Estudio y análisis de algoritmos genéticos, redes neuronales y lógica difusa para la resolución de problemas de alta complejidad computacional

Autor: Julio César Chávez García

Tesis presentada para obtener el título de:
Ing. En Sistemas Computacionales

Nombre del asesor:
Luis Mateo Patricio Pineda

UNIVERSIDAD VASCO DE QUIROGA

FACULTAD DE INGENIERÍA EN SISTEMAS COMPUTACIONALES

“Estudio y análisis de Algoritmos Genéticos, Redes neuronales y Lógica Difusa para la resolución de problemas de alta complejidad computacional”

TESIS

QUE PARA OBTENER EL TÍTULO DE

INGENIERO EN SISTEMAS COMPUTACIONALES

PRESENTA

JULIO CÉSAR CHÁVEZ GARCÍA

ASESOR

M.I. LUIS MATEO PATRICIO PINEDA

CLAVE: 16PSU0049F

ACUERDO: LIC100846

MORELIA, MICHOACAN SEPTIEMBRE - 2013

A mi familia y amigos.

ÍNDICE GENERAL

RESUMEN

La presente investigación de tipo documental se desarrolla con el fin de formar unas bases sólidas en el campo de la inteligencia computacional, que faculten al desarrollador aplicar los conocimientos teóricos y convertirlos en aplicaciones prácticas que se vean reflejadas en soluciones a problemas de distinta índole. Se estudian los algoritmos genéticos, las redes neuronales y la lógica difusa para conocer las características y ventajas del uso de cada una de estas técnicas. Se desarrollaron un algoritmo genético, así como una red neuronal del tipo "Mapa Auto-organizado de Kohonen" con el propósito de resolver el problema del agente viajero.

ABSTRACT

This documentary research is developed in order to form a solid foundation in the field of computational intelligence, which allows the developer to apply the theoretical knowledge into practical applications that become solutions to different problems. Genetic algorithms, neural networks and fuzzy logic were studied to determine the characteristics and the advantages of the use of each paradigm. A genetic algorithm as well as a "Kohonen Self-Organizing map" were developed with the purpose of solving the traveling salesman problem.

PLANTEAMIENTO DEL PROBLEMA

Existen problemas de gran dificultad de resolución por medio de planteamientos matemáticos y/o con algoritmos comunes. El objetivo de esta tesis es el de analizar y comparar diferentes opciones de procesamiento de datos para la resolución de dicho tipo de problemas.

Consistirá entonces esta investigación en encontrar ventajas y desventajas, así como alcances y limitaciones, comparaciones (diferencias y similitudes), entre cada una de las opciones que serán abordadas, para de esta manera conocer sus bases, funcionamiento y capacidad de resolución de problemas y finalmente desarrollar un algoritmo que aborde el problema del agente viajero.

A lo largo del tiempo y debido a su complejidad, el problema del agente viajero ha sido uno de los problemas más comúnmente estudiados. Este problema consiste en que dadas una cantidad finita de ciudades y las distancias entre éstas, el agente viajero debe visitar cada ciudad exactamente una vez habiendo recorrido la menor distancia posible. En algunas variantes de este algoritmo el principio y final del recorrido deben ser la misma ciudad. Existen también ocasiones en las que no se pretende minimizar la distancia, siendo el objetivo principal la optimización de costos.

Este problema es de los conocidos NP-completos, pues en función del tamaño de su entrada, no puede ser resuelto en forma polinomial.

De lo anterior, es que diversas técnicas computacionales y heurísticas han sido utilizadas con el fin de encontrar la solución óptima a este problema, que hasta la actualidad no ha sido resuelto, pero para el cual se han encontrado soluciones lo suficientemente aceptables en un periodo de tiempo razonable.

ANTECEDENTES

Desde los años cincuenta, se le atribuyen los caminos hamiltonianos a William Rowan Hamilton, los cuales consisten en pasar por cada vértice de un grafo exactamente una vez.

Cien años más tarde George Dantzig, Ray Fulkerson y Selmer Johnson publicaron un método para la solución del problema del agente viajero, titulado "Solutions of a large scale traveling salesman problem".

El problema del agente viajero ha sido objeto de estudio desde hace mucho tiempo, y se ha intentado resolver de diferentes maneras y por numerosos métodos, tales como: la enumeración de todas las soluciones factibles, métodos exactos (o algoritmos óptimos), heurísticas (como la del vecino más cercano), inserción aleatoria, búsquedas locales (como el algoritmo 2-opt), y hasta algoritmos híbridos, que combinen más de una técnica de búsqueda y/o optimización.

Aunque existen muchas aproximaciones y estudios que se han hecho para intentar resolver este problema, sólo se mencionarán algunos estudios recientes en los que se combinan las tres técnicas que serán estudiadas en esta tesis:

- "A Study of Traveling Salesman Problem Using Fuzzy Self Organizing Map" de Arindam Chaudhuri y Kajal De. Desarrollan un algoritmo híbrido que consta de un mapa autoorganizado difuso en combinación con un algoritmo 2-opt y realizan comparaciones frente a los algoritmos evolutivos y el algoritmo de Lin Kerningham.
- "Fuzzy Inspired Hybrid Genetic Approach to Optimize Travelling Salesman Problem" de Bindu et al. publicado en el International Journal of Computer Science & Communication Networks (2012, pp. 416-420). Bindu optimiza los resultados de un algoritmo genético ordinario utilizando lógica difusa en su operador de cruzamiento. Desarrollo llevado a cabo en MATLAB.

En "Using fuzzy evolutionary programming to solve traveling salesman problems" de Jarno Martikainen y Seppo J. Ovaska (2005, pp. 49-54), ambos de la Universidad de Tecnología en Helsinki, se presenta un algoritmo de descomposición controlada difusa para acelerar algoritmos evolutivos cuando se trata de problemas de optimización a larga escala, basados en el método "divide y vencerás".

OBJETIVOS

Objetivo general

Analizar y comprender los algoritmos genéticos, las redes neuronales y la lógica difusa para conocer sus principios y fundamentos; así como sus alcances y limitaciones.

Lo anterior con el propósito de *aprender a implementar algoritmos que automaticen la búsqueda para soluciones a problemas no resueltos* o cuyo espacio de búsqueda no permita encontrar una aproximación aceptable en un tiempo considerable.

Objetivos particulares

Como objetivos particulares de esta tesis se tienen los que se enumeran a continuación:

- Estudiar cada una de las técnicas mencionadas en el objetivo general, con el fin de comprender su funcionamiento e implementación, qué tipo de problemas pueden ser abordados por este tipo de algoritmos, y qué tipo de problemas resultan intratables.
- Hacer una comparación entre los alcances y limitaciones de cada uno de estas técnicas, para conocer bajo qué circunstancias y para qué tipo de problemas resulta más eficiente el uso de cualquiera de los tres.
- Comprender en qué casos se puede hacer uso de más de alguna de las tres técnicas mencionadas (sistemas híbridos) para la solución de problemas.

Desarrollar los algoritmos correspondientes que permitan poner en práctica la teoría estudiada previamente.

ALCANCES Y LIMITACIONES

Alcances

Formar una base de conocimientos sólida sobre cada uno de los las técnicas a tratar, de tal manera que se obtengan las habilidades suficientes para hacer uso práctico de ellas en la resolución de problemas difícilmente manejables por medio de algoritmos comunes.

Limitaciones

La principal limitante radica en el tipo de hardware empleado para implementar los algoritmos, ya que se trata de un ordenador personal sin ningún tipo de hardware especializado en el campo de estudio/desarrollo. Así mismo se tienen las limitantes de recursos humanos (1 persona), materiales y de tiempo (1 año).

JUSTIFICACIÓN

El conocimiento adquirido en los temas que se estudian podrá verse reflejado al pasar de la teoría de estas técnicas al mundo del diseño y desarrollo de aplicaciones y proyectos reales; tal como se ve en la inteligencia computacional aplicada a los videojuegos, aplicaciones en los negocios y aplicaciones financieras, así como ciencias biológicas y medicina, y en la industria misma; desarrollando y diseñando modelos adaptativos y de control, sistemas inteligentes y de predicción, que aplicados a sus respectivas áreas puedan verse plasmados en mejores productos y avances científico-tecnológicos, ya que actualmente la inteligencia computacional es el centro de muchos nuevos desarrollos tecnológicos.

1 INTRODUCCIÓN

Pese a la velocidad de procesamiento con la que cuentan las computadoras hoy en día, es importante el tiempo en que un algoritmo proporciona una respuesta. Entre otros factores, esto se basa en el número de operaciones que dicho algoritmo va a realizar en función del tamaño de los datos de entrada, denominado *orden de complejidad algorítmica*.

La Teoría de la Complejidad Computacional estudia y clasifica problemas computacionales según su dificultad. En esta teoría se tienen diversos tipos de problemas como los de clase P, cuya respuesta se obtiene en un tiempo polinómico con una máquina de Turing; clase NP (polinomial no determinista) cuya solución puede ser obtenida por un algoritmo no determinista en un tiempo polinómico; y finalmente, NP-completo, para la cual es imposible encontrar un algoritmo eficiente que obtenga una solución óptima en un tiempo polinómico.

El problema que se va a tratar (Problema del Agente Viajero) es de tipo NP-completo, en el cual el número de posibles soluciones es aproximadamente n! (n factorial). Para el problema de esta tesis, se tienen un total de 15 ciudades, lo que es igual a $1.307674368 \times 10^{12}$ soluciones diferentes. El número de soluciones posibles crece exponencialmente respecto al número de ciudades a visitar, de tal forma que tomaría años a una computadora encontrar la solución óptima por medio de fuerza bruta.

Frente a esta problemática existen diversas técnicas que intentan aproximarse a la solución óptima en un periodo de tiempo relativamente corto. Entre estas técnicas se encuentran los algoritmos genéticos, las redes neuronales y la lógica difusa.

2 MARCO TEORICO

En este capítulo se presentan las bases teóricas de lo que es la llamada computación evolutiva, así como las redes neuronales y la lógica difusa, para que una vez comprendidos sus fundamentos, se enfoque el estudio a técnicas especializadas como son los algoritmos genéticos y los mapas auto-organizados de Kohonen y a partir de éstas, desarrollar dos algoritmos que permitan obtener soluciones aproximadas al problema del agente viajero, así como realizar una comparación entre ambas técnicas.

Primero se estudiarán las bases de la computación evolutiva, posteriormente los algoritmos genéticos, que son una especialización de la primera. De manera similar sucederá con el estudio de las redes neuronales y el desarrollo de un Mapa Auto-organizado de Kohonen.

