo arreglo para poder facilitar los cálculos en la implementación paralela. La paralelización de esta función busca tomar ventaja de la estructura de los datos y de la vectorización en Python para distribuir operaciones sobre arreglos (la estructura de datos), de modo que el cálculo de la distancia Hamming se realiza de la siguiente forma:

1: **for** *ii* ← 0, procesadores-1 **do**
2: var tmp ← AGREGAR(PROCESADOR-*ii*(DISTANCIA HAMMING(pob, *ii*)))
3: **end for**
4: distancia ← AGREGAR(2·SUMA(var tmp)/(tamano·tamano·poblacion·(poblacion-1)))
5: var tmp ← [ ]

Con la función para la distancia Hamming definida como:

1: **function** DISTANCIA HAMMING(pob, *ii*)
2: suma ← 0
3: **for** *i* ← *ii*, LONGITUD(pob)-1, *i* += *procesadores* **do**
4: suma ← suma+SUMA(SUMA(ABS(pob[(*i*+1):]-pob[*i*])).RESHAPE(tamano·tamano))
5: **end for**
6: **return** suma
7: **end function**

Con esta forma de calcular la distancia Hamming, se busca reducir los tiempos de cómputo por medio de la distribución de operaciones y la paralelización que tampoco requieren de comunicaciones durante los cálculos, ya que cada procesador tiene predeterminada la fracción de trabajo que debe realizar. La forma en la que se reparte el trabajo está determinada por medio del *for* de la función Hamming y el número de procesador *ii*, ya que con esto se divide el arreglo que contiene a todos los individuos de todas las subpoblaciones y cada procesador trabaja de forma asíncrona sobre diferentes regiones del arreglo, lo que disminuye el tiempo de cálculo.

En cuanto al proceso evolutivo, este se implementa de forma análoga al caso serial en cada subpoblación, por lo que debe de ir acompañado de un ciclo *for*.

El cálculo de la entropía es exactamente igual que el caso serial, pero utilizando un arreglo que contenga a todos los individuos de todas las subpoblaciones.

Para el esquema de migración en el algoritmo se eligió utilizar como topología de comunicación al anillo bidireccional (figura 2.9(a)), ya que resulta fácil su implementación para cualquier número de subpoblaciones mayor o igual a 2. La forma de implementar la MIGRACION ENTRE SUBPOBLACIONES es la siguiente:

```
1: lista temp ← [ ]
2: for i ← 0, subp-1 do
3:     lista temp ← pob[i][0]
4: end for
5: for i ← 0, subp-1 do
6:     indice_der ← BUSCAR POSICION EN ARREGLO(aptitud pob[i], aptitud pob[
   MOD(i+1, subp)][0])
7:     indice_izq ← BUSCAR POSICION EN ARREGLO(aptitud pob[i], aptitud pob[
   MOD(i-1, subp)][0])
8:     pob[i] ← ELIMINAR ULTIMO ELEMENTO(pob[i])
9:     pob[i] ← AGREGAR ELEMENTO(indice_der, lista temp[MOD(i+1, subp)])
10:    pob[i] ← ELIMINAR ULTIMO ELEMENTO(pob[i])
11:    pob[i] ← AGREGAR ELEMENTO(indice_izq, lista temp[MOD(i-1, subp)])
12: end for
```

La idea del pseudocódigo anterior es la de mostrar cómo son las comunicaciones entre las subpoblaciones, así como la atención que se debe tener cuando se realiza migración para reemplazar de forma correcta a los individuos. Con el fin de mantener el control en la migración se utilizan las variables auxiliares *lista temp*, *indice_der* e *indice_izq*, donde la primera se encarga de colectar el mejor individuo de cada subpoblación y las otras dos sirven para identificar el lugar en el cual se agregan los individuos que migran de las subpoblaciones que se encuentren a la derecha e izquierda de la subpoblación actual, de acuerdo a la topología de anillo bidireccional (figura 2.9(a)).

El resto del algoritmo es análogo a la versión serial, por lo que no es necesario explicar detalladamente la parte final de este.

3.4 Equipo utilizado

Tanto para la versión serial como para la paralela se utilizó el mismo equipo. Éste cuenta con dos 2 procesadores Intel Xeon E5620 con una frecuencia de reloj de 2.40GHz, cada uno de los cuales cuenta con 8 núcleos, por lo que en total se tienen 16 núcleos disponibles. El equipo cuenta con una arquitectura de memoria compartida (figura 1.6), con un total de 16GB de memoria RAM. Como sistema operativo del equipo está basado en Linux, utilizando la version 12.04 LTS de Ubuntu. El sistema operativo permite ambientes de programación como Python. La versión 2.7 de Python fue utilizada para ambas implementaciones (serial y paralela), mientras que el módulo *multiprocessing* fue la herramienta utilizada para implementar la paralelización.

Capítulo 4

Resultados

4.1 Eficiencia computacional

En esta sección se presentan los resultados relacionados con la eficiencia computacional, ya que es una parte importante dentro de este trabajo. A lo largo de éste, se ha hecho énfasis en la complejidad intrínseca del problema de minimizar la energía de un spin glass y la necesidad de un mayor poder computacional para reducir los tiempos de espera para obtener un resultado, por lo que para poder mostrar de mejor forma las diferencias entre el algoritmo serial y el paralelo se realizaron diferentes pruebas donde se modificaron los parámetros, que tienen una gran influencia sobre el desempeño computacional en ambos algoritmos.

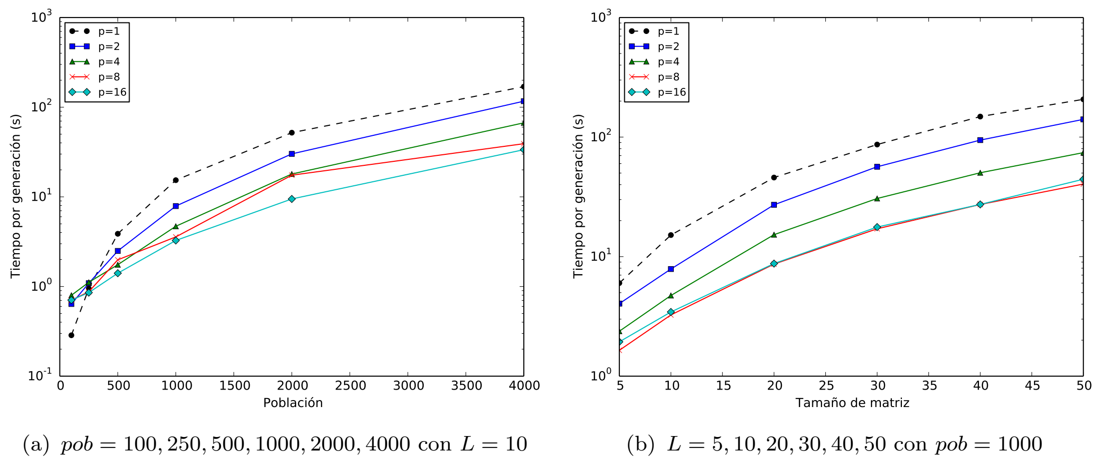
Estos parámetros son el tamaño de la población utilizada y de la matriz del problema, ya que éstos son de las principales fuentes de la carga computacional. Por esta razón, en esta sección se presenta un análisis de la eficiencia computacional de los algoritmos en función de estos parámetros, comenzando con el tamaño de la población y posteriormente con el tamaño de la matriz.

Para las pruebas relacionadas con el tamaño de la población, se consideraron los tamaños de población $pob = 100, 250, 500, 1000, 2000$ y 4000, dejando fijo $L = 10$. Los tamaños considerados para la matriz son $L = 5, 10, 20, 30, 40$ y 50 con $pob = 1000$, utilizando 1 procesador en el caso serial, mientras que para el caso paralelo se utilizaron 2, 4, 8 y 16 procesadores.

Para cada una de las combinaciones anteriores se realizaron 10 pruebas independientes, donde cada prueba consistía en 10 generaciones, con lo que se

obtuvieron 100 datos por combinación, los cuales se promediaron para obtener información sobre el desempeño de los algoritmos.

A continuación se muestra una gráfica donde se observan las diferencias en los tiempos de ejecución del algoritmo, utilizando los parámetros anteriores en cada caso.



(a) $pob = 100, 250, 500, 1000, 2000, 4000$ con $L = 10$

(b) $L = 5, 10, 20, 30, 40, 50$ con $pob = 1000$

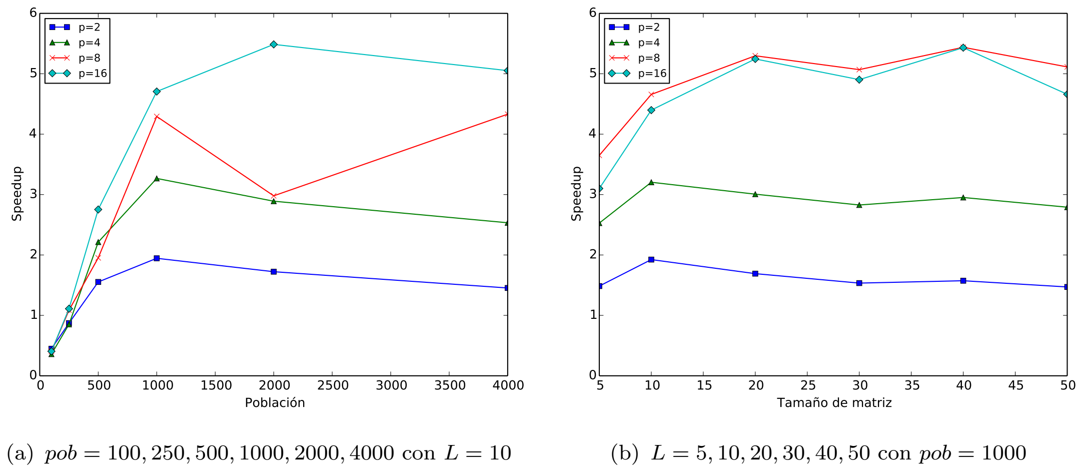
FIGURA 4.1: Tiempos de ejecución por generación en función del tamaño de la población (a) y del tamaño de la matriz (b) para el algoritmo serial $p = 1$ y el algoritmo paralelo $p = 2, 4, 8, 16$.

En este caso se puede ver que el tamaño de la población influye en el tiempo de forma significativa y de la misma forma el tamaño de la matriz, aunque en ninguno de los dos casos el tiempo se incrementa de forma exponencial con el parámetro en cuestión, ya que la escala del tiempo es logarítmica y en ningún caso de la figura 4.1 se observa una línea recta.

Hay que hacer notar que las diferencias de tiempo son pequeñas porque se consideran tiempos por generación, de modo que la diferencia de tiempo real en una implementación es mucho mayor, ya que ésta se incrementa con el número de generaciones en dicha implementación. Por ejemplo, realizar 100 corridas de 150 generaciones para obtener promedios como en la siguiente sección, para el caso de $pob = 4000$ con $L = 10$ la versión serial tardaría 29.43 días, mientras que en con la versión paralela se puede reducir el tiempo hasta 5.82 días. Mientras que para el caso de $L = 50$ con $pob = 1000$, el tiempo de la versión serial sería de 35.91 días, y el paralelo de 7.02 días.