2.1 Computación evolutiva

2.1.1 Introducción

La *computación evolutiva* (CE) es una rama de la inteligencia artificial empleada en problemas con espacios de búsqueda extensos y no lineales, en los cuales otros métodos no serían capaces de encontrar soluciones en un tiempo razonable (problemas de optimización combinatoria). La CE (a la cual pertenecen los algoritmos genéticos) también es utilizada en diseño, búsqueda, control y aprendizaje.

La CE toma sus bases de los procesos naturales de evolución, inspirados sobre todo en la selección, reproducción, mutación y competición; encontrados en la obra de Charles Darwin *El origen de las especies* (Darwin, 1859), donde se plasma, como los individuos de determinado medio ambiente intentan adaptarse lo mejor posible a su entorno aumentando la probabilidad de conservación y propagación de sus genes a la siguiente generación.

2.1.2 Antecedentes

La CE comenzó a surgir sin ser considerada como tal desde los años 50 a partir de ciertas ideas que exponían a la evolución natural como un proceso de aprendizaje. Comenzaban a realizarse experimentos tratando de imitar la evolución natural, experimentos de optimización y aprendizaje automático; intentos de programas que lograran mejorarse a sí mismos.

Todas estas ideas expuestas en artículos, *papers* y experimentos fueron contribuciones de Alan Turing, Fraser (1957), Friedberg (1958), Barricelli, entre otros. Hacia los años 60 y 70, H. J. Bremermann había realizado experimentos inspirados en la evolución simulada aplicados a optimizaciones numéricas (Bremermann, 1962).

En 1965 fueron propuestas por Ingo Rechenberg y Hans-Paul Schwefel las estrategias evolutivas. Posteriormente Lawrence J. Fogel introduce una de las primeras técnicas dentro de los lineamientos actuales de la computación evolutiva (Fogel et al., 1966), de ahí que sea considerado como el creador de la *programación evolutiva* (PE); su trabajo consistió en hacer evolucionar autómatas de estados finitos por medio de mutación artificial.

Los *algoritmos genéticos* (AG) fueron introducidos por John H. Holland en los años 60, pretendiendo que las computadoras pudieran aprender por sí mismas e intentando encontrar soluciones aproximadas a problemas de búsqueda. En un principio la técnica introducida por Holland fue concebida con el nombre de "Planes Reproductivos" para posteriormente tomar el nombre de "Algoritmos Genéticos".

Años más tarde, "D. Goldberg, implementó el primer AG aplicado en problemas industriales" (Barrionuevo, 2008), sucediendo a partir de este y otros trabajos que los AG fueran más ampliamente considerados como materia de estudio, y por consecuente, implementados en un mayor número de áreas de desarrollo e investigación.

En un principio sus aplicaciones fueron mayoritariamente teóricas, pero con el paso del tiempo se dio el salto hacia lo comercial, permitiendo así que fueran utilizados en diversas ramas de estudio como la economía, bioquímica, biología molecular, ingeniería aeroespacial, etc.

2.1.3 Fundamentos biológicos

Aunque no se trata de una cuestión vital para el entendimiento de este tipo de algoritmos, resulta importante tener noción de ciertos términos biológicos que son utilizados dentro de la CE.

La evolución fue considerada por Darwin como un proceso de selección natural, donde ciertos miembros de la población sobrevivirán y se reproducirán. Producto de esta reproducción (donde el núcleo de una célula reproductora masculina se fusiona con una célula reproductora femenina) se obtiene una mezcla y combinación de *genes* que contendrá la descendencia.

En la naturaleza, los individuos compiten entre sí, las especies evolucionan para adaptarse al medio que las rodea y tener más posibilidades de sobrevivir.

Nilsson (2001) describe que "de la obra de Darwin, éste defendió el principio de la evolución mediante selección natural, mencionando que:

- Cada individuo tiende a transmitir rasgos a su progenie.
- Sin embargo, la naturaleza produce individuos con rasgos diferentes.
- Los individuos más adaptados, aquellos que poseen los rasgos más favorables, tienden a tener más progenie que aquellos con rasgos no favorables, conduciendo, así, a la población como un todo hacia la obtención de rasgos favorables.
- Durante largos periodos se puede acumular la variación, produciendo especies completamente nuevas cuyos rasgos las hacen especialmente adaptadas a nichos ecológicos particulares". (p. 543)

"Los seres vivos funcionamos y nos perpetuamos gracias a nuestra capacidad de obtener energía del medio ambiente. Esta energía es utilizada en el organismo para realizar trabajos y procesos celulares" (Barrera, 1992, p. 9).

"En plantas y animales superiores, cada célula[1] contiene un solo núcleo que, a su vez, contiene *cromosomas;* En general, los cromosomas vienen por pares, y cada padre contribuye con un cromosoma para cada pareja" (Nilsson, 2001, p. 542).

El ácido desoxirribonucleico (ADN) es el material genético de muchos organismos, modelado como una doble hélice que se asemeja a una escalera girando sobre una columna cilíndrica imaginaria (Figura 2.1).

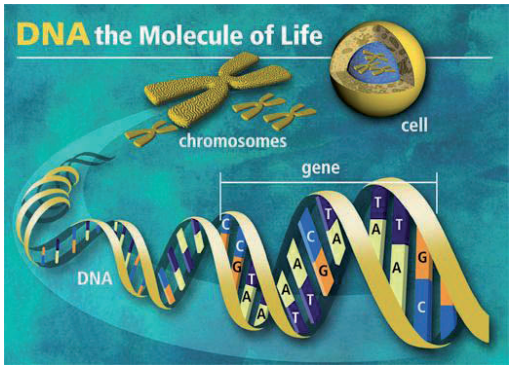

Figura 2.1. Representación del ADN.

Los *genes* radican en los cromosomas, a lo largo de la hebra de ADN. En otras palabras, están formados por un segmento del ADN y son la unidad básica del material hereditario.

Los atributos esenciales del gen se le atribuyen a Gregor Mendel quien "realizó cruzas entre plantas de diferentes variedades de chícharos con características morfológicas distintas y bien definidas" (Barrera, 1992, p. 19).

[1] Componente básico de los seres vivos.

De su trabajo se concluyó que las características de los individuos provienen del padre y la madre como resultado de la aportación de factores hereditarios (genes).

Las características visibles de los organismos se deben a los genes; este conjunto de características físicas, tales como el color de cabello, ojos, piel, complexión, etc. son heredadas de los padres y se les conoce como *fenotipo*.

El papel de los genes en el organismo se revela mediante cambios en el fenotipo llamados *mutaciones*, que ocurren ocasionalmente durante el proceso de copiado de cromosomas cuando algunas de las secuencias del ADN del total de la constitución genética o *genotipo* del organismo es modificada, y se produce un gen alterado distinto al correspondiente del padre, esto hace que se pierda o se modifique la función del gen codificado.

Dicho de otra manera, la mutación podría describirse como cualquier cambio en la secuencia del ADN. Las mutaciones genéticas pueden ser de dos tipos: *mutación en un punto*, en la cual es cambiada una simple base; y *mutación multipunto*, en la cual una o más bases (no múltiplos de tres) son insertadas o borradas (Paton, 1997, traducido por autor, handbook p. a.2.2:7 sección a.2.2.4)

"Los efectos de las mutaciones pueden ser ligeros o muy severos, dependiendo de la función de la proteína codificada por el gen afectado" (Barrera, 1992, p. 19). Si el propósito del gen no mutado es, digamos, determinar la forma de una enzima crucial, el gen mutado puede determinar una mejor enzima[2] (Nilsson, 2001, p. 543).

[2] Molécula formada principalmente por proteína que producen las células vivas y que actúa como catalizador y regulador en los procesos químicos del organismo.

(http://www.wordreference.com/definicion/enzima)

Un gen puede presentar formas alternativas que determinan la expresión de alguna característica particular. Estas formas alternativas de un mismo gen se conocen como *alelos.*

Al prepararse para el apareamiento, los cromosomas se juntan, duplican y forman haces que se ven como cuerdas trenzadas que parecen producir tensiones que conducen a una gran cantidad de divisiones y reconexiones, mezclando así los genes de los cromosomas implicados (Nilsson, 2001, p. 542). A este intercambio de fragmentos entre cromosomas se le llama *recombinación*.

Desde un punto de vista molecular, la selección natural puede realizarse mediante la variación que se produce a partir de la recombinación y la mutación (Nilsson, 2001, p. 544). La recombinación es un operador aplicado a dos o más candidatos seleccionados e integra genes existentes en nuevas combinaciones. La mutación es aplicada a un candidato específico produciendo nuevos genes.

En resumen, los organismos están formados por células, que contienen a su vez un mismo número de cromosomas. Los cromosomas son cadenas de ADN y se constituyen de genes. Los genes codifican información, por ejemplo el color de cabello. En los genes se tienen valores (rubio, castaño, negro) llamados alelos. Los genes son conocidos por su posición en el cromosoma, a esto se le conoce como *locus*.

Tabla 2.1 Mapeo de términos de la naturaleza y su representación en el mundo de los AG

Naturaleza	**Computadora**
Población	Conjunto de posibles soluciones
Individuo	Una posible solución
Adecuación	Calidad de la solución
Cromosoma	Codificación de una solución
Gen	Parte de la codificación de una solución
Alelo	Posibles valores de un gen

Nota: La tabla 2.1 presenta una comparación entre la naturaleza y los AG en computación empleando algunos de los términos introducidos en este apartado.

2.1.4 Generalidades de la computación evolutiva

Dentro de la CE se encuentran principalmente los siguientes paradigmas:

- Estrategias evolutivas.
- Programación evolutiva.
- Algoritmos genéticos.
- Programación genética.

Todos los anteriores siguen prácticamente el diagrama de flujo de la Figura 2.2, difiriendo en esencia, en los detalles técnicos. Por ejemplo, mientras un algoritmo genético utiliza cadenas sobre un alfabeto finito (la mayoría de las veces binario); la programación genética, utiliza árboles como su principal estructura de datos.

Así, técnicamente, una representación posible puede ser preferida sobre otras si ésta encaja mejor en el problema dado, esto es, que sea más fácil o natural el codificar las soluciones candidatas (Eiben & Smith, 2003, p. 18).

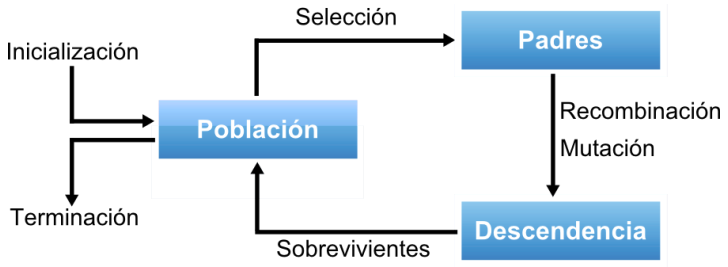

Figura 2.2. Diagrama de flujo general de un algoritmo evolutivo (Eiben & Smith, 2003, p.17).

El siguiente pseudo-código correspondiente al diagrama anterior.

```
COMIENZO
   INICIALIZAR población con soluciones candidatas aleatorias;
   EVALUAR cada candidato;
   MIENTRAS (no se satisfaga la CONDICIÓN DE TERMINACIÓN) HACER
      SELECCIONAR padres;
      RECOMBINAR padres;
      MUTAR descendencia resultante;
      EVALUAR nuevos candidatos;
      SELECCIONAR individuos para la siguiente generación;
   FIN HACER
FIN
```

Figura 2.3. Pseudo-código general de un algoritmo evolutivo.(Breve Descripción del Algoritmo)

"En CE se trata a la naturaleza como una inmensa máquina de resolver problemas y se intenta utilizar esa potencialidad por medio de programas" (Barrionuevo, 2008). De esta forma, como una analogía directa con el comportamiento natural, la idea básica detrás de los algoritmos evolutivos (AE) consiste en evolucionar cierta población (datos de entrada) que pueden ser generados aleatoriamente (de aquí que se consideren como algoritmos probabilistas o estocásticos), donde sus mejores individuos son seleccionados para formar una nueva población con esperanzas de que ésta sea mejor que la anterior.

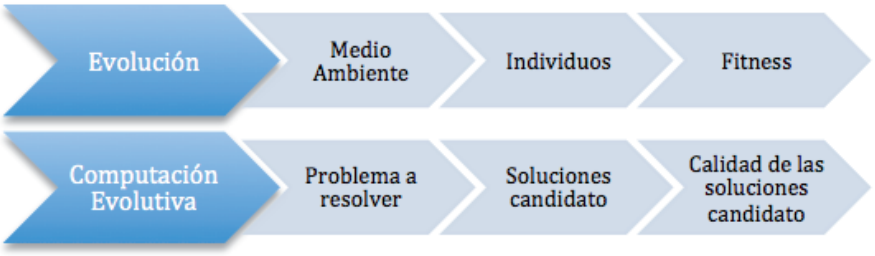

Figura 2.4. Comparativa entre evolución y computación evolutiva (El Huron Inteligente, 2012).

Cada individuo representa una solución potencial a un problema dado. Mediante una función de evaluación se mide la adecuación de estos individuos, resultado de las operaciones genéticas ya mencionadas (recombinación, mutación, etc.), proporcionando así algoritmos de búsqueda simples, pero adaptativos, potentes y robustos.

"Después de cierto número de generaciones, el programa converge, esperando que el mejor individuo represente una solución (razonable) próxima a la solución óptima" (Michalewicz, 1998, p. 2, traducido por autor).

Los AE suelen encontrar buenas aproximaciones a la solución de los problemas abordados, aunque corren el riesgo de caer en soluciones óptimas locales y no globales, esto debido a que en ocasiones algunos individuos son muy superiores a la media en cuanto a su adecuación, impidiendo así que otros individuos contribuyan con su descendencia en la siguiente generación, lo que se ve reflejado en pérdida de información e incapacidad para evolucionar. Aún así es posible aplicar ciertas técnicas para mantener la diversidad de los individuos y contar con una mayor probabilidad de acceder o acercarse lo más posible al máximo global o solución óptima.

Barrionuevo (2008) sustenta que "un AE no requiere muchos conocimientos matemáticos o propios del problema para el cual está siendo diseñado. Debido a su naturaleza evolutiva, el AE busca soluciones sin considerar un conocimiento específico del problema, es decir, son métodos *débiles*" (p. 434).

Un AE puede trabajar con varias soluciones simultáneamente, en lugar de hacerlo de forma secuencial. Además de su facilidad de ejecución en arquitecturas modernas paralelas, utilizan operadores probabilísticos en lugar de deterministas.

"También puede un AE implementar cualquier clase de función o funciones objetivo y/o restricciones, así como representar cualquier conjunto de variables de decisión definidas en espacios continuos, discretos o híbridos" (Barrionuevo, 2008, p. 434).

Puede suceder que los AE tarden mucho en converger, o no hacerlo. Por otro lado, pueden converger prematuramente, arrojando soluciones no óptimas. Todo esto dependiendo en parte de los parámetros usados en los algoritmos. Los máximos locales resultan más fáciles de manejar manteniendo la diversidad, pues habiendo suficientes individuos para poblar todos los máximos locales, es más sencillo que algún individuo acceda al máximo global.

2.1.5 Componentes de los algoritmos evolutivos

En general, los AE tienen ciertos componentes, procedimientos u operadores que los definen. Además de las funciones de iniciación y terminación se tiene que es importante contar con:

- una representación apropiada (definición de individuos),
- una forma de crear una población inicial de soluciones,
- una función de evaluación o adecuación,
- un conjunto de operadores evolutivos,
- parámetros y mecanismos de selección y reemplazo.

2.1.5.1 Representación (definición de los individuos).

Consiste en ligar el "mundo real" con el "mundo del AE". En el diseño de un AE es importante una representación apropiada de las soluciones del problema porque así se establece "un puente entre el problema original y el espacio de búsqueda donde

se tratará de encontrar la solución" (Eiben & Smith, 2003, p. 18, traducido por autor). De esta representación depende la eficiencia del algoritmo.

Como una analogía a los conceptos revisados anteriormente, los objetos que forman las posibles soluciones en el contexto del problema original son llamados *fenotipos;* su codificación, los individuos dentro del AE, son llamados *genotipos* (Eiben & Smith, 2003).

La representación constituye comúnmente el primer paso del diseño y equivale a hacer un mapeo de los fenotipos hacía los genotipos que los representarán. Por ejemplo, el número 10 extraído de un problema del "mundo real" (numeración en base decimal), sería considerado un fenotipo, y 1010 el genotipo que lo representa en el mundo del AE (con numeración binaria).

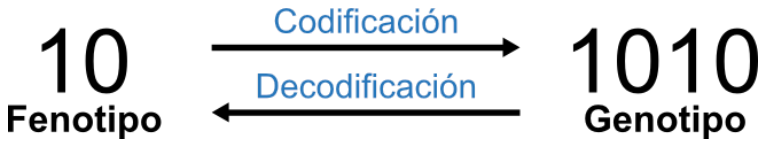

Figura 2.5. Representación de los datos.

La búsqueda evolutiva completa se lleva a cabo en el espacio de los genotipos. Una solución - un buen fenotipo - es obtenido decodificando el mejor genotipo una vez terminado el programa (Eiben & Smith, 2003).

Otro punto importante son los sinónimos que pueden llegar a ser utilizados; por ejemplo, una *solución candidata*, un *fenotipo* o un *individuo*, son utilizados en el contexto del problema original para denotar puntos dentro del posible espacio de soluciones. Por otro lado, los términos *genotipo*, *cromosoma* o un *individuo* son usados para puntos en el espacio donde la búsqueda se llevará a cabo.

Finalmente, el termino *representación* así como puede ser utilizado para el mapeo de fenotipo a genotipo (codificar, decodificar) o vice versa, puede también, ser empleado en el sentido de las "estructuras de datos" que serán utilizadas.

2.1.5.2 Inicialización

La mayoría de las veces los individuos de la primer población son generados de forma aleatoria dentro del espacio de búsqueda, aunque la incorporación de conocimiento (el uso de heurísticas) puede ayudar, dependiendo de la aplicación y/o el problema a tratar.

2.1.5.2.1 *Población*

Es la unidad de la evolución, se trata de un grupo de individuos que representan posibles soluciones a un problema dado. En palabras de Eiben & Smith "los individuos son estáticos, no cambian ni se adaptan, la población sí" (Eiben & Smith 2003, p. 20, traducido por autor). El número de individuos es el tamaño de cierta población, que en muchos de los casos es estática, aunque en otros puede variar durante la búsqueda.

Las poblaciones pueden tener diferentes tipos de estructuras; desde estructuras simples, hasta estructuras sofisticadas con información adicional como medidas, espacios, relaciones, etc.

Las *operaciones de selección* (selección de padres y sobrevivientes) funcionan sobre poblaciones, no así en el caso de los *operadores de variación*, que actúan más bien, sobre individuos (Eiben & Smith, 2003). Por ejemplo, la mutación se lleva a cabo sobre un individuo específico, mientras la selección de un individuo se lleva a cabo sobre una población de individuos.

La diversidad de una población es la medida del número de diferentes soluciones presentes.

2.1.5.3 Función de evaluación o adecuación

Esta función representa los requerimientos de adaptación y las bases para la selección. Sirve para "medir la adecuación de cualquier solución y hará el papel de entorno, en el cual las mejores soluciones, esto es, aquellas con mejor adecuación, tengan mayor probabilidad de selección y supervivencia" (Barrionuevo, 2008, p. 435).

Esta función define lo que se considera como el *mejoramiento*, asignando una medida de calidad a los genotipos.

A menudo, el problema original es un problema de optimización. En este caso el nombre *función objetivo* es comúnmente utilizado en el contexto del problema original, y la *función de evaluación* o adecuación puede ser idéntica o simplemente una transformación de la función objetivo.

2.1.5.4 Operadores evolutivos

Éstos actúan como reglas de transición probabilísticas para guiar la búsqueda. Combinan entre sí soluciones candidatas (individuos) existentes con el propósito de obtener otras nuevas.

Según Eiben & Smith (2003), "los operadores de variación en la CE son divididos en dos tipos basados en su aridad" (p. 21, traducido por autor), es decir, el número de objetos que puede tener como entradas.

2.1.5.4.1 *Mutación*

Es un operador de variación que se aplica a un genotipo, obteniendo de este un hijo o descendencia ligeramente modificada. El operador es siempre estocástico causando un cambio aleatorio, esto descarta que algún operador heurístico que actúe sobre algún individuo para modificarlo pueda ser considerado como operador de mutación.

El rol de la mutación varía según el paradigma de CE utilizado. Por ejemplo, en la programación genética su uso es casi nulo; en los algoritmos genéticos es más utilizado; mientras que en la programación evolutiva es el único operador de variación utilizado durante la búsqueda.