Otra forma para poder apreciar las diferencias en el tiempo de ejecución de los algoritmos es mediante el *speedup* que se muestra en la figura 4.2 y que es el factor

que dice qué tan rápida es la versión paralela comparada con la serial, calculada de acuerdo a la ecuación (1.4).



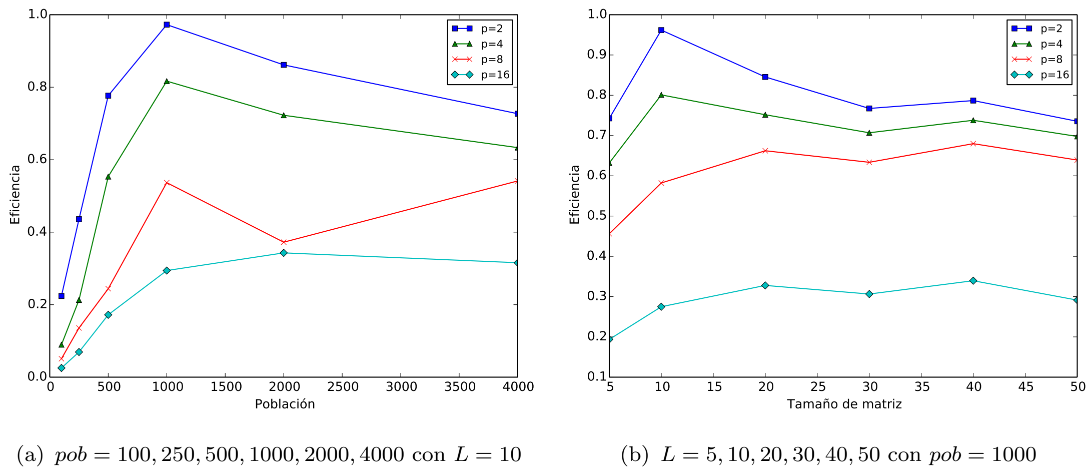
(a) $pob = 100, 250, 500, 1000, 2000, 4000$ con $L = 10$ (b) $L = 5, 10, 20, 30, 40, 50$ con $pob = 1000$

FIGURA 4.2: *Speedup* en función del tamaño de la poblacion (a) y del tamaño de la matriz (b) para el algoritmo paralelo con $p = 2, 4, 8, 16$.

Es importante notar en la figura 4.2, que para los casos de poblaciones pequeñas (100 y 250) el speedup es menor que 1, lo que quiere decir que las implementaciones paralelas del algoritmo son más lentas que la serial en estos casos, mientras que esto cambia y se magnifica con las poblaciones grandes. Esto implica que el costo de las llamadas a los procesadores, así como los tiempos de espera son considerables cuando no hay mucha carga computacional, mientras que esto se vuelve menos importante cuando la carga computacional es grande, lo que se puede ver con el speedup de los casos con poblaciones más grandes donde éste llega a ser mayor a 5 para el caso de $p = 16$.

En el caso de los diferentes tamaños de matriz, como se tiene una población grande, si se compara con las poblaciones pequeñas de la figura 4.2(a), se puede entender que aunque sean matrices de un tamaño pequeño, el algoritmo paralelo sea más rápido que el serial y por lo tanto tener un speedup mayor a 1 desde las primeras instancias, aunque se hace notar que el speedup que se obtuvo para el caso $p = 16$, pues resultó ser inferior al caso $p = 8$, mientras que para los diferentes tamaños de poblaciones se siguió la tendencia de un mayor speedup con un mayor número de procesadores.

Ahora, aunque se logren tener valores muy grandes en el speedup, también se debe tener en cuenta la eficiencia del algoritmo en cuestión, ya que aunque se pueden reducir los tiempos de ejecución aumentando el número de procesadores



(a) $pob = 100, 250, 500, 1000, 2000, 4000$ con $L = 10$ (b) $L = 5, 10, 20, 30, 40, 50$ con $pob = 1000$

FIGURA 4.3: Eficiencia en función del tamaño de la poblacion (a) y del tamaño de la matriz (b) para el algoritmo paralelo con $p = 2, 4, 8, 16$.

(hasta cierto punto), esto no es necesariamente lo óptimo, pues se puede estar desperdiciando la capacidad de los procesadores. En la figura 4.3 se muestra la eficiencia correspondiente a cada una de las cuatro implementaciones paralelas ($p = 2, 4, 8, 16$).

Si se hacen las comparaciones entre las figuras 4.2 y 4.3 se tiene que aunque el speedup es mayor cuando $p = 16$, la eficiencia es mayor cuando $p = 2$, por lo que se deben considerar estos dos factores cuando se busca implementar de la mejor forma posible un algoritmo paralelo. Por ejemplo, un buen equilibrio es el que se tiene cuando $p = 8$ y $pob = 1000$, ya que muestra una eficiencia superior al 50% para tamaños de matriz $L \geq 10$.

4.2 Diversidad y resultados de los algoritmos

El objetivo principal de este trabajo es el de comparar el desempeño de los algoritmos genéticos en su versión serial y paralela, utilizando herramientas auxiliares como la diversidad para tratar de identificar las diferencias en el desempeño de las dos versiones del algoritmo.

Para poder establecer el número de generaciones y la frecuencia de las migraciones necesarias en las implementaciones de los spin glasses, se realizaron pruebas de control utilizando el modelo de Ising con los parámetros $pob = 1000$ y $L = 10$, y en el caso paralelo $p = 8$. Estos parámetros corresponden a los de la sección

anterior, ya que estos presentaban el mejor aprovechamiento de los recursos computacionales.

Además de todo lo anterior, en el caso paralelo se tiene que la población $pob = 1000$ es dividida en 4 subpoblaciones aisladas de 250 individuos cada una, aunque para los cálculos de diversidad se utilizan copias de todas estas subpoblaciones para tener de vuelta una sola de 1000 individuos y poder realizar las comparaciones necesarias con el caso serial.

Como resultados de las pruebas de control, se estableció el número de generaciones $gen = 150$ y la frecuencia de las migraciones como 20, con los cuales se realizaron 100 corridas independientes, encontrando la configuración del estado base del modelo de Ising en un 17% de las veces en el algoritmo serial, mientras que en el caso paralelo fue un 40%.

Una vez establecidos los parámetros para los algoritmos (serial y paralelo), se llevaron a cabo 100 corridas independientes para los spin glasses en cada caso, con lo que se obtuvieron los comportamientos promedios que se presentan en esta sección.

La primera de las comparaciones entre el algoritmo genético serial y la versión paralela es la correspondiente al comportamiento de la aptitud promedio de las mejores soluciones encontradas, ya que esto presenta una primera noción sobre el comportamiento del algoritmo para encontrar la mejor solución.

Aunque las diferencias en la figura 4.4 no sean tan marcadas entre ambas implementaciones, se puede notar que en el caso paralelo, el descenso de la energía es más suave y al final el valor del promedio de la energía es un poco menor que en el caso serial. Con esto se puede inferir que en promedio, el algoritmo genético paralelo converge después que el caso serial, permitiéndole encontrar mejores soluciones (de menor energía) de forma más frecuente.

Para mostrar lo anterior, en la figura 4.5 se presentan los histogramas de las mejores soluciones encontradas correspondientes a la figura 4.4. De este modo se puede ver de forma más clara las diferencias en el desempeño de ambos algoritmos, ya que aunque en ambos casos la mejor solución encontrada tiene la misma energía, la frecuencia con la que se encuentra es diferente. Además, en el caso paralelo la distribución de las soluciones encontradas es más estrecha y ésta centrada en $E = -124.90 \pm 2.94$, mientras que en el caso serial $E = -123.50 \pm 2.99$.

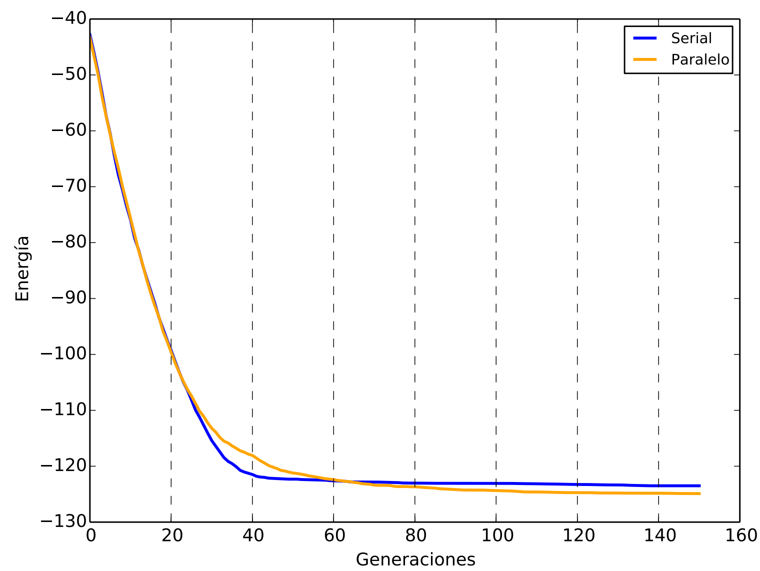
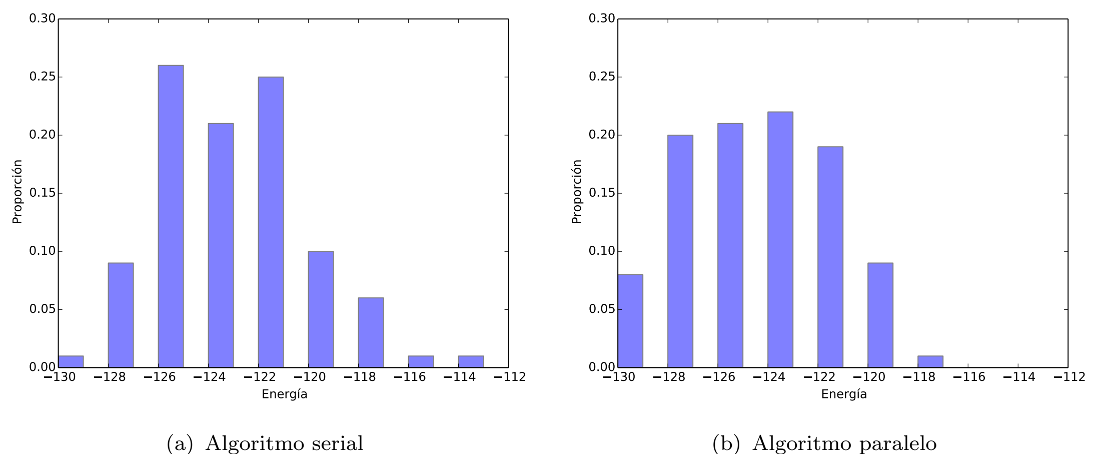


FIGURA 4.4: Energía promedio de las mejores soluciones encontradas en cada generación sobre las 100 pruebas realizadas. Las líneas punteadas indican las generaciones en las que hubo migración para el caso paralelo.