2.1.5.4.2 *Recombinación o cruzamiento.*

Es un operador de variación que combina, como su nombre lo indica, información (características) de dos genotipos padres en uno o más genotipos hijos o de descendencia; donde las partes que serán combinadas de cada padre son elegidas aleatoriamente, por lo tanto, la recombinación es también un operador estocástico.

Parecido como sucede en la mutación, la recombinación es el único operador utilizado en la programación genética; en los algoritmos genéticos, es probablemente el operador más utilizado; mientras que en la programación evolutiva, su uso es casi nulo.

Aunque biológicamente no sucede así, matemáticamente, es posible implementar de una manera relativamente fácil, el operador de recombinación con aridad mayor a 2. Ha sido probado a través de muchos estudios, que esto puede llegar a tener efectos positivos durante la evolución (Eiben & Smith, 2003).

El principio de la recombinación es simple: que por medio del apareamiento de dos individuos con diferentes características deseables, podamos producir una descendencia la cual combine ambas características (Eiben & Smith, 2003). Este principio ha sido exitoso durante mucho tiempo en la naturaleza.

Es necesario tener en cuenta, que la descendencia creada aleatoriamente por los algoritmos evolutivos, arrojará en algunos casos combinaciones poco deseables, incluso combinaciones peores que las de sus antecesores.

Cabe recalcar que la mutación y la recombinación respectivamente, entre otras cosas, ayudan a crear diversidad y a explotar soluciones factibles para obtener una mejor solución. También es importante hacer notar que estos operadores dependen de la representación. Por ejemplo, utilizando genotipos binarios, la mutación puede verse como la inversión del bit (0 a 1 y vice versa); por otro lado, utilizando estructuras de árboles, el operador de mutación tendría que ser diferente.

2.1.5.5 Parámetros y mecanismos de selección y reemplazo

La evolución de los AE se va guiando a través de los valores que puedan llegar a tomar ciertos parámetros, tales como el tamaño de la población, el número de iteraciones, etc. Es importante un adecuado diseño del algoritmo de acuerdo al problema que se va a enfrentar, poniendo énfasis en cuestiones como: el número de cromosomas en una población, pues en una población pequeña pronto todos los cromosomas tendrán rasgos idénticos, por otro lado, una población grande requerirá un alto tiempo de cálculo; el tipo de mutación, que puede derivar o no, en poblaciones ligadas o desligadas de sí mismas; el permitir que un cromosoma aparezca más de una vez en la población, etc.

Los mecanismos de selección nos permiten distinguir entre los individuos de acuerdo a su calidad. Seleccionando en base a ciertos parámetros los individuos que se convertirán en padres para la próxima generación, con el afán de obtener un incremento en la calidad de los individuos (posibles soluciones).

Por lo general, *la selección de padres* es probabilística, siendo que los individuos con mayor calidad tienen más posibilidades de ser escogidos. Aun así, los individuos con menor calidad también tienen oportunidad de ser seleccionados, lo que ayuda a mantener una mayor diversidad y evitar pérdida de información que puede ser útil en futuras generaciones. Esto sirve para evitar que la búsqueda se vuelva voraz y se caiga en un óptimo local.

Además de contar con un mecanismo para la selección de padres, se debe contar con un *mecanismo para la selección de sobrevivientes (reemplazo),* que es generalmente determinista y que se basa también en parámetros de calidad en favor de los individuos.

Este mecanismo es llamado una vez que se ha obtenido la descendencia de los padres seleccionados por el mecanismo anterior. Los *valores de adecuación* (fitness) determinan que individuos pasan a la siguiente generación.

2.1.5.6 Condición de término

Debe existir una condición que asegure que el algoritmo se detendrá en determinado momento. Se tiene que el algoritmo puede terminar cuando se alcance el óptimo o se encuentre cerca de éste, dentro de un rango de variación (esto es determinado por la función de evaluación); el algoritmo también puede terminar cuando se cumpla alguna de las siguientes opciones según Eiben & Smith (2003) comúnmente utilizadas:

- "cuando se alcanza el máximo tiempo de procesamiento permitido por el CPU,
- el número total de evaluaciones de adecuación alcanza el límite dado,
- si durante un periodo de tiempo (por ejemplo, el número de generaciones o el número de evaluaciones de adecuación), el mejoramiento de la adecuación se mantiene bajo un valor de umbral (no hay mejoría significante),
- la diversidad de la población cae bajo un umbral dado" (p. 24, traducido por autor).

en caso de no conocerse un óptimo, sólo se toman en cuenta las opciones anteriores.

2.1.6 Funcionamiento de los algoritmos evolutivos

Existen ciertas consideraciones a tomar en cuenta sobre los algoritmos evolutivos. Tomando como ejemplo una función objetivo a maximizar, la figura 2.6 muestra tres estados de la búsqueda: el comienzo, un punto en medio de la búsqueda y el final de la evolución.

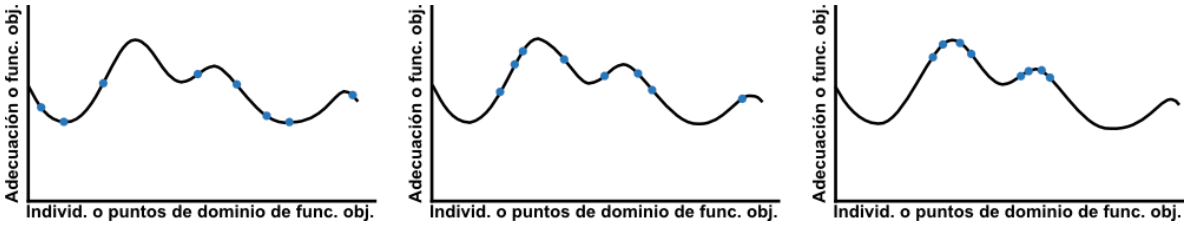

Figura 2.6. Proceso de un AE, ilustrado en términos de su distribución de población (Eiben & Smith, 2003).

Inicialmente, los individuos son aleatoriamente esparcidos sobre el espacio de búsqueda. Después de algunas generaciones (segunda fase), una vez aplicados los operadores anteriormente descritos, la población comienza a situarse en regiones de mejor adecuación respecto a la función de evaluación. Finalmente, cuando los parámetros han sido correctamente establecidos, la población se encuentra concentrada alrededor de unos pocos picos cercanos al óptimo.

Puede suceder que la población "escale la colina equivocada" y todos los individuos estén posicionados alrededor de un óptimo local, pero no del global. Los AE de búsqueda pueden llegar a ser considerados en términos de compensación entre la exploración[3] y explotación.[4]

La "convergencia prematura" es un efecto que puede darse, resultado de la pérdida de diversidad de la población, lo que conlleva al algoritmo a quedar atrapado

[3] Nuevos individuos en regiones no exploradas.

[4] Concentración de individuos en soluciones aceptables.

en un óptimo local (Eiben & Smith, 2003). Sin embargo, existen técnicas para contrarrestar estos efectos, algunas de las cuales serán abordadas más adelante.

Acerca del comportamiento del algoritmo durante la búsqueda. La figura 2.7 muestra la adecuación de la población en el dominio del tiempo.

Figura 2.7. Adecuación de la población en el dominio del tiempo (Eiben & Smith, 2003).

Por otra parte, la Figura 2.8 sirve de apoyo para considerar si en realidad se obtiene alguna ventaja de utilizar heurísticas de inicialización para obtener una mejor primera población. Desde esta perspectiva, podemos cuestionar si es necesario dicho esfuerzo, considerando que la población obtenida a partir de la heurística puede ser rápidamente alcanzada después de *n* generaciones.

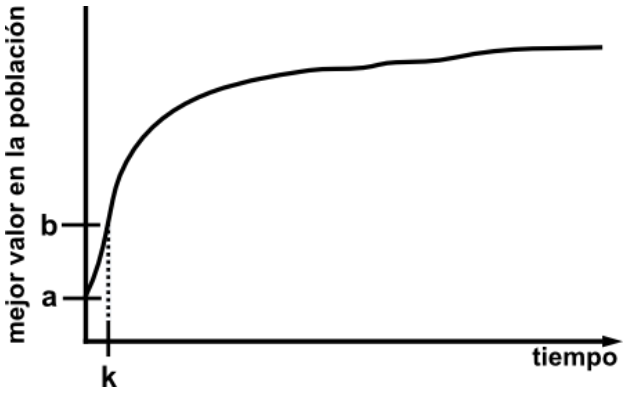

Figura 2.8. Ilustra el hecho de que bajo ciertas circunstancias resulte cuestionable la inicialización heurística (Eiben & Smith, 2003).

Este comportamiento de evolución también puede ser útil para indicar la condición respecto a la terminación del algoritmo. La figura 2.9 ilustra el progreso en la evolución de la población durante la primera y segunda mitad del tiempo de búsqueda, resultando cuestionables una vez más, el esfuerzo y tiempo necesarios para obtener cierta solución.

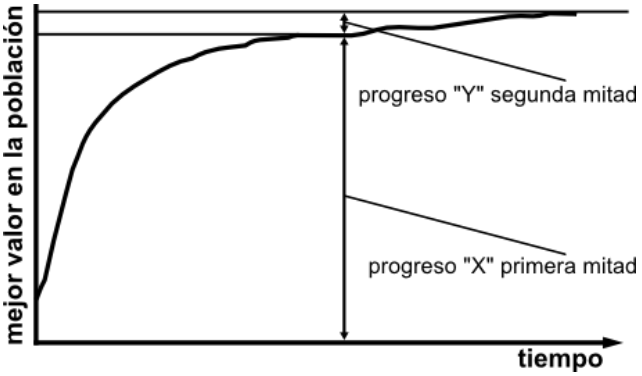

Figura 2.9. La gráfica muestra el progreso en la evolución de la población respecto al tiempo de ejecución del algoritmo (Eiben & Smith, 2003).

Finalmente, es necesario mencionar que pueden diseñarse algoritmos para problemas específicos y estos algoritmos generalmente presentan mejor rendimiento, pero sólo en el problema para el cual fueron diseñados. Mientras más alejados del problema para el cual los algoritmos específicos han sido diseñados, más rápidamente pierden rendimiento.

Es por ello, que el gran campo de aplicación de los AG se encuentra en donde no existen técnicas especializadas, sin embargo, se puede pensar también en algoritmos híbridos que arrojen mejores resultados.

Existen una gran variedad de algoritmos de búsqueda y optimización, como los de búsqueda local, métodos enumerativos y los de optimización combinatoria que utilizan diversas estructuras de datos para tratar de alcanzar su objetivo. El espacio de búsqueda de estos algoritmos se vuelve prácticamente infinito en muchos de los casos, por lo que resultan útiles sólo cuando el número de valores es manejable.

Algunos de ellos encuentran la solución óptima si ésta existe, otros quedan estancados en óptimos locales que pueden llegar a ser sumamente inferiores al óptimo global.

Entre las diferencias que podemos encontrar en los AE respecto a algoritmos como los de búsqueda local, es que en los AE trabajan con cierta población que define una función de distribución probabilística no uniforme, en contraste con las distribuciones uniformes de otros algoritmos. Eiben & Smith (2003) plantean que:

> La ventaja que da a los AE mantener la diversidad en sus individuos, es que de esta manera se puede escapar de los óptimos locales, se puede trabajar en espacios discontinuos de búsqueda; y si se pueden mantener copias de una solución, proveen una manera natural y robusta de tratar con problemas con ruido y falta de certeza asociada a la asignación de la calificación de adecuación de una solución candidata. (p. 34, traducido por autor)

2.2 Algoritmos Genéticos

Son éstos, el tipo de AE más ampliamente conocido y el cual será utilizado para resolver el problema del agente viajero. Como ya fue mencionado, se trata de métodos de búsqueda y/o de optimización basados en la selección natural.

Haciendo una analogía con la naturaleza los AG codifican cadenas finitas, generalmente binarias, en donde cada cadena es una solución candidata conocida como *cromosoma*, los alfabetos utilizados se conocen como *genes* y los valores de estos genes como *alelos* (Sastry, Goldberg, & Kendall, 2005).

2.2.1 Representación de los individuos

Utilizando términos biológicos, una solución a determinado problema sería llamado individuo.

Aunque tradicionalmente los AG han utilizado cadenas binarias, también han sido desarrolladas aplicaciones más recientes utilizando otro tipo de representaciones, tales como: vectores, listas ordenadas, grafos, expresiones lisp o parámetros reales.

En los AG clásicos, dada la representación binaria, a veces es necesario modificar el problema, y es aquí cuando hay que tener en cuenta si resulta mejor modificar el problema o la representación de los datos. Por ejemplo, citando a Michalewicz (1998):

> La representación binaria tradicional utilizada en los algoritmos tiene algunos inconvenientes al ser aplicados a problemas multidimensionales, de alta precisión. Por ejemplo, para 100 variables con dominios en el rango de [-500, 500] donde se necesita una precisión de seis dígitos después del punto decimal, la longitud de la solución binaria es 300. Esto genera un espacio de búsqueda de aproximadamente 10^{1000}. Para problemas como éste, el algoritmo genético se desempeña pobremente. (p. 97)

Así, el primer paso en la construcción de cualquier algoritmo evolutivo es escoger la representación adecuada, definiendo el genotipo y llevando a cabo el mapeo entre genotipo y fenotipo. Es importante la correcta elección de la representación de los datos, porque de esta depende, en parte, el desempeño del algoritmo (Eiben & Smith, 2003).

Es importante tener en cuenta qué representaciones óptimas para abordar determinados problemas, pueden no serlo para algunos otros; y que en algunas ocasiones puede llegar a ser más natural utilizar representaciones mixtas para atacar el problema.

Una vez que ha sido codificada la información y se ha obtenido una función de evaluación de las soluciones, se puede proceder con los pasos mostrados en las

figuras 2.2 y 2.3: inicialización → evaluación → selección → recombinación → mutación → reemplazo.

2.2.1.1 Representación binaria

La representación utilizada normalmente en un AG consta de cadenas binarias de bits (0's y 1's). Se tiene que decidir la longitud de la cadena, en función del número de variables en la solución y el número de bits necesarios para codificarlas, ya que es importante asegurarse que la codificación permita que todas las cadenas posibles de bits denoten una solución válida, y vice versa; que todas las soluciones puedan ser representadas en términos de bits.

Existen ocasiones que apoyarse en la codificación Gray pueda resultar útil para la obtención de resultados, dado el peso de los bits. Por ejemplo, cambiar de 3 a 4 resulta igual que cambiar de 3 a 2 en notación decimal (un paso en ambos casos), mientras en binario los cambios serían 011 → 100 y 011 → 010 (tres y un paso respectivamente). La notación Gray se asegura de que enteros consecutivos se obtengan mediante el cambio de un sólo bit.

2.2.1.2 Representación con enteros

Las representaciones binarias no siempre son lo más adecuado como ya se revisó anteriormente, una opción en estos casos son las representaciones por medio de ordinales.

2.2.1.3 Representación con punto flotante y valores reales

Resulta útil cuando se requieren específicamente cadenas de valores reales o los datos provienen de una distribución continua más que de una discreta, teniendo como limitante la cantidad de decimales en la implementación cuando se utiliza el punto flotante.

2.2.1.4 Representación con permutación

Son problemas en los que las soluciones son representadas en manera natural, como problemas en los que se tiene que decidir el orden en el cual una secuencia de eventos debe ocurrir. Por ejemplo, una cadena en un AG ordinario permite que los números ocurran más de una vez, dichas secuencias de enteros no representarán soluciones válidas. Algoritmos de planificación de tareas o el problema del viajero son dos ejemplos de situaciones que se manejan por medio de permutación.

2.2.2 Inicialización

La inicialización puede ser aleatoria, aunque se puede aplicar conocimiento específico del problema para iniciar con mejores soluciones candidatas (Sastry et al., 2005), teniendo en cuenta (como ya se trató anteriormente) el esfuerzo necesario para lograr esto. Esta población está formada por *n* número de individuos, donde *n* es un parámetro de entrada.

2.2.3 Evaluación

Una vez inicializada la población, o creadas las poblaciones descendientes, se mide su adecuación por medio de una función de evaluación o función objetivo y consiste en evaluar el fenotipo de una posible solución.

2.2.4 Selección

Se basa en la supervivencia del más apto, de esta forma se guardan copias de las mejores soluciones, es decir, las que han tenido un mayor valor de adecuación. Bajo el criterio de selección establecido, se escogen aquellos individuos de la población que se reproducirán. Existen diferentes mecanismos de selección, a continuación se tratan algunos de ellos.

2.2.4.1 Selección proporcional a la función de desempeño

Depende de la adecuación absoluta de un individuo en comparación con los valores de la adecuación absoluta del resto de la población. En este tipo de selección

se incluyen la rueda de ruleta (Goldberg, 1989, citado en Sastry et al., 2005) y la selección estocástica, donde el tamaño de los espacios en la rueda para cada individuo reflejan la probabilidad de selección.

2.2.4.1.1 *Selección utilizando la rueda de la ruleta*

Se evalúa la adecuación f_i de cada individuo en la población. A cada cromosoma de la población se le asigna la siguiente probabilidad:

$$p_i = \frac{f_i}{\sum_{j=1}^{n} f_j}$$

donde *n* es el tamaño de la población. Se calcula en seguida, la probabilidad acumulada *q* para cada individuo:

$$p_i = \sum_{j=1}^{i} p_j$$

posteriormente se genera un número aleatorio $r \in (0, 1]$. Si $r \leq q_1$ se selecciona el primer cromosoma de la generación, x_1, en otro caso se selecciona el *i-ésimo*, cromosoma x_i, tal que $q_{i-1} < r \leq q_i$.

Este tipo de selección tiene algunos inconvenientes como la convergencia prematura a causa de individuos mucho mejores que otros y poca variación en la selección, dados los valores de adecuación muy similares entre los individuos.

Para ilustrar lo explicado, en la Tabla 2.2 se puede apreciar un ejemplo con cinco individuos, sus adecuaciones, probabilidades de ser seleccionados y la probabilidad acumulativa, cuando la adecuación total es igual a 95.

Tabla 2.2 Ejemplo de selección proporcional

Cromosoma	Adecuación f_i^t	Probabilidad p_i^t	Probabilidad acumulada q_i^t
1	28	28/95=0.295	0.295
2	18	0.189	0.484
3	14	0.147	0.631
4	9	0.095	0.726
5	26	0.274	1.000

(Sastry, Goldberg, & Kendall, 2005).

Suponiendo que se genera un número aleatorio *r*, por ejemplo 0.585, se selecciona el tercer cromosoma, ya que $q_2 = 0.484 < 0.585 \leq q_3 = 0.63$.

2.2.4.2 Selección por ranking

Los tipos de selección anteriores pueden tener problemas cuando las diferencias de adecuación son grandes pues la probabilidad de ser seleccionados de los individuos con adecuación baja es casi nula. Este tipo de mapeo aplica primero un ranking a los individuos y de acuerdo a esto, cada cromosoma recibe una adecuación. Existen dos principales tipos de mapeo: lineal y exponencial. Las figuras 2.10 y 2.11 muestran la diferencia entre la selección por adecuación y por ranking.

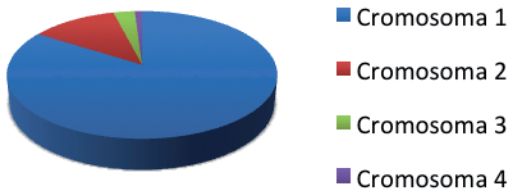

Figura 2.10. Población en base a su adecuación.

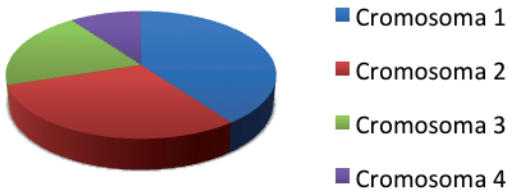

Figura 2.11. Población en base a ranking.

2.2.4.3 Selección por torneo

Existen ocasiones en que la población es demasiado grande o se encuentra distribuida, de tal suerte que resulta costoso en tiempo obtener información de cada uno de los individuos. En este caso la selección por torneo es útil además de fácil de implementar. No es necesario tener información de toda la población y puede utilizarse con o sin reemplazo, características por las cuales es quizá el método de selección más utilizado.

En la selección por torneo, se seleccionan aleatoriamente *k* cromosomas (generalmente con *k* = 2*)*, compiten entre ellos a través de una función de adecuación, a partir de la cual se selecciona el individuo más apto como padre. El número de torneos requeridos varía según el número *n* de individuos a seleccionar.

La probabilidad de que un individuo sea seleccionado en torneo, depende principalmente de su rango en la población, el tamaño del torneo, la probabilidad de que el más adaptado sea seleccionado y de si los individuos son seleccionados con o sin reemplazo (Eiben & Smith, 2003).