(a) Algoritmo serial

(b) Algoritmo paralelo

FIGURA 4.5: Histograma de las mejores soluciones encontradas en cada una de las 100 pruebas realizadas.

Con esto se tiene que el algoritmo genético paralelo no sólo es más rápido que la versión serial, sino que también presenta mayores posibilidades de encontrar una mejor solución.

Ahora, con los siguientes resultados se trata de encontrar la relación que existe entre el desempeño de los algoritmos encontrando las mejores soluciones de forma más frecuente y la diversidad de las poblaciones con las que trabajan. Para esto se utilizan las herramientas del capítulo 2: distancia Hamming, Entropía y finalmente la correlación Spearman para medir la relación que hay entre las dos anteriores y la optimización de la energía.

Los primeros resultados que se muestran en la figura 4.6 son los correspondientes a la distancia Hamming.

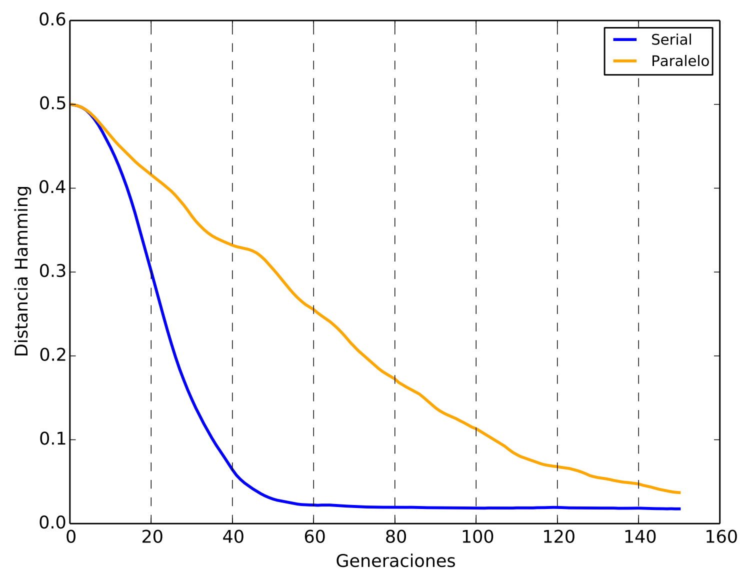


Figura 4.6: Distancia Hamming promedio de las poblaciones de los algoritmos genéticos serial y paralelo sobre las 100 pruebas realizadas. Las líneas punteadas indican las generaciones en las que hubo migración para el caso paralelo.

Las diferencias que existen en el comportamiento de la distancia Hamming en el algoritmo serial comparado con el paralelo son muy claras, ya que en el caso serial la medida de la distancia Hamming decae de forma similar a una campana Gaussiana, mientras que en el caso paralelo podría ser incluso un decaimiento lineal la mayor parte del tiempo. Entonces, apoyado en las gráficas de las figuras 4.5 y 4.6, se puede inferir que una diversidad relativamente alta a lo largo del algoritmo genético conduce a mejores resultados.

Ahora, el comportamiento de la distancia Hamming en el caso paralelo se debe a los esquemas de migración que se implementan, y la forma de asociarlo es pensando de la siguiente forma:

Primero, en este caso se tienen 4 subpoblaciones de 250 individuos que evolucionan de forma aislada durante intervalos de tiempo (generaciones) predeterminados y la naturaleza estocástica del algoritmo puede hacer que cada una de estas subpoblaciones se dirija por caminos diferentes, lo que se simula diferentes grupos de cromosomas en los individuos de las diferentes poblaciones.

Segundo, realizar la migración se seleccionan a los individuos con las mejores características de cada subpoblación, los cuales pueden ser muy diferentes a los individuos de las nuevas poblaciones a donde van a llegar, lo que permite que en las subsecuentes generaciones estas características se extiendan dentro de la subpoblación, evitando que la diversidad de éstas decaiga de forma abrupta, aunque al final ésta tiene que disminuir para poder converger a una solución.

Los resultados obtenidos al calcular la entropía de las poblaciones en ambos algoritmos también muestran resultados similares, los cuales se pueden ver en la figura 4.7.

Se puede ver que hay una gran diferencia entre los comportamientos de la entropía de la población del algoritmo serial y del paralelo; sólo que la entropía parece ser más sensible a la influencia de los esquemas de migración, ya que en la figura 4.7 se pueden ver incrementos de la entropía en las generaciones posteriores a las migraciones, que se identifican por las líneas punteadas en la gráfica. De nuevo, al mantener una entropía alta durante el algoritmo, se tiene una mejor exploración del espacio de búsqueda y por lo tanto aumentan las posibilidades de encontrar soluciones óptimas.

Si se retoma a la ecuación (2.8), se tiene que la entropía mide la diversidad en los valores de energía en función de la magnitud en la que estos aparecen dentro de una población, entonces si se registran incrementos en la entropía se puede entender que hay un incremento en la variedad de configuraciones con diferentes energías.

Ahora, es importante notar que la entropía es una medida de diversidad que pasa por alto las degeneraciones en las diferentes configuraciones con las mismas energías, pues no está diseñada para medirlas; sin embargo, esto puede detectarse

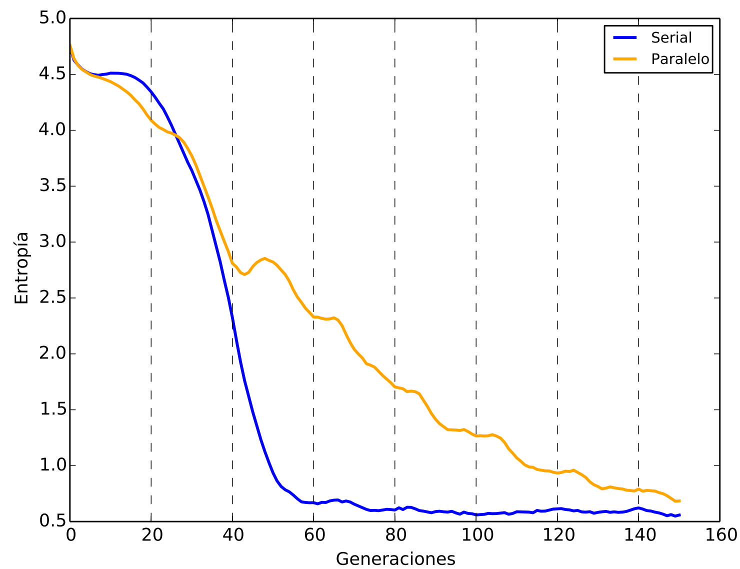


FIGURA 4.7: Entropía promedio de las poblaciones de los algoritmos genéticos serial y paralelo sobre las 100 pruebas realizadas. Las líneas punteadas indican las generaciones en las que hubo migración para el caso paralelo.

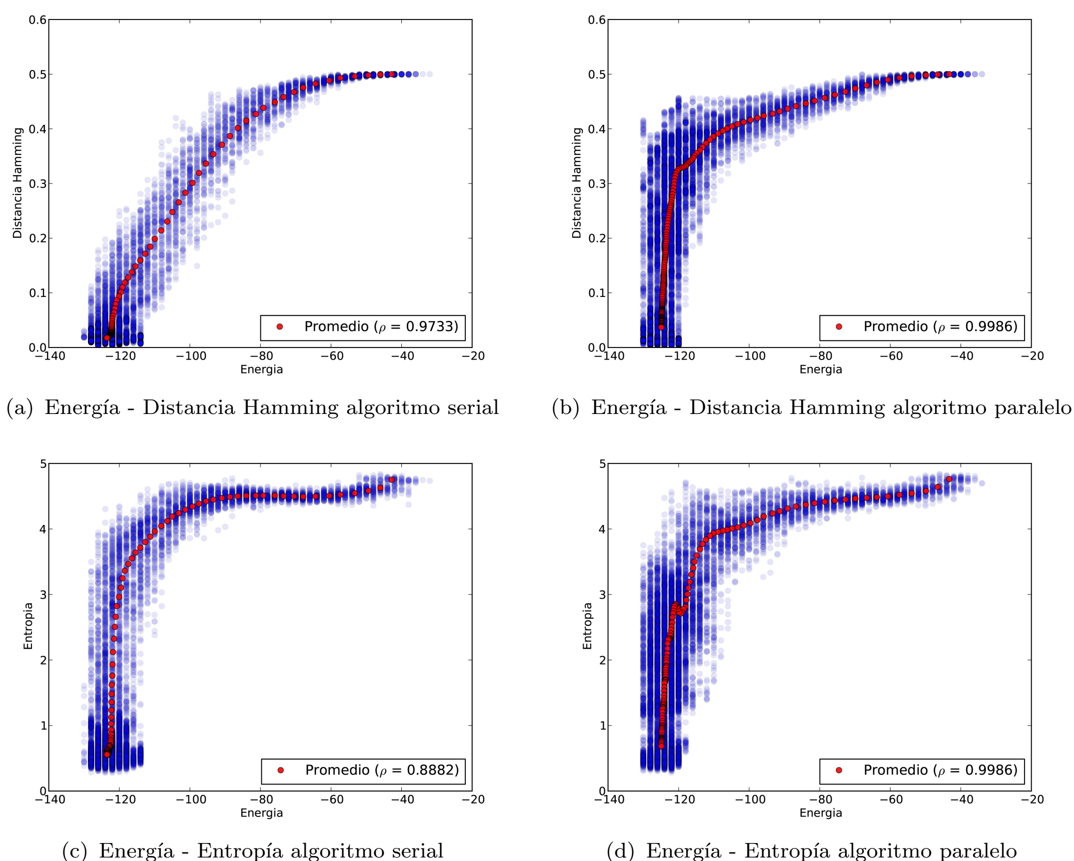
con la distancia Hamming, ya que está diseñada para medir las diferencias en las configuraciones en lugar de las diferencias en sus energías.

Las características en las que se enfocan la distancia Hamming y la entropía para medir la diversidad de la población explican el comportamiento que se observa en los casos paralelos de las figuras 4.6 y 4.7, respectivamente. El decaimiento en el caso de la distancia Hamming se debe a la presión que el algoritmo ejerce para converger a una solución, y lo que frena ese decaimiento son dos cosas: primero, el aislamiento que existe entre las subpoblaciones, lo que permite explorar diferentes regiones en el espacio de configuraciones, y la segunda es el ingreso de "nuevo" material genético a las subpoblaciones con la migración, el cual va perdiendo su efecto con el paso de las generaciones por la misma convergencia. Mientras que para la entropía los incrementos que se presentan después de las migraciones son un claro ejemplo de los efectos de la migración para tratar de mantener la diversidad dentro de una población, ya que con la inclusión de nuevo material genético en la población y el paso de las generaciones, las diferencias de material genético empiezan a crear nuevas configuraciones con diferentes energías, aunque no necesariamente mejores a las ya existentes, para después ser homogeneizadas de nuevo por la presión del algoritmo para converger.

También se realizaron pruebas estadísticas para tratar de encontrar alguna

dependencia entre la diversidad de la población en un algoritmo genético y la calidad de la solución encontrada. La hipótesis que se sugiere es que al mantener una diversidad relativamente alta de la población a lo largo de las generaciones en el algoritmo, se logra una encontrar una solución de mejor calidad (menor energía).

Para poder probar lo anterior se utilizó la correlación Spearman, pues como ya se mencionó en el capítulo 2, esta prueba mide qué tan relacionadas pueden estar dos variables por medio de alguna función monotónica, por lo que se espera encontrar qué tanto están relacionadas la optimización de la energía y la diversidad de la población para encontrar la mejor solución posible.



(a) Energía - Distancia Hamming algoritmo serial

(b) Energía - Distancia Hamming algoritmo paralelo

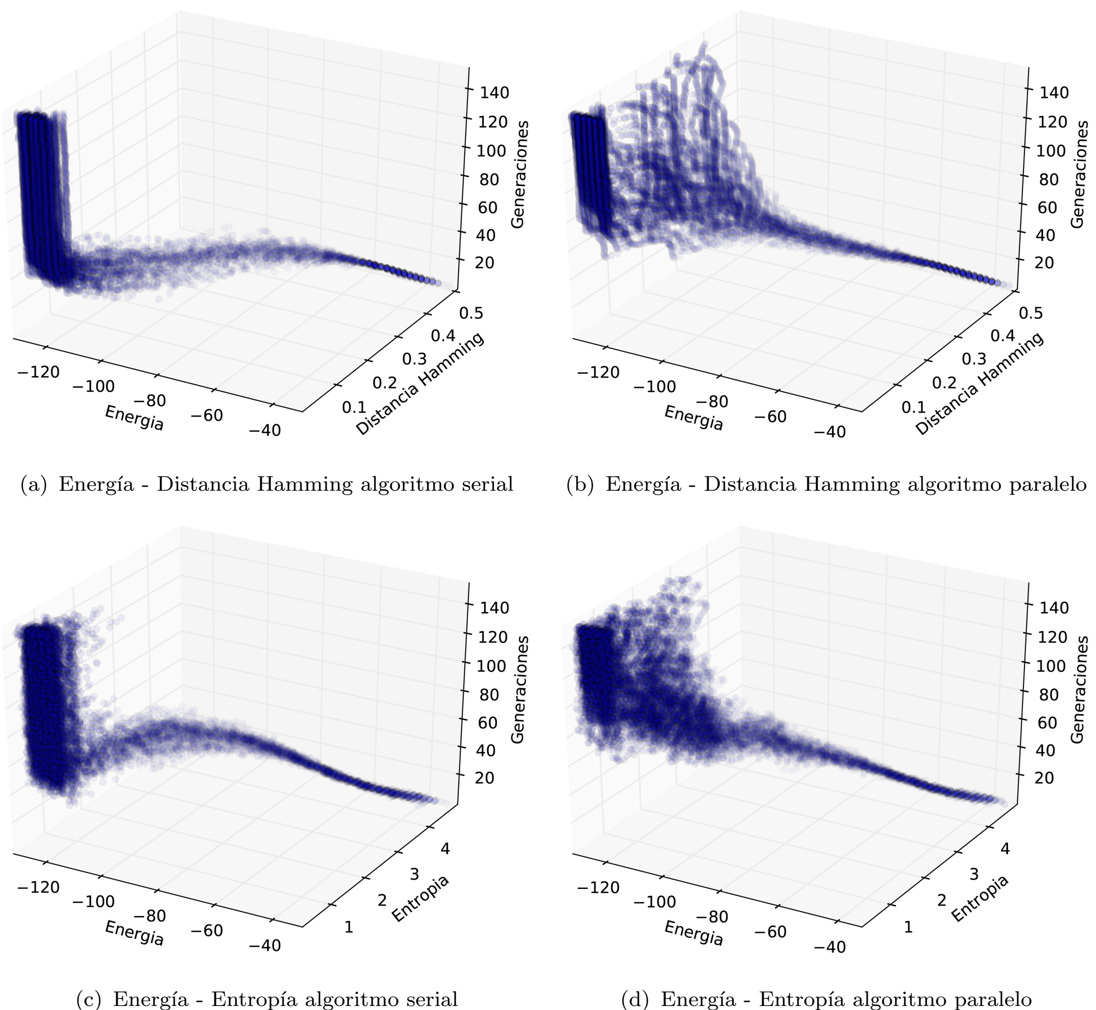
(c) Energía - Entropía algoritmo serial

(d) Energía - Entropía algoritmo paralelo

FIGURA 4.8: Dispersiones (Energía - Distancia Hamming y Energía - Entropía) de los algoritmos genéticos serial y paralelo de las 100 pruebas realizadas.

Por una parte se tiene que la optimización de la energía tiene un comportamiento decreciente para el caso del mejor individuo de la población, ya que su energía siempre es menor o igual debido al elitismo dentro del algoritmo. Y es este motivo el que sugiere utilizar la correlación Spearman, pues con la diversidad no necesariamente pasa lo mismo, especialmente en el caso del algoritmo paralelo, lo cual se podría ver reflejado en los coeficientes de correlación.

De modo que se consideraron como variables al valor de la energía del mejor individuo de la población, tanto en el caso serial como en el paralelo, y a las medidas de diversidad asociadas a la población en cada generación, así que de cada una de las 100 pruebas realizadas se obtuvo un dato de cada variable para cada una de las 150 generaciones, con lo que se obtuvieron los patrones de dispersión de la figura 4.8.



(a) Energía - Distancia Hamming algoritmo serial

(b) Energía - Distancia Hamming algoritmo paralelo

(c) Energía - Entropía algoritmo serial

(d) Energía - Entropía algoritmo paralelo

Figura 4.9: Dispersiones en el tiempo (Energía - Distancia Hamming y Energía - Entropía) de los algoritmos genéticos serial y paralelo de las 100 pruebas realizadas.

En cada uno de los casos se consideraron los promedios por generaciones para observar el comportamiento del algoritmo también de forma promedio y después se calculó el coeficiente de la correlación Spearman ρ de los promedios.

Los resultados de las correlaciones para cada una de las dispersiones anteriores muestran que la correlación Spearman no es suficientemente estricta como para

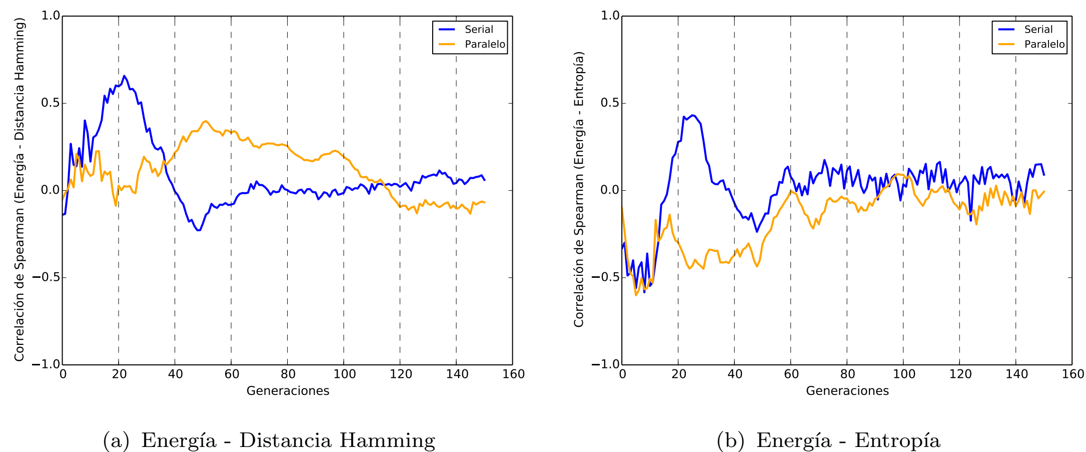
poder marcar las diferencias que se observan en los comportamientos promedio de cada dispersión, pues éstos presentan pequeñas diferencias y sin embargo en todos los casos se tiene un alto coeficiente de correlación, lo que ciertamente establece la relación entre la diversidad de la población y la energía de la mejor solución encontrada, pero al final esto no es de mucha ayuda para establecer un punto de comparación que muestre las diferencias que se observan entre el algoritmo serial y el paralelo.

Ahora, si se incluye el tiempo (generaciones) como otra variable en las dispersiones de la figura 4.8 se pueden identificar de mejor forma los mínimos locales (subóptimos) y el comportamiento de la población en los mismos, ya que además de observar los valores de energía en los que se presentan estos mínimos locales, se puede ver el tiempo que la población pasa en ellos.

En la figura 4.9 se presentan las dispersiones en función del tiempo, y en el caso particular de la dispersión de energía y distancia Hamming en el algoritmo paralelo (figura 4.9(b)) se puede observar que en muchas ocasiones las trayectorias se estancan en algún mínimo local, por lo que se tiene trayectoria vertical en el eje del tiempo (generaciones) y se observa un cambio en la trayectoria hacia una energía menor cuando se sale del mínimo local. Esto muestra que hay más posibilidades de escapar de configuraciones subóptimas en el algoritmo paralelo ya que en el caso serial (figura 4.9(a)) las trayectorias no realizan tantos cambios como en el caso paralelo.

También se puede obtener más información con la correlación Spearman si se hacen cortes sobre los planos que define el eje del tiempo, es decir, si se considera como el conjunto de datos el valor de la energía del individuo más apto de la población y el valor de diversidad de la población en cada generación para cada una de las 100 pruebas independientes. Utilizando como referencia la figura 4.9, se tendría que si se selecciona el plano correspondiente a la generación 1, todos los puntos que quedan sobre este plano tienen información que relaciona energía con diversidad y corresponden a cada una de las 100 pruebas independientes, por lo que pueden servir para marcar tendencias del algoritmo, ya que es otro modo de medir su comportamiento promedio diferente al utilizado anteriormente en la figura 4.8.

En las gráficas de la figura 4.10 se tienen las tendencias del algoritmo para la correlación Spearman entre la energía de la mejor solución y alguna de las medidas de diversidad en función del tiempo.



(a) Energía - Distancia Hamming

(b) Energía - Entropía

FIGURA 4.10: Correlación Spearman (Energía - Distancia Hamming y Energía - Entropía) en función del tiempo de los algoritmos genéticos serial y paralelo sobre las 100 pruebas realizadas. Las líneas punteadas indican las generaciones en las que hubo migración para el caso paralelos.

De la figura 4.10(a), correspondiente a la correlación Spearman entre energía y distancia Hamming, se puede observar una correlación positiva considerable dentro de las primeras 40 generaciones para el caso serial, lo que significa que la mayor parte de la optimización ocurre en esta etapa y después se tiene una correlación prácticamente nula debido a la convergencia. Ésta convergencia es más rápida comparada con el caso paralelo, pues aunque la correlación no es tan fuerte como en las primeras generaciones del caso serial, ésta se mantiene por un periodo más largo, lo que significa que la convergencia tarda un poco más.