2.2.5 Recombinación (cruzamiento)

En el proceso de recombinación se crea nueva descendencia a partir de dos o más padres, de los cuales se toma una parte para generar una nueva solución con la esperanza de que sea mejor que sus antecesores. Es uno de los operadores más

importantes de los algoritmos genéticos donde una parte de la calidad de las nuevas soluciones depende del adecuado diseño de los mecanismos de recombinación.

En la mayoría de los operadores de recombinación, ésta es aplicada probabilísticamente a dos individuos en base a una probabilidad de cruzamiento p_c, en el rango [0.5, 1.0]. Se genera un número aleatorio *r* generalmente en el rango de [0, 1), si $r \leq p_c$, entonces ambos padres son sometidos a recombinación; en caso contrario, si $r > p_c$, los dos individuos de la descendencia se producen realizando copias de los padres.

2.2.5.1 Cruzamiento en un punto

Junto con el cruzamiento de dos puntos, sea quizá uno de los métodos más utilizados. El procedimiento consiste en seleccionar un punto aleatorio entre [0, *l*-1], donde *l* es la longitud de la cadena. En este punto, ambos padres son divididos en 2 y los genes resultantes después del punto de cruce son intercambiados como lo ilustra la Figura 2.12.

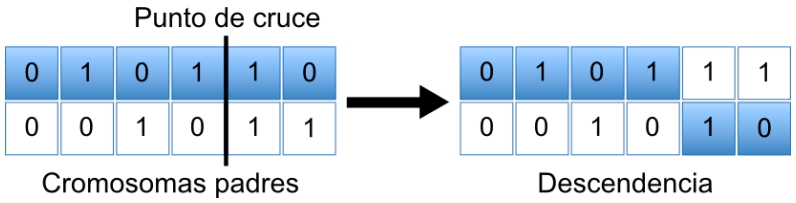

Figura 2.12. Cruzamiento en un punto.

2.2.5.2 Cruzamiento en k puntos

En este caso las cadenas pueden ser divididas en segmentos de dos o más genes continuos, es decir, se seleccionan k puntos aleatorios en el rango de [0, *l*-1].

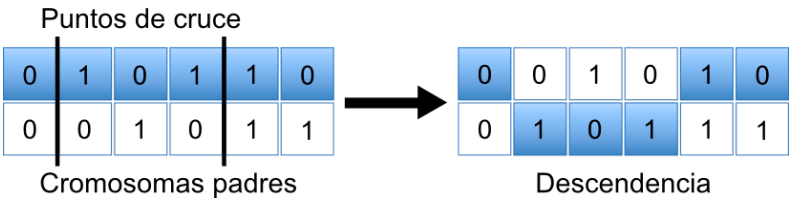

Figura 2.13. Cruzamiento en k puntos, donde k = 2.

2.2.5.3 Cruzamiento uniforme

El cruzamiento uniforme hace uso de una probabilidad de intercambio $\boldsymbol{p_e}$, usualmente 0.5; se generan valores aleatorios dentro del rango de [0, 1] para cada gen del cromosoma y finalmente aquellos valores menores a $\boldsymbol{p_e}$ son tomados del primer padre, los valores mayores a $\boldsymbol{p_e}$ son tomados del segundo padre.

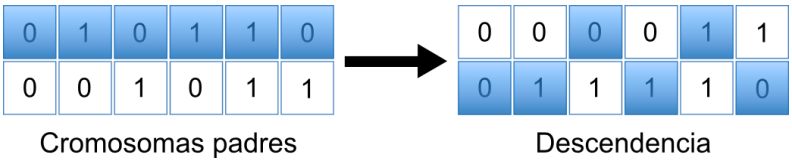

Figura 2.14. Cruzamiento uniforme.

Mientras el cruzamiento en *k* puntos tiende a mantener juntos los genes localizados unos cerca de otros en su representación, el cruzamiento uniforme no exhibe una tendencia proporcional, sino más bien tiende a transmitir el 50% de los genes de cada padre. Resulta imposible determinar cuál de estos operadores se comporta mejor frente a cierto problema. Sin embargo, al momento de diseñar un algoritmo para un problema específico, es de gran utilidad la comprensión de los tipos de tendencia de selección presentados por los diferentes operadores de recombinación.

2.2.5.4 Cruzamiento para enteros y flotantes

Para el caso de los enteros, el cruzamiento se realiza de la misma manera que con las cadenas de bits. Para los números de coma flotante existen la *recombinación discreta,* que genera únicamente, nuevos decimales, teniendo la desventaja de que

en este caso solamente la mutación proporciona nuevos valores a la población; y la *recombinación aritmética o intermedia,* que produce en los genes valores entre los alelos de los genes a combinarse. Si se crea una descendencia *C* a partir de los padres $P_1 = x$ y $P_2 = y$, el valor del alelo para el gen *i* sería: $C_i = \alpha x_i + (1 - \alpha) y_i$, para α en [0, 1]. Este último método sí genera nuevos valores con la desventaja de que el rango de los valores de los alelos para cada gen en la población, se reduce dada su tendencia a promediar (Eiben & Smith, 2003).

2.2.5.5 Cruzamiento aritmético

Se describirán tres tipos de cruzamiento aritmético. En todos se escoge un parámetro α con posibles valores entre [0, 1], sin embargo en la práctica es más común utilizar un valor constante, generalmente 0.5, con lo que se obtiene un cruzamiento aritmético uniforme (Michalewicz, 1996, citado en Eiben & Smith, 2003).

2.2.5.5.1 *Recombinación aritmética simple.*

Se escogen dos padres ($P_1 = x, P_2 = y$) y un punto de cruce y a partir de este, para cada par de genes se obtienen los promedios de los alelos de ambos padres y se colocan en ambos hijos. Los demás valores permanecen igual, de P_1 a C_1 y de P_2 a C_2:

$$C_1 = \langle x_1, \dots, x_k, \alpha \cdot y_{k+1} + (1 - \alpha) \cdot x_{k+1}, \dots, \alpha \cdot y_n + (1 - \alpha) \cdot x_n \rangle$$

para C_2 se utiliza la misma fórmula intercambiando *x* por *y*.

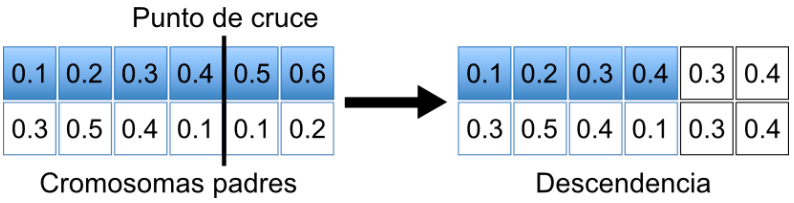

Figura 2.15. Recombinación aritmética simple: k = 4, α = 1/2.

2.2.5.5.2 *Recombinación aritmética de un solo valor.*

Se escoge de la misma manera una posición de cruce y se obtiene el promedio de ambos padres en esa posición. El valor obtenido se asigna a ambos hijos y los demás valores pasan igual:

$$C_1 = \langle x_1, \ldots, x_{k-1}, \alpha \cdot y_k + (1-\alpha) \cdot x_k, x_{k+1} \ldots, x_n \rangle$$

para C_2 se utiliza la misma fórmula intercambiando *x* por *y*.

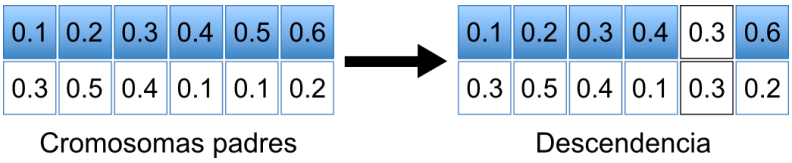

Figura 2.16. Recombinación aritmética de un solo valor: k = 4, α = 1/2.

2.2.5.5.3 *Recombinación aritmética completa.*

Se trata del operador más usado de los tres, en el cual para cada gen, se obtiene el promedio de los alelos de cada padre.

$$C_1 = \langle \alpha \cdot \bar{x} + (1-\alpha) \cdot \bar{y} \qquad C_2 = \alpha \cdot \bar{y} + (1-\alpha) \cdot \bar{x}$$

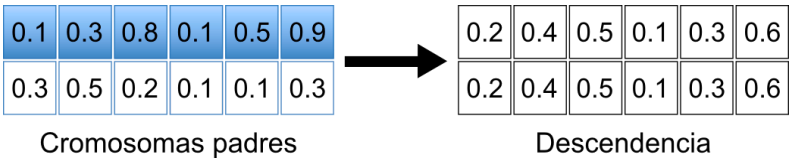

Figura 2.17. Recombinación aritmética completa: α = 1/2.

2.2.5.6 Cruzamiento para representaciones de permutación

Existen diversos mecanismos de selección diseñados para permutación, su objetivo principal es transmitir la mayor cantidad de información posible de los padres hacia su descendencia. En seguida se explican algunos de estos métodos.

2.2.5.6.1 *Cruzamiento en base al orden*

Los algoritmos basados en *k* cruces suelen arrojar soluciones erróneas para problemas con permutación.

En el caso del cruzamiento en base al orden se seleccionan dos padres al azar (P_1 y P_2) y dos puntos de cruzamiento aleatorios. El segmento entre ambos puntos se copia de P_1 al primer hijo (C_1). Posteriormente, comenzando del segundo punto de cruce, se copian los genes de P_2 que no se encuentren presentes actualmente en C_1 en el orden en que aparecen en P_2, volviendo al inicio de C_1 al acabarse la lista (ver Figura 2.18). Para C_2 se realiza un procedimiento similar.

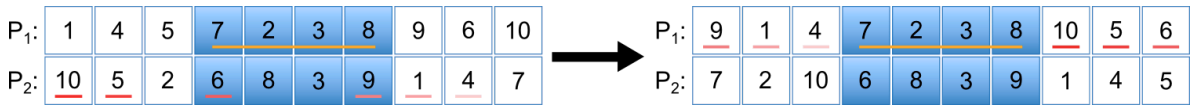

Figura 2.18. Cruzamiento en base al orden.

2.2.5.7 Cruzamiento parcialmente emparejado (PMX)

Este fue inicialmente propuesto por Goldberg y Linge (1985) y es uno de los operadores más utilizados, además de que existen actualmente diversas variaciones de este método.

Primeramente se escogen dos padres y dos puntos de cruce. En seguida, empezando del primer punto hacía el segundo, para cada gen de P_1 en la posición *i* (P_{1i}), éste se intercambia por el gen P_{1j} que contiene el mismo valor que P_{2i}, y ambos valores se colocan en sus respectivas posiciones en C_1. Si el valor de P_{2i}, es decir P_{1j}, ya se encuentra dentro de los dos puntos de cruce en P_1, entonces se coloca el valor de P_{1i} en la posición que contiene el valor de P_{2j}. Se realiza un procedimiento similar para C_2. Las figuras 2.19 a 2.24 ilustran de manera más clara este procedimiento.