En cuanto a la correlación Spearman para la energía de la mejor solución y la entropía (figura 4.10(b)), se observa que la mayor correlación alcanzada en el caso serial coincide en el tiempo con el máximo en la correlación de energía y distancia Hamming, mientras que en el caso paralelo no se presenta esta situación, pues se tiene que los incrementos en la entropía ocasionados por las migraciones interfieren de forma significativa con la correlación Spearman, ya que está diseñada para comportamientos monotónicos por lo que hace más difícil el tener una conclusión sobre el valor de los coeficientes encontrados a lo largo de las generaciones.

Finalmente, hay que hacer mención sobre la degeneración de estados, la cual se debe a la naturaleza del problema, pues para cada configuración se tiene su dual (todos los spines invertidos), a cuales se suman las plaquetas frustadas, que son otra fuente de degeneración de estados en los spin glasses. Y para ejemplificar la degeneración de estados, en la figura 4.11 se muestran algunas de las mejores

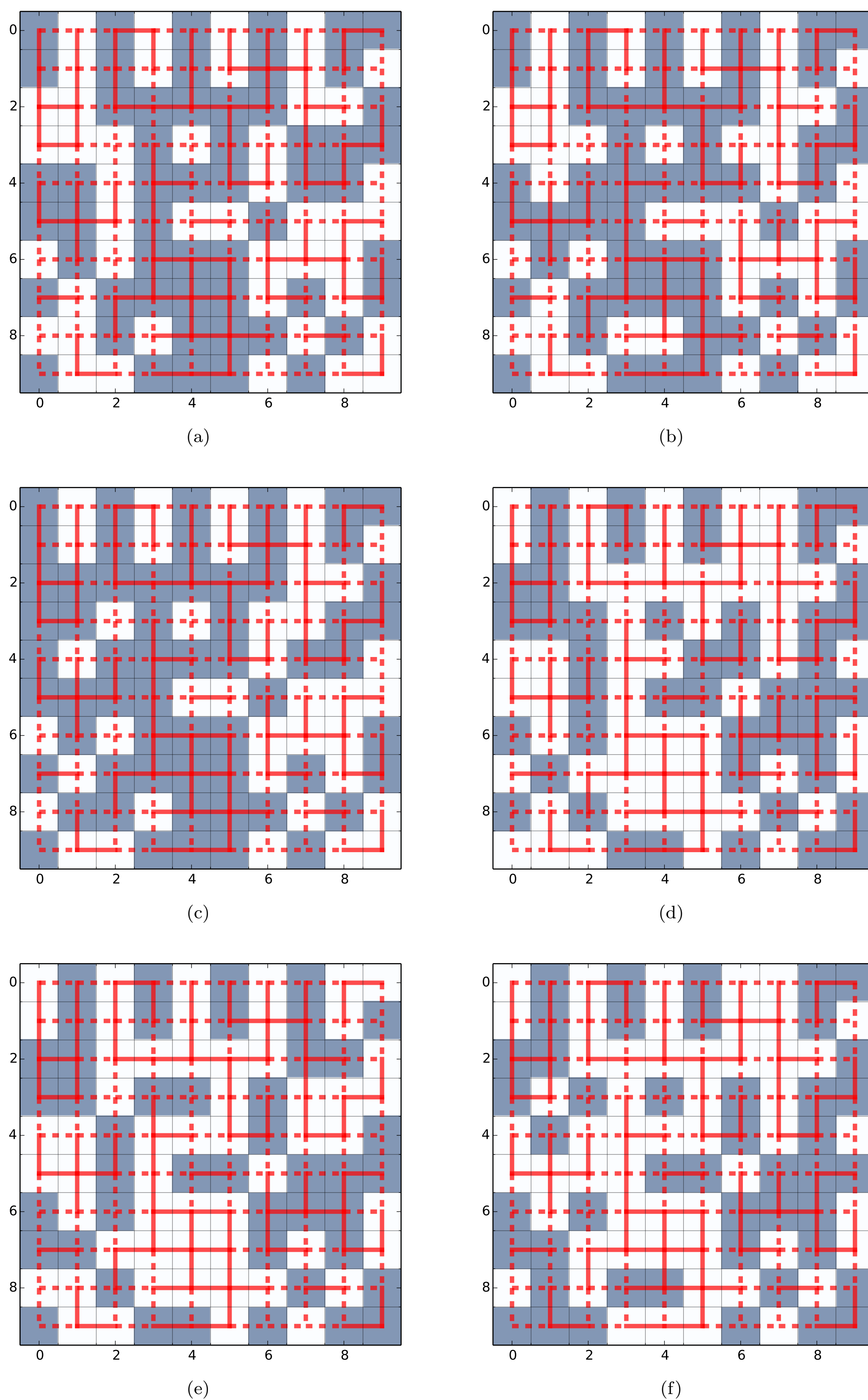


FIGURA 4.11: Configuraciones encontradas para los spin glasses. Todas las configuraciones aquí presentadas tienen una energía $E = -130$, que corresponde a la energía de las mejores soluciones encontradas en ambos algoritmos.

soluciones encontradas durante las 100 pruebas tanto del caso serial como el paralelo.

La energía correspondiente a las mejores soluciones encontradas es $E = -130$ y para tratar de tener una noción del grado de optimización que se logró con los algoritmos utilizados se puede hacer una estimación del número de interacciones que se pudieron satisfacer. La forma de realizar esto es considerando el tamaño del sistema, pues se tiene que para una matriz de lado $L = 10$ hay 180 interacciones, por lo que en un sistema sin plaquetas frustradas (modelo de Ising) la energía mínima es $E = -180$. De modo que si se hace el cociente de la energía de la mejor solución encontrada con la energía mínima de un sistema del mismo tamaño sin frustraciones, se tiene que el porcentaje de optimización es del 72.22%, pero no se tiene la certeza de que las configuraciones con energía $E = -130$ sean las soluciones óptimas para la configuración particular de interacciones utilizadas en este trabajo.

Capítulo 5

Conclusiones

5.1 Conclusión

En este trabajo, se estudió el problema de los spin glasses con el modelo de Edwards-Anderson como un problema de optimización combinatoria, el cual a pesar de ser un modelo simple presenta un alto grado de complejidad para poder encontrar las configuraciones de los estados base, debido a las degeneraciones en los estados energéticos originadas por plaquetas frustradas en el sistema.

Debido a esta complejidad del problema es necesario recurrir a diferentes técnicas para tratar de resolverlo, siendo los algoritmos genéticos una de estas opciones, y en particular para el objetivo de este trabajo, la versión paralela de este tipo de algoritmos fue implementada con el fin de buscar diferencias en cuanto al comportamiento de los algoritmos y la calidad de las soluciones encontradas en cada caso.

La diversidad en las poblaciones se convirtió en un parámetro que sirvió para poder determinar las diferencias de comportamiento entre la versión serial y paralela resolviendo el mismo problema.

Las diferencias encontradas entre los dos algoritmos (serial y paralelo), aparte de mostrar diferencias significativas en los tiempos de ejecución, también se reflejan en la calidad de las soluciones encontradas, pues aunque en ambos casos se encontraron soluciones con la misma energía, en el caso del algoritmo paralelo esto sucede con una mayor frecuencia, y de igual forma la distribución que muestra

la calidad de las soluciones encontradas favorece a esta implementación sobre la versión serial.

5.2 Trabajo futuro

En cuanto a la parte física del problema hay muchas modificaciones que se pueden hacer más allá del modelo de Edwards-Anderson, como puede ser el trabajar con interacciones que tengan una magnitud asignada por una distribución de probabilidad o una distribución Gaussiana, como el modelo de Sherrington-Kirkpatrick que trabaja con interacciones no-discretas.

Otra modificación que puede considerarse es la de trabajar sobre diferentes estructuras cristalinas, como puede ser una red hexagonal, ya que en este caso la paridad en el número de vecinos e interacciones cambia, lo que puede modificar un poco la dinámica de optimización de la energía.

Para el caso particular del código desarrollado para este trabajo, hay muchas modificaciones que podrían realizarse para mejorar el desempeño computacional del programa.

El programa utilizado al estar desarrollado en Python se puede mejorar en las partes correspondientes a los ciclos, ya que estos son las partes más lentas del programa por la forma en la que trabaja Python. Una forma de poder lograr una mejora significativa en el desempeño computacional podría ser modificando estos ciclos de modo que utilicen Numba. Numba es un módulo de Python que permite compilar los ciclos dentro de un código para que estos se ejecuten de forma mucho más eficiente y por lo tanto reducir de forma importante los tiempos de ejecución del algoritmo.

Apéndice A

Código fuente

A.1 Algoritmo genético serial

```

from numpy import array, transpose, savetxt, min, max, histogram, zeros, log, isnan, arange,
     loadtxt
from random import randint, random
from operator import add
from copy import deepcopy
from scipy import stats
import os
import time

# generar individuos
#########################################################
def individual(dim, prob):
    ind = array([[(random() < prob)*1 for i in xrange(dim)] for j in xrange(dim)])
    return ind
############################################################################

# generar poblacion
#########################################################
def population(pob, dim, prob):
    return [ individual(dim, prob) for x in xrange(pob) ]
############################################################################

# fitness
#########################################################
def fitness(ind, dim):
    ind = ind*2-1
    total = 0
    for i in xrange(dim):
        for j in xrange(dim-1):
            sig = j+1
            total -= m_inter_hor[i][j]*ind[i,j]*ind[i,sig]
```

```
            total -= m_inter_ver[j][i]*ind[j,i]*ind[sig,i]
    return total
##############################################################################

# dist hamming
##########################################################
def hamming(pop, ii):
    suma_tmp = 0
    suma_tmp = sum(sum(abs(pop[(ii+1):]-pop[ii])).reshape(dim*dim))
    return suma_tmp
##############################################################################

# operador mutacion
##########################################################
def mutation(child, dim, prob):
    if random() < 0.5:
        child[randint(0, dim-1)] = array([(random() < prob)*1 for i in xrange(dim)])
    else:
        child.T[randint(0, dim-1)] = array([(random() < prob)*1 for i in xrange(dim)])
    return child
##############################################################################

# operador crossover
##########################################################
def crossover(pop, dim, prob, desired_length, mutate = 0.1):
    children = []
    children_cross = []
    index = []
    children_mut = []
    index_mut = []
    while len(children) < desired_length:
        male1 = randint(0, len(pop)-1)
        par_tmp = randint(0, len(pop)-1)
        while par_tmp == male1:
            par_tmp = randint(0, len(pop)-1)
        male2 = par_tmp
        if male1 <= male2:
            male = pop[male1]
            index.append(male1)
        else:
            male = pop[male2]
            index.append(male2)
        par_tmp = randint(0, len(pop)-1)
        while par_tmp == male1 or par_tmp == male2:
            par_tmp = randint(0, len(pop)-1)
        female1 = par_tmp
        par_tmp = randint(0, len(pop)-1)
        while par_tmp == male1 or par_tmp == male2 or par_tmp == female1:
            par_tmp = randint(0, len(pop)-1)
        female2 = par_tmp
        if female1 <= female2:
            female = pop[female1]
            index.append(female1)
        else:
            female = pop[female2]
```

```
            index.append(female2)
        child = [deepcopy(male), deepcopy(female)]
        corte = randint(0, dim-1)
        if random() < 0.5:
            child[0][corte] = female[corte]
            child[1][corte] = male[corte]
        else:
            child[0].T[corte] = female.T[corte]
            child[1].T[corte] = male.T[corte]
        children_cross.extend(deepcopy(child))
        if mutate > random():
            index_mut.append(len(index)-2)
            child[0] = mutation(child[0], dim, prob)
            children_mut.append(child[0])
        children.append(child[0])
        if desired_length-len(children) > 0:
            if mutate > random():
                index_mut.append(len(index)-1)
                child[1] = mutation(child[1], dim, prob)
                children_mut.append(child[1])
            children.append(child[1])
        children_cross = children_cross[:desired_length]
        index = index[:desired_length]
    return children, children_cross, index, children_mut, index_mut
#############################################################################

# main
#########################################################

# valores iniciales
generations = 5
pob = 100
dim = 10
prob = 0.5
rep = 2

retain = 0.05
mutate = 0.05
calcular_hamming = True

# directorios de resultados
directory = './s-dim-%s-pob-%s-rep-%s-%s' %(dim, pob, rep, time.strftime("%d_%m_%Y-%H_%M_%S"))
if not os.path.exists(directory):
    os.makedirs(directory)
    os.makedirs(directory+'/conf')

# matrices de interacciones
m_inter_hor = array([[(random() < prob) for i in xrange(dim)] for j in xrange(dim)])*2-1
m_inter_ver = array([[(random() < prob) for i in xrange(dim)] for j in xrange(dim)])*2-1
m_inter_hor = loadtxt('m_inter_hor.txt', dtype='int')
m_inter_ver = loadtxt('m_inter_ver.txt', dtype='int')
savetxt(directory+'/m_inter_hor.txt', m_inter_hor, fmt='%3d')
savetxt(directory+'/m_inter_ver.txt', m_inter_ver, fmt='%3d')

###########################
```

```
142
143 # stat
144 if rep > 1:
145     hist_mejor = []
146     disp_fit = []
147     disp_ent = []
148     if calcular_hamming == True:
149         disp_dist = []
150     disp_ef_cross = []
151     disp_ef_mut = []
152
153 for kk in xrange(rep):
154     dist_prom = zeros(generations+1)
155     ent = zeros(generations+1)
156     ef_cross = []
157     ef_mut = []
158
159 ###########################
160
161     pop = population(pob, dim, prob)
162
163     # calcular fitness
164     ################################
165     graded = []
166     prom_tmp = 0
167     for i in xrange(pob):
168         graded.append([fitness(pop[i], dim), i])
169         prom_tmp += graded[i][0]
170     ################################
171
172     cross = array(graded).T[0]
173     fitness_history = [prom_tmp*1.0/pob]
174     fitness_mejor = [transpose(array(sorted(graded)))[0][0]]
175
176     pop_aux = deepcopy(pop)
177     pop = [pop_aux[x] for x in transpose(array(sorted(graded)))[1]]
178     pop_in = deepcopy(pop)
179     pop_mejor = [pop[0]]
180
181     # dist hamming
182     ################################
183     if calcular_hamming == True:
184         tmp = 0
185         for ii in xrange(pob-1):
186             tmp += hamming(pop, ii)
187         dist_prom[0] = tmp*2.0/(dim*dim*pob*(pob-1))
188     ################################
189
190     # entropia
191     #########################
192     bins = max(array(graded).T[0])-min(array(graded).T[0])
193     if bins <= 0:
194         bins = 1
195     ent_tmp = histogram(array(graded).T[0], bins, normed = 1)[0]
196     for ee in ent_tmp:
```

```
197         if ee != 0:
198             ent[0] += -(ee*log(ee)/log(2))
199     #########################
200
201     i = 0
202     while i < generations:
203
204         # evolucion
205         ##################################################
206         retain_length = int(len(pop)*retain)
207         if (retain_length < 1):
208             retain_length = 1
209         parents = deepcopy(pop[:retain_length])
210         desired_length = len(pop) - len(parents)
211         evol = crossover(pop, dim, prob, desired_length, mutate)
212         parents.extend(evol[0])
213         pop = deepcopy(parents)
214         ##################################################
215
216         graded = []
217         cross_aux = []
218         mut_aux = []
219         prom_tmp = 0
220         for j in xrange(pob):
221             graded.append([fitness(pop[j], dim), j])
222             prom_tmp += graded[j][0]
223             if j < len(evol[1]):
224                 cross_aux.append(fitness(evol[1][j], dim))
225                 if j < len(evol[4]):
226                     mut_aux.append(fitness(evol[3][j], dim))
227         cross_aux = array(cross_aux)
228
229         # efecto operador crossover
230         ############
231         ef_cross.append(sum(cross_aux-cross[evol[2]])*1.0/len(cross_aux))
232         ############
233
234         # efecto operador mutacion
235         ############
236         if len(mut_aux) == 0:
237             mut_len = 1
238         else:
239             mut_len = len(mut_aux)
240         ef_mut.append(sum(mut_aux-cross_aux[evol[4]])*1.0/mut_len)
241         ############
242
243         cross = array(graded).T[0]
244         fitness_history.append(prom_tmp*1.0/pob)
245         fitness_mejor.append(transpose(array(sorted(graded)))[0][0])
246         pop_aux = deepcopy(pop)
247         pop = [pop_aux[x] for x in transpose(array(sorted(graded)))[1]]
248         pop_mejor.append(pop[0])
249
250         # dist hamming
251         ################################
```

```
252         if calcular_hamming == True:
253             tmp = 0
254             for ii in xrange(pob-1):
255                 tmp += hamming(pop, ii)
256             dist_prom[i+1] = tmp*2.0/(dim*dim*pob*(pob-1))
257         ################################
258 
259         # entropia
260         #########################
261         bins = max(array(graded).T[0])-min(array(graded).T[0])
262         if bins <= 0:
263             bins = 1
264         ent_tmp = histogram(array(graded).T[0], bins, normed = 1)[0]
265         for ee in ent_tmp:
266             if ee != 0:
267                 ent[i+1] += -(ee*log(ee)/log(2))
268         #########################
269 
270         print i, fitness_history[i], fitness_history[i+1]
271         i+=1
272 
273     savetxt(directory+'/conf/conf%s-fit'%(kk)+str(fitness_mejor[-1])+'.txt', pop_mejor[-1],
         fmt='%3d')
274 
275     if rep > 1:
276         hist_mejor.append(fitness(pop[0], dim))
277         disp_fit.append(fitness_mejor)
278         disp_ent.append(ent[:i+2])
279         if calcular_hamming == True:
280             disp_dist.append(dist_prom[:i+2])
281         disp_ef_cross.append(ef_cross)
282         disp_ef_mut.append(ef_mut)
283 
284 if rep > 1:
285 
286     prom_fit = zeros(generations+1)
287     prom_ent = zeros(generations+1)
288     if calcular_hamming == True:
289         prom_dist = zeros(generations+1)
290     prom_ef_cross = zeros(generations)
291     prom_ef_mut = zeros(generations)
292     for ii in xrange(rep):
293         prom_fit += array(disp_fit[ii])
294         prom_ent += array(disp_ent[ii])
295         if calcular_hamming == True:
296             prom_dist += array(disp_dist[ii])
297         prom_ef_cross += array(disp_ef_cross[ii])
298         prom_ef_mut += array(disp_ef_mut[ii])
299     prom_fit /= rep
300     prom_ent /= rep
301     if calcular_hamming == True:
302         prom_dist /= rep
303     prom_ef_cross /= rep
304     prom_ef_mut /= rep
305     spear_fit_ent = []
```

```
306     spear_fit_dist = []
307     for ii in xrange(generations+1):
308         spear_fit_ent.append(stats.spearmanr(array(disp_fit).T[ii], array(disp_ent).T[ii])[0])
309         if calcular_hamming == True:
310             spear_fit_dist.append(stats.spearmanr(array(disp_fit).T[ii],
         array(disp_dist).T[ii])[0])
311
312     savetxt(directory+'/hist_mejor.txt', hist_mejor)
313     savetxt(directory+'/prom_fit.txt', prom_fit)
314     savetxt(directory+'/prom_ent.txt', prom_ent)
315     savetxt(directory+'/prom_ef_mut.txt', prom_ef_mut)
316     savetxt(directory+'/prom_ef_cross.txt', prom_ef_cross)
317     savetxt(directory+'/spearman_fit_ent.txt', spear_fit_ent)
318     if calcular_hamming == True:
319         savetxt(directory+'/prom_dist.txt', prom_dist)
320         savetxt(directory+'/spearman_fit_dist.txt', spear_fit_dist)
```