Figura 2.19. Cruzamiento PMX.

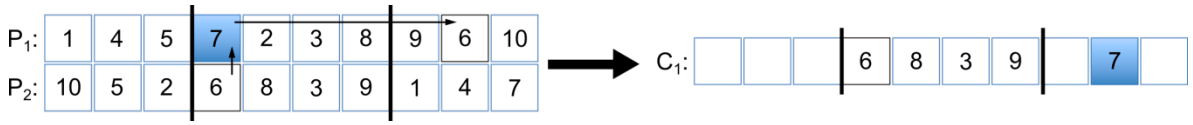

Figura 2.20. Cruzamiento PMX.

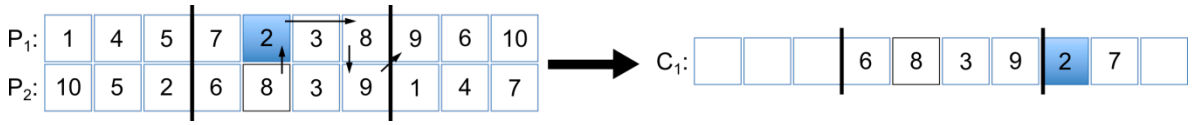

Figura 2.21. Cruzamiento PMX.

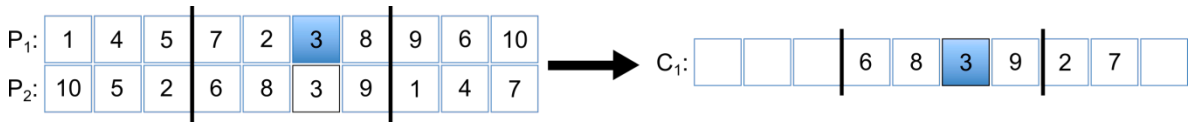

Figura 2.22. Cruzamiento PMX.

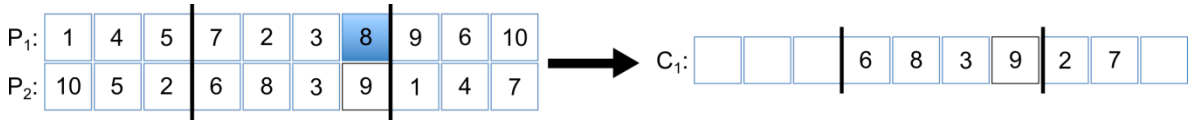

Figura 2.23. Cruzamiento PMX.

Figura 2.24. Cruzamiento PMX.

2.2.5.8 Cruzamiento de ciclo (CX)

El operador CX trata de preservar tanta información como sea posible acerca de la posición absoluta de los elementos.

En este método después de seleccionar ambos padres, se toma el alelo de la primera posición de P_1 y se liga a la misma posición en P_2, después se obtiene de P_1 el gen que contenga el mismo alelo P_2 y se liga al ciclo, el procedimiento se repite hasta regresar a la posición inicial. Finalmente se colocan en C_1 los alelos del primer ciclo de P_1 y los del segundo ciclo de P_2. Los alelos que coincidan en la misma posición tanto en P_1 como en P_2 conservan el mismo lugar en C_1 y C_2 (Ver Figura 2.25).

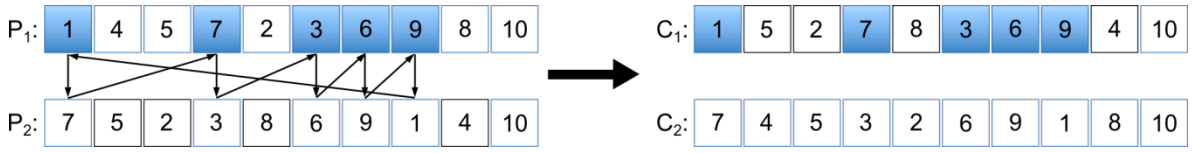

Figura 2.25. Cruzamiento CX.

2.2.6 Mutación

Obtiene una posible solución a partir de la modificación aleatoria de un genotipo. Existen diversas maneras de aplicar este tipo de cambios a los rasgos de la población. En algunas ocasiones con la recombinación se obtienen cada vez mejores cromosomas, sin embargo, puede darse el caso en que el mismo gen en todos los cromosomas tenga el mismo alelo. La mutación enfrenta este problema agregando diversidad a la población para tratar de asegurar que se explora todo el espacio de búsqueda.

Mientras en las estrategias evolutivas, la mutación es el principal operador de búsqueda o variación; en los AG toma un segundo lugar con pequeña probabilidad p_m. Aunque existen más clases de mutación, se describen a continuación las más comunes.

2.2.6.1 Mutación en representaciones binarias

El tipo de mutación más utilizado es el cambio de 0 y 1, aunque es posible el desarrollo de otros operadores específicos para cada problema. El número de genes afectados no es fijo, sino que depende de la longitud del cromosoma y la probabilidad de los genes (ver figura 2.26). En promedio $l \cdot p_m$ valores cambiarán (Eiben & Smith, 2003).

Figura 2.26. Mutación de bits para codificaciones binarias.

2.2.6.2 Mutación para representaciones de enteros

Los siguientes tipos de mutación cambian los valores de cada gen independientemente, a partir de una probabilidad p_m definida por el usuario.

2.2.6.2.1 *Mutación por reajuste aleatorio*

En este caso, con una probabilidad p_m se escoge aleatoriamente un nuevo valor a partir del conjunto de valores permitidos.

2.2.6.2.2 *Mutación de tipo 'creep'*

Se agrega un pequeño valor (positivo o negativo) a cada gen con probabilidad p_m, generalmente estos valores son muestreados aleatoriamente para cada posición y generan únicamente pequeños cambios en la población. (Eiben & Smith, 2003).

2.2.6.3 Mutación para representaciones de punto flotante

Si se tienen valores provenientes de distribuciones continuas, es común cambiarlos aleatoriamente dentro de un dominio con límites inferior y superior $[L_i, U_i] \subseteq \mathbb{R}$, resultando la siguiente transformación:

$$\langle x_1, \dots, x_n \rangle \rightarrow \langle x_1', \dots, x_n' \rangle \text{ donde } x_i, x_i' \in [L_i U_i]$$

Según el tipo de distribución de probabilidad se tienen dos tipos de mutación: uniforme y no uniforme.

2.2.6.3.1 *Mutación uniforme*

Para esta mutación los valores de x_i' son generados aleatoriamente y uniformemente dentro del dominio $[L_i, U_i]$; este tipo de mutación es análoga al cambio de bit para representaciones binarias y reajuste aleatorio para enteros (Eiben & Smith, 2003).

2.2.6.3.2 *Mutación no uniforme con distribución fija*

Generalmente es diseñada para pequeños cambios, agregando al gen un valor tomado aleatoriamente de la distribución Gaussiana (o distribución normal) con valor cero y desviación estándar especificada por el usuario. De ser necesario el resultado deberá ser limitado al rango $[L_i, U_i]$.

2.2.6.4 Operadores de mutación para representaciones de permutación

Contrario a la mutación simple en que un solo gen de la cadena es alterado, en este caso los alelos son movidos a través del genoma. Aquí el parámetro de mutación se interpreta como la probabilidad de que la cadena sea sometida a mutación.

2.2.6.4.1 *Mutación por intercambio*

El operador elige dos elementos aleatoriamente para intercambiar sus posiciones.

Figura 2.27. Mutación por intercambio.

2.2.6.4.2 *Mutación por inserción*

Se escogen dos alelos aleatoriamente y se mueve uno al lado del otro.

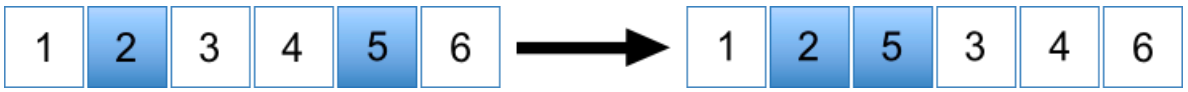

Figura 2.28. Mutación por inserción.

2.2.6.4.3 *Mutación por revolvimiento.*

El genoma completo o alguna parte de este son seleccionados aleatoriamente para cambiar sus posiciones dentro de la cadena.

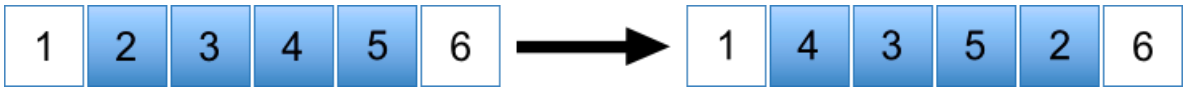

Figura 2.29. Mutación por revolvimiento o (scramble).

2.2.6.4.4 *Mutación por inversión.*

Se seleccionan dos posiciones en la cadena y se invierte el orden de los genes involucrados.

Figura 2.30. Mutación por inversión.

2.2.7 Reemplazo (selección de sobrevivientes).

La descendencia creada a través de los operadores de mutación y recombinación necesita integrarse a la población, de esta manera la memoria de trabajo del AG se reduce de un grupo de μ padres y λ hijos, a *n* individuos que serán parte de la siguiente generación.

2.2.7.1 Reemplazo basado en la edad (Delete-all).

Esta técnica sustituye todos los miembros de la población actual y los reemplaza por el mismo número de cromosomas que han sido creados. Quizá sea este el método más ampliamente utilizado por su sencillez de implementación y ausencia de parámetros.

2.2.7.2 Reemplazo de estado estable (Steady-state).

En este caso existe un parámetro que determina cuantos miembros *n* de la generación *i-1*, serán reemplazados por *n* miembros de la descendencia: generación ***i.*** En este caso, según la configuración del parámetro, quedará determinado el tipo de individuos que se eliminan de la población. Se pueden eliminar individuos en base a su adecuación (la más baja), aunque esto contribuye al rápido mejoramiento de la población y por consecuencia, a una convergencia prematura; es posible también escoger los *n* individuos al azar; eliminar los cromosomas utilizados como padres; etc.

2.2.7.3 Reemplazo de estado estable sin duplicados (Steady-state-no-duplicates).

Es el mismo tipo de reemplazo con la diferencia de que no permite que cromosomas duplicados sean agregados a la población, permitiendo que se explore a mayor profundidad el espacio de búsqueda.