A.2 Algoritmo genético paralelo

```
1
2  import multiprocessing as mp
3  from numpy import array, transpose, savetxt, min, max, histogram, zeros, log, isnan, arange,
       loadtxt
4  from random import randint, random
5  from operator import add
6  from copy import deepcopy
7  from scipy import stats
8  import os
9  import time
10
11 # generar individuos
12 ##########################################################
13 def individual(dim, prob):
14     ind = array([[(random() < prob)*1 for i in xrange(dim)] for j in xrange(dim)])
15     return ind
16
17 # generar poblacion
18 ##########################################################
19 def population(pob, dim, prob):
20     return [ individual(dim, prob) for x in xrange(pob) ]
21
22 # fitness
23 ##########################################################
24 def fitness(ind, dim, inx=[]):
25     ind = ind*2-1
26     total = 0
27     for i in xrange(dim):
28         for j in xrange(dim-1):
29             sig = j+1
30             total -= m_inter_hor[i][j]*ind[i,j]*ind[i,sig]
31             total -= m_inter_ver[j][i]*ind[j,i]*ind[sig,i]
32     if (len(inx)>0):
33         return total, inx[0]
34     else:
35         return total
36
37 # fitness poblacion
38 def fit_pob(pop, ii, dim, pop_evol = [], pop_mut = []):
39     fit_tmp = []
40     cross_tmp = []
41     mut_tmp = []
42     for i in xrange(ii, len(pop), procesadores):
43         fit_tmp.append(fitness(pop[i], dim, [i]))
44         if len(pop_evol) > 0 and i < len(pop_evol):
45             cross_tmp.append(fitness(pop_evol[i], dim, [i]))
46             if len(pop_mut) > 0 and i < len(pop_mut):
47                 mut_tmp.append(fitness(pop_mut[i], dim, [i]))
48     return fit_tmp, cross_tmp, mut_tmp
49
50 # dist hamming
51 ##########################################################
```

```
52 def hamming(pop, ii):
53     suma_tmp = 0
54     for i in xrange(ii, len(pop)-1, procesadores):
55         suma_tmp += sum(sum(abs(pop[(i+1):]-pop[i])).reshape(dim*dim))
56     return suma_tmp
57 
58 # operador mutacion
59 ############################################################
60 def mutation(child, dim, prob):
61     if random() < 0.5:
62         child[randint(0, dim-1)] = array([(random() < prob)*1 for i in xrange(dim)])
63     else:
64         child.T[randint(0, dim-1)] = array([(random() < prob)*1 for i in xrange(dim)])
65     return child
66 
67 # operador crossover
68 ############################################################
69 def crossover(pop, dim, prob, desired_length, mutate = 0.1):
70     children = []
71     children_comp = []
72     index = []
73     children_mut = []
74     index_mut = []
75     while len(children) < desired_length:
76         male1 = randint(0, len(pop)-1)
77         par_tmp = randint(0, len(pop)-1)
78         while par_tmp == male1:
79             par_tmp = randint(0, len(pop)-1)
80         male2 = par_tmp
81         if male1 <= male2:
82             male = pop[male1]
83             index.append(male1)
84         else:
85             male = pop[male2]
86             index.append(male2)
87         par_tmp = randint(0, len(pop)-1)
88         while par_tmp == male1 and par_tmp == male2:
89             par_tmp = randint(0, len(pop)-1)
90         female1 = par_tmp
91         par_tmp = randint(0, len(pop)-1)
92         while par_tmp == male1 and par_tmp == male2 and par_tmp == female1:
93             par_tmp = randint(0, len(pop)-1)
94         female2 = par_tmp
95         if female1 <= female2:
96             female = pop[female1]
97             index.append(female1)
98         else:
99             female = pop[female2]
100            index.append(female2)
101        child = [deepcopy(male), deepcopy(female)]
102        corte = randint(0, dim-1)
103        if random() < 0.5:
104            child[0][corte] = female[corte]
105            child[1][corte] = male[corte]
106        else:
```

```
107             child[0].T[corte] = female.T[corte]
108             child[1].T[corte] = male.T[corte]
109         children_comp.extend(deepcopy(child))
110         if mutate > random():
111             index_mut.append(len(index)-2)
112             child[0] = mutation(child[0], dim, prob)
113             children_mut.append(child[0])
114         children.append(child[0])
115         if desired_length-len(children) > 0:
116             if mutate > random():
117                 index_mut.append(len(index)-1)
118                 child[1] = mutation(child[1], dim, prob)
119                 children_mut.append(child[1])
120             children.append(child[1])
121         children_comp = children_comp[:desired_length]
122         index = index[:desired_length]
123     return children, children_comp, index, children_mut, index_mut
124 
125 #funcion auxiliar
126 #########################################################
127 def log_result(result):
128     result_list.append(result)
129 
130 # main
131 #########################################################
132 
133 # parametros
134 generations = 5
135 pob = 100
136 dim = 10
137 prob = 0.5
138 rep = 2
139 
140 retain = 0.05
141 mutate = 0.05
142 
143 sub_pob = 4
144 int_pob = 4
145 procesadores = mp.cpu_count()
146 calcular_hamming = True
147 
148 # directorios de resultados
149 directory = './p-dim-%s-pob-%s-rep-%s-nuc-%s-%s' %(dim, pob, rep, procesadores,
     time.strftime("%d_%m_%Y-%H_%M_%S"))
150 if not os.path.exists(directory):
151     os.makedirs(directory)
152     os.makedirs(directory+'/conf')
153 
154 # matrices de interacciones
155 m_inter_hor = array([[(random() < 0) for i in xrange(dim)] for j in xrange(dim)])*2-1
156 m_inter_ver = array([[(random() < 0) for i in xrange(dim)] for j in xrange(dim)])*2-1
157 m_inter_hor = loadtxt('m_inter_hor.txt', dtype='int')
158 m_inter_ver = loadtxt('m_inter_ver.txt', dtype='int')
159 savetxt(directory+'/m_inter_hor.txt', m_inter_hor, fmt='%3d')
160 savetxt(directory+'/m_inter_ver.txt', m_inter_ver, fmt='%3d')
```

```
161
162 ###########################
163
164 if rep > 1:
165     hist_mejor = []
166     disp_fit = []
167     disp_ent = []
168     if calcular_hamming == True:
169         disp_dist = []
170     disp_ef_cross = []
171     disp_ef_mut = []
172
173 # marcas aux graficas
174 x_aux = []
175 for kk in xrange(generations):
176     if kk%int_pob == 0 and kk != 0:
177         x_aux.append(kk)
178
179 for kk in xrange(rep):
180     hist_aux = []
181     dist_real = zeros(generations+1)
182     ent = zeros(generations+1)
183     ef_cross = zeros(generations)
184     ef_mut = zeros(generations)
185     hist_plot = []
186
187     # variables paralelo
188     p_pop = []
189     p_fit_mejor = [ [] for zz in xrange(sub_pob)]
190     p_hist_aux = [ [] for zz in xrange(sub_pob)]
191     p_ent = [ zeros(generations+1) for zz in xrange(sub_pob)]
192     p_cross = [ [] for zz in xrange(sub_pob)]
193     p_cross_tmp = [ [] for zz in xrange(sub_pob)]
194     p_ef_cross = [ zeros(generations) for zz in xrange(sub_pob)]
195     p_ef_mut = [ zeros(generations) for zz in xrange(sub_pob)]
196
197     k_sub_pob = 1
198     para_pob = pob/sub_pob
199     mod_pob = pob%sub_pob
200     while mod_pob > 0:
201         print k_sub_pob, para_pob+1
202         k_sub_pob += 1
203         mod_pob -= 1
204         p_pop.append(population(para_pob+1, dim, prob))
205     while k_sub_pob <= sub_pob:
206         print k_sub_pob, para_pob
207         k_sub_pob += 1
208         p_pop.append(population(para_pob, dim, prob))
209
210     p_pop_in = deepcopy(p_pop)
211
212     # calcular fitness
213     ###################################################
214     summed = 0
215     result_list = []
```

```
216     pop_list = []
217     graded = []
218     real_aux = []
219     for zz in xrange(sub_pob):
220         print 'zz', zz, 'len', len(p_pop[zz])
221         real_aux.extend(p_pop[zz])
222         pool = mp.Pool(procesadores)
223         for ii in xrange(procesadores):
224             pool.apply_async(fit_pob, args = (p_pop[zz], ii, dim), callback = log_result)
225         pool.close()
226         pool.join()
227         result_list_tmp = []
228 
229         for ii in xrange(procesadores):
230             result_list_tmp.extend(result_list[ii][0])
231         result_list = result_list_tmp
232         graded.extend(result_list)
233         p_cross[zz] = deepcopy(result_list)
234         p_cross[zz] = array(sorted(p_cross[zz], key = lambda campo: campo[1])).T[0]
235         pop_list.extend([p_pop[zz][x[1]] for x in sorted(result_list)])
236         summed += reduce(add, (x[0] for x in result_list))*1.0/len(p_pop[zz])
237 
238         p_fit_mejor[zz] = [sorted(result_list)[0][0]]
239         p_hist_aux[zz] = array(result_list).T[0]
240         hist_aux.extend(list(array(result_list).T[0]))
241         p_pop[zz] = pop_list
242         pop_list = []
243         result_list = []
244 
245     # entropia
246     #########################
247     bins = max(hist_aux)-min(hist_aux)
248     if bins <= 0:
249         bins = 1
250     ent_tmp = histogram(hist_aux, bins, normed = 1)[0]
251     for ee in ent_tmp:
252         if ee != 0:
253             ent[0] += -(ee*log(ee)/log(2))
254     #########################
255 
256     # dist hamming
257     ########################
258     if calcular_hamming == True:
259         pool = mp.Pool(procesadores)
260         for ii in xrange(procesadores):
261             pool.apply_async(hamming, args = (real_aux, ii), callback = log_result)
262         pool.close()
263         pool.join()
264         dist_real[0] = sum(result_list)*2.0/(dim*dim*pob*(pob-1))
265         result_list = []
266     ########################
267 
268     hist_aux = []
269     fitness_history = [summed/sub_pob]
270     aux = fitness_history[0]
```

```
271     fitness_mejor = [sorted(transpose(p_fit_mejor)[len(transpose(p_fit_mejor))-1])[0]]
272     pop_mejor =
         [p_pop[list(transpose(p_fit_mejor)[len(transpose(p_fit_mejor))-1]).index(fitness_mejor[0])][0]]
273     p_pop_in = deepcopy(p_pop)
274 
275     i = 0
276     while i < generations:
277 
278         # evolucion
279         #################################################
280         p_evol = [ [] for zz in xrange(sub_pob)]
281         for zz in xrange(len(p_pop)):
282             retain_length = int(len(p_pop[zz])*retain)
283             if (retain_length < 1):
284                 retain_length = 1
285             parents = deepcopy(p_pop[zz][:retain_length])
286             desired_length = len(p_pop[zz]) - len(parents)
287             p_evol[zz] = crossover(p_pop[zz], dim, prob, desired_length, mutate)
288             parents.extend(p_evol[zz][0])
289             p_pop[zz] = deepcopy(parents)
290         p_cross_aux = [ [] for zz in xrange(sub_pob)]
291         p_mut_aux = [ [] for zz in xrange(sub_pob)]
292         #################################################
293 
294         # calcular fitness
295         ###################################################
296         p_fit_ind = [ [] for zz in xrange(sub_pob)]
297         summed = 0
298         graded = []
299         real_aux = []
300         for zz in xrange(sub_pob):
301             real_aux.extend(p_pop[zz])
302             pool = mp.Pool(procesadores)
303             for ii in xrange(procesadores):
304                 pool.apply_async(fit_pob, args = (p_pop[zz], ii, dim, p_evol[zz][1],
         p_evol[zz][3]), callback = log_result)
305             pool.close()
306             pool.join()
307             result_list_cross_tmp = []
308             for ii in xrange(procesadores):
309                 result_list_cross_tmp.extend(result_list[ii][1])
310             result_list_mut_tmp = []
311             for ii in xrange(procesadores):
312                 result_list_mut_tmp.extend(result_list[ii][2])
313             result_list_tmp = []
314             for ii in xrange(procesadores):
315                 result_list_tmp.extend(result_list[ii][0])
316             result_list = result_list_tmp
317 
318             graded.extend(result_list)
319             p_cross_aux[zz] = deepcopy(result_list_cross_tmp)
320             p_cross_aux[zz] = array(sorted(p_cross_aux[zz], key = lambda campo: campo[1])).T[0]
321             p_ef_cross[zz][i] =
         sum(p_cross_aux[zz]-p_cross[zz][p_evol[zz][2]])*1.0/len(p_cross_aux[zz])
322             ef_cross[i] += p_ef_cross[zz][i]
```

```
323             if len(result_list_mut_tmp) > 0:
324                 p_mut_aux[zz] = deepcopy(result_list_mut_tmp)
325                 p_mut_aux[zz] = array(sorted(p_mut_aux[zz], key = lambda campo: campo[1])).T[0]
326                 p_ef_mut[zz][i] =
    sum(p_mut_aux[zz]-p_cross_aux[zz][p_evol[zz][4]])*1.0/len(p_mut_aux[zz])
327             else:
328                 p_ef_mut[zz][i] = 0.0
329             ef_mut[i] += p_ef_mut[zz][i]
330             p_cross[zz] = deepcopy(result_list)
331             p_cross[zz] = array(sorted(p_cross[zz], key = lambda campo: campo[1])).T[0]
332 
333             pop_list.extend([p_pop[zz][x[1]] for x in sorted(result_list)])
334             summed += reduce(add, (x[0] for x in result_list))*1.0/len(p_pop[zz])
335             p_hist_aux[zz] = array(result_list).T[0]
336             hist_aux.extend(list(array(result_list).T[0]))
337             p_fit_mejor[zz].append(sorted(result_list)[0][0])
338             p_fit_ind[zz] = list(sorted(transpose(result_list)[0]))
339             p_pop[zz] = pop_list
340             pop_list = []
341             result_list = []
342 
343         ef_cross[i] /= sub_pob
344         ef_mut[i] /= sub_pob
345 
346         # entropia global
347         #########################
348         bins = max(hist_aux)-min(hist_aux)
349         if bins <= 0:
350             bins = 1
351         ent_tmp = histogram(hist_aux, bins, normed = 1)[0]
352         for ee in ent_tmp:
353             if ee != 0:
354                 ent[i+1] += -(ee*log(ee)/log(2))
355         #########################
356 
357         # dist hamming
358         ########################
359         if calcular_hamming == True:
360             pool = mp.Pool(procesadores)
361             for ii in xrange(procesadores):
362                 pool.apply_async(hamming, args = (real_aux, ii), callback = log_result)
363             pool.close()
364             pool.join()
365             dist_real[i+1] = sum(result_list)*2.0/(dim*dim*pob*(pob-1))
366             result_list = []
367         ########################
368         hist_aux = []
369 
370         fitness_history.append(summed/sub_pob)
371         fitness_mejor.append(sorted(transpose(p_fit_mejor)[len(transpose(p_fit_mejor))-1])[0])
372 
    pop_mejor.append(p_pop[list(transpose(p_fit_mejor)[len(transpose(p_fit_mejor))-1]).index(fitness_mejor[i+1])][
373 
374         # migracion entre poblaciones
375         ########################
```

```
376         if (i+1)%int_pob == 0:
377             int_tmp = []
378             for zz in xrange(len(p_pop)):
379                 int_tmp.append(p_pop[zz][0])
380             for zz in xrange(len(p_pop)):
381                 print zz, '<-', (zz+1)%sub_pob, p_fit_ind[zz][0], '<-',
      p_fit_ind[(zz+1)%sub_pob][0]
382                 index_der = sum((p_fit_ind[(zz+1)%sub_pob][0] > p_fit_ind[zz])*1)
383                 if index_der == len(p_pop[zz]):
384                     index_der -= 1
385                 print (zz-1)%sub_pob, '->', zz, p_fit_ind[(zz-1)%sub_pob][0], '->',
      p_fit_ind[zz][0]
386                 index_izq = sum((p_fit_ind[(zz-1)%sub_pob][0] > p_fit_ind[zz])*1)
387                 if index_izq == len(p_pop[zz]):
388                     index_izq -= 1
389                 if index_izq == index_der:
390                     p_pop[zz].pop()
391                     p_pop[zz].pop()
392                     if p_fit_ind[(zz+1)%sub_pob][0] <= p_fit_ind[(zz-1)%sub_pob][0]:
393                         index_izq += 1
394                         p_pop[zz].insert(index_der, int_tmp[(zz+1)%sub_pob])
395                         p_pop[zz].insert(index_izq, int_tmp[(zz-1)%sub_pob])
396                     else:
397                         index_der += 1
398                         p_pop[zz].insert(index_izq, int_tmp[(zz-1)%sub_pob])
399                         p_pop[zz].insert(index_der, int_tmp[(zz+1)%sub_pob])
400                 elif index_izq < index_der:
401                     p_pop[zz].insert(index_izq, int_tmp[(zz-1)%sub_pob])
402                     p_pop[zz].pop()
403                     p_pop[zz].insert(index_der, int_tmp[(zz+1)%sub_pob])
404                     p_pop[zz].pop()
405                 else:
406                     p_pop[zz].insert(index_der, int_tmp[(zz+1)%sub_pob])
407                     p_pop[zz].pop()
408                     p_pop[zz].insert(index_izq, int_tmp[(zz-1)%sub_pob])
409                     p_pop[zz].pop()
410         ########################
411 
412         print i, fitness_history[i], fitness_history[i+1]
413         i+=1
414 
415     savetxt(directory+'/conf/conf%s-fit'%(kk)+str(fitness_mejor[-1])+'.txt', pop_mejor[-1],
      fmt='%3d')
416 
417     if rep > 1:
418         hist_mejor.append(fitness_mejor[i])
419         disp_fit.append(fitness_mejor)
420         disp_ent.append(ent[:i+2])
421         if calcular_hamming == True:
422             disp_dist.append(dist_real[:i+2])
423         disp_ef_cross.append(ef_cross)
424         disp_ef_mut.append(ef_mut)
425 
426 if rep > 1:
427     prom_fit = zeros(generations+1)
```

```
428     prom_ent = zeros(generations+1)
429     if calcular_hamming == True:
430         prom_dist = zeros(generations+1)
431     prom_ef_cross = zeros(generations)
432     prom_ef_mut = zeros(generations)
433     for ii in xrange(rep):
434         prom_fit += array(disp_fit[ii])
435         prom_ent += array(disp_ent[ii])
436         if calcular_hamming == True:
437             prom_dist += array(disp_dist[ii])
438         prom_ef_cross += array(disp_ef_cross[ii])
439         prom_ef_mut += array(disp_ef_mut[ii])
440     prom_fit /= rep
441     prom_ent /= rep
442     if calcular_hamming == True:
443         prom_dist /= rep
444     prom_ef_cross /= rep
445     prom_ef_mut /= rep
446     spear_fit_ent = []
447     spear_fit_dist = []
448     for ii in xrange(generations+1):
449         spear_fit_ent.append(stats.spearmanr(array(disp_fit).T[ii], array(disp_ent).T[ii])[0])
450         if calcular_hamming == True:
451             spear_fit_dist.append(stats.spearmanr(array(disp_fit).T[ii],
         array(disp_dist).T[ii])[0])
452
453     savetxt(directory+'/hist_mejor.txt', hist_mejor)
454     savetxt(directory+'/prom_fit.txt', prom_fit)
455     savetxt(directory+'/prom_ent.txt', prom_ent)
456     savetxt(directory+'/prom_ef_mut.txt', prom_ef_mut)
457     savetxt(directory+'/prom_ef_cross.txt', prom_ef_cross)
458     savetxt(directory+'/spearman_fit_ent.txt', spear_fit_ent)
459     if calcular_hamming == True:
460         savetxt(directory+'/prom_dist.txt', prom_dist)
461         savetxt(directory+'/spearman_fit_dist.txt', spear_fit_dist)
```

Bibliografía

[1] Erick Cantú-Paz. *Efficient and Accurate Parallel Genetic Algorithms*, volume 1. Springer, 2000.

[2] Marc Mezard and Andrea Montanari. *Information, Physics and Computation.* Oxford University Press, 2009.

[3] Konrad H. Fischer and John A. Hertz. *Spin Glasses*, volume 1. Cambridge University Press, 1993.

[4] Kurt Binder and A. Peter Young. Spin glasses: Experimental facts, theoretical concepts, and open questions. *Reviews of Modern Physics*, 58(4):801, 1986.

[5] Hidetoshi Nishimori. *Statistical Physics of Spin Glasses and information processing: an introduction.* Oxford University Press, 2001.

[6] Helmut G. Katzgraber, Mathias Körner, and A.P. Young. Universality in three-dimensional Ising Spin Glasses: A Monte Carlo study. *Physical Review B*, 73(22):224432, 2006.

[7] Peter Alexander Foster. Parallel Combinatorial Optimisation for Finding Ground States of Ising Spin Glasses. Master's thesis, The University of Edinburgh, 2008.

[8] Martin Pelikan and David E Goldberg. Hierarchical boa solves ising spin glasses and maxsat. In *Genetic and Evolutionary Computation—GECCO 2003*, pages 1271–1282. Springer, 2003.

[9] John L Hennessy and David A Patterson. *Computer Architecture: A Quantitative Approach.* Elsevier, 2012.

[10] Ian Foster. *Designing and Building Parallel Programs.* Addison Wesley Publishing Company, 1995.

[11] Jason Sanders and Edward Kandrot. *CUDA by Example: An Introduction to General-Purpose GPU Programming.* Addison-Wesley Professional, 2010.

[12] John Nickolls, Ian Buck, Michael Garland, and Kevin Skadron. Scalable Parallel Programming with CUDA. *Queue*, 6(2):40–53, 2008.

[13] Athanasios Migdalas, Panos M. Pardalos, and Sverre Storøy. *Parallel Computing in Optimization.* Springer Publishing Company, Incorporated, 2012.

[14] Michael Flynn. Some computer organizations and their effectiveness. *Computers, IEEE Transactions on*, 100(9):948–960, 1972.

[15] Ian Foster. *Designing and Building Parallel Programs*, volume 95. Addison-Wesley Reading, 1995.

[16] David Beasley, Ralph R. Martin, and David R. Bull. An overview of genetic algorithms: Part 1, Fundamentals. *University Computing*, 15:58–69, 1993.

[17] John H. Holland. *Adaptation in Natural and Artificial Systems: An Introductory Analysis with Applications to Biology, Control, and Artificial Intelligence.* MIT Press, 1992.

[18] D Beaslev, David R Bull, and Ralph R Martin. An overview of genetic algorithms: Part 2, Research Topics. *University computing*, 15(4), 1993.

[19] John R. Koza. *Genetic Programming: On the Programming of Computers by Means of Natural Selection.* MIT Press, 1992.

[20] Kenneth A. De Jong. *Evolutionary Computation: A Unified Approach.* MIT Press, 2006.

[21] Jayshree Sarma and Kenneth De Jong. Generation Gap Methods. *Evolutionary Computation*, 1:205–211, 2000.

[22] Edmund Burke, Steven Gustafson, Graham Kendall, and Natalio Krasnogor. Advanced population diversity measures in genetic programming. In *Parallel Problem Solving from Nature—PPSN VII*, pages 341–350. Springer, 2002.

[23] Rasmus K. Ursem. Diversity-guided evolutionary algorithms. In *Parallel Problem Solving from Nature—PPSN VII*, pages 462–471. Springer, 2002.

[24] Alex Rogers and Adam Prugel-Bennett. Genetic drift in genetic algorithm selection schemes. *Evolutionary Computation, IEEE Transactions on*, 3(4): 298–303, 1999.

[25] Justinian P. Rosca. Entropy-driven adaptive representation. In *Proceedings of the Workshop on Genetic Programming: From Theory to Real-World Applications*, volume 9, pages 23–32, 1995.

[26] Jerrold H. Zar. Spearman rank correlation. *Encyclopedia of Biostatistics*, 1998.

[27] Erick Cantú-Paz. A survey of parallel genetic algorithms. *Calculateurs paralleles, reseaux et systèmes repartis*, 10(2):141–171, 1998.