2.2.7.4 Reemplazo con elitismo.

Intenta evitar que se pierda el individuo más apto de la población. De esta manera, si el mejor individuo (o mejores) de la población es elegido para ser reemplazado y ningún individuo nuevo a ser insertado tiene la misma o mejor adecuación que éste, entonces se conserva, quedando fuera alguno de los nuevos individuos candidatos a formar parte de la población.

Habiendo concluido el estudio de lo que a los algoritmos genéticos se refiere, se procederá con la segunda técnica del objeto de estudio. De igual manera, se estudiarán de manera general, los fundamentos biológicos de las redes neuronales, sus características, así como los principales tipos de redes y sus aplicaciones.

2.3 Redes Neuronales

Las redes neuronales artificiales (RNA) son sistemas, hardware o software, de procesamiento de información, que trata de funcionar como lo hace el cerebro copiando esquemáticamente su estructura neuronal (o lo que se conoce de ésta).

Una de las principales ventajas de este tipo de sistemas, es la capacidad de predecir comportamientos de ciertos sistemas naturales. De esta forma se pueden controlar las salidas de un sistema mediante las entradas adecuadas necesarias. Otra ventaja clara es el poder trabajar con un amplio rango de datos que puedan presentarse con ruido: de manera imprecisa y distorsionada.

Las RNA se representan como un modelo simplificado de las redes neuronales biológicas y su funcionamiento se basa en la experiencia y la capacidad de aprender a partir de señales que vengan del exterior, de esta manera les resulta relativamente sencillo adaptarse a ambientes cambiantes, de ahí que sean considerados *sistemas adaptativos*.

2.3.1 Breve reseña histórica

Los antecedentes de las redes neuronales se remontan a los tiempos de Alan Turing y sus ideas en relación al cerebro y la computación. En los años 40 Warren McCulloch y Walter Pitts (McCulloch y Pitts, 1943) realizaron investigaciones sobre la manera en que trabajan las neuronas. En esa misma década, las bases de la teoría de las redes neuronales se llevaron a cabo a partir de los estudios realizados por Donald Hebb quien trató de explicar los procesos de aprendizaje desde un punto de vista psicológico/biológico, donde el aprendizaje se lleva a cabo mediante cambios en una neurona por medio de activaciones.

En los 50s, Frank Rosenblatt comenzó a desarrollar el perceptrón, que es el tipo de red neuronal artificial más sencilla y antigua. A finales de los 60s, MINKSKY y PAPERT (1969) demuestran las limitaciones del perceptrón y quedaron abandonadas las ideas de las RNA que habían surgido en años anteriores. En los 70s con la llegada de los sistemas expertos se estancó rápidamente el progreso de la IA, pero al descubrir que había problemas que los algoritmos secuenciales no podían resolver, como problemas de la vida real, surgieron alternativas como las redes neuronales, la lógica difusa y los algoritmos genéticos principalmente para atacar dichos problemas.

A partir de los 80s resurgieron las redes neuronales de John Hopfield y se retomaron los trabajos iniciales de Paul Werbos sobre el algoritmo de aprendizaje por medio de propagación hacia atrás, originalmente concebido por David Rumelhart y G. Hinton; se desarrolló esta vez el perceptrón multicapa, a partir de ese momento

las redes neuronales han tenido más auge y se ha dedicado más tiempo a la investigación y desarrollo de aplicaciones.

2.3.2 Fundamentos biológicos

Como ya se mencionó, las RNA surgieron inspiradas por los modelos de redes neuronales biológicas, las cuales, como se menciona en Sánchez & Alanís (2006): “no requieren modelos de referencia, se desempeñan de manera adecuada en presencia de incertidumbre; aprenden a realizar nuevas tareas y son altamente adaptables” (p. 1).

De igual forma como se trató en el apartado de los algoritmos genéticos, será importante estudiar de manera breve el fundamento biológico en el que se basan las RNA para poder entender mejor ciertos términos que resultarán útiles, sus analogías y por su puesto la manera en que cada uno de los elementos de desenvuelve dentro de su entorno.

El primero de los términos a revisar es el de una *célula*, que es el elemento básico que conforma a los seres vivos, sus principales componentes pueden apreciarse en la Figura 2.31.

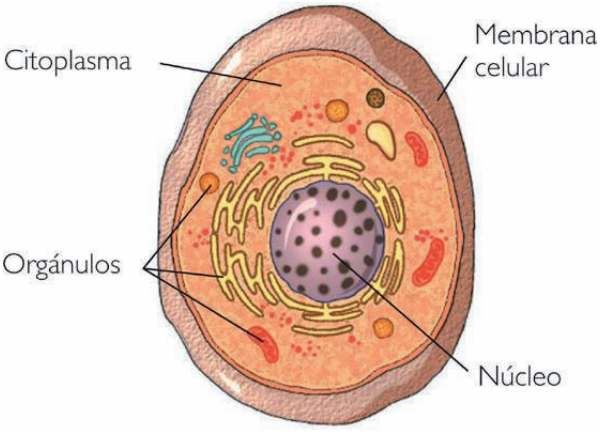

Figura 2.31. Célula animal.

No es necesario estudiar cada una de las partes de la célula, es suficiente con saber que las *neuronas* son una de las diversas formas de células animales que existen y que son el componente básico del sistema nervioso, estimándose que éste cuenta con alrededor de cien mil millones de neuronas que varían de forma y tamaño, aunque en general son similares.

Para poder entenderlo de una manera sencilla, tal como lo ilustra la Figura 2.32, funcionalmente “las neuronas constituyen procesadores de información sencillos. Como todo sistema de este tipo, poseen un canal de entrada de información, *las dendritas*; un órgano de cómputo, *el soma* y un canal de salida, *el axón*” (Del Brío & Sanz, 2002, p. 5).

Detallando un poco los términos recién introducidos, una neurona común cuenta con un número finito de dendritas[5] o ramificaciones procedentes del soma, que también es conocido como el cuerpo celular y que se encarga de la suma de las señales provenientes de las dendritas.

Las dendritas sirven como terminales receptoras de impulsos nerviosos, estos impulsos provienen de un axón perteneciente a otra neurona. Los axones son generalmente ramificaciones muy largas que cuando llegan a su final se ramifican nuevamente, lo que se conoce como *arborización terminal* (Timothy, 2010).

[5] Término proveniente del vocablo griego *"dendro",* que significa árbol.

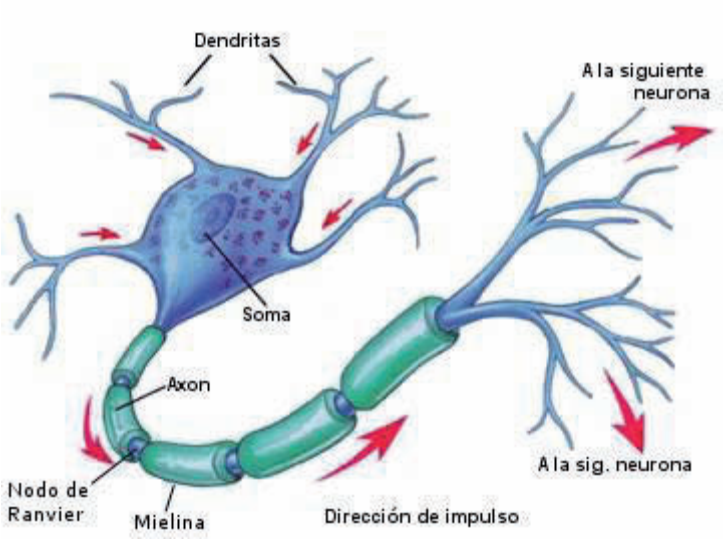

Figura 2.32. Neurona.

Otro término importante es la *sinapsis*, que no es otra cosa más que la unión de dos neuronas. Las sinapsis permiten que una célula influya en la actividad de otras, por medio de la transmisión de las señales provenientes del axón de dicha célula hacia las dendritas o terminales de entrada de alguna otra, posteriormente esta información es procesada por el soma celular.

El tipo de sinapsis más común no establece un contacto físico entre las neuronas, sino que existe una pequeña separación aproximada de 0.2 micas entre la neurona transmisora (*neurona presináptica)* y la neurona receptora (*neurona postsináptica)* (Del Brío & Sanz, 2002).

Las neuronas, como las demás células, están cubiertas por una delgada membrana cuya función es separar el interior del exterior, pues cada parte cuenta con diferentes propiedades tanto químicas como eléctricas.

Las señales nerviosas que se transmiten fuera de la neurona son químicas, mientras por dentro se llevan a cabo las transmisiones eléctricas. La generación de la señal nerviosa está determinada por la membrana neuronal y los iones presentes a ambos lados de ella (Del Brío & Sanz, 2002).

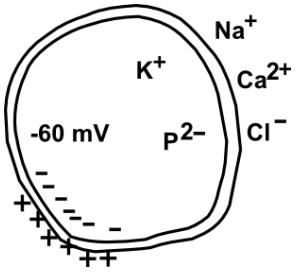

Figura 2.33. Distribución de los iones en la neurona dentro y fuera de la membrana celular (Del Brío & Sanz, 2002).

La membrana genera impulsos eléctricos que viajan a través del axón cubierto por la vaina de Mielina (Figura 2.32), la cual funciona como una capa aislante (Sánchez & Alanís, 2006). Esta información se transfiere a otras neuronas por medio de la sinapsis. El impulso se propaga a lo largo del axón por el cambio de conductancia de la membrana, que contiene canales iónicos selectivos al tipo de ión[6], es decir, que abre y cierra canales a los iones de sodio y potasio (K^+ y Na^+).

En reposo, el interior de la neurona permanece cargado negativamente en relación al medio exterior existiendo una diferencia de potencial entre -60 y -70 mV.

Con un suficiente número de excitaciones se puede elevar el umbral de disparo por encima de los -45mV, provocando una serie de despolarizaciones en los nodos de Ranvier (Ver Figura 2.32), provocando que el potencial cambie de -60/-70mV a un potencial entre +30 y +50mV[7]. Cuando uno de los nodos se despolariza, provoca la despolarización en otro nodo, generando una cadena de reacciones. Así, el potencial viaja de forma discontinua de un nodo a otro a lo largo del axón (transmisión eléctrica). Después de pasar por un cierto punto, el potencial necesita aproximadamente 1ms para volver a ser excitado (Figura 2.34).

[6] Átomo o agrupación de átomos que por su pérdida o ganancia de uno o más electrones adquiere carga eléctrica (Diccionario de la Real Academia Española).

[7] Diferencias de potencial que varían según diversos autores.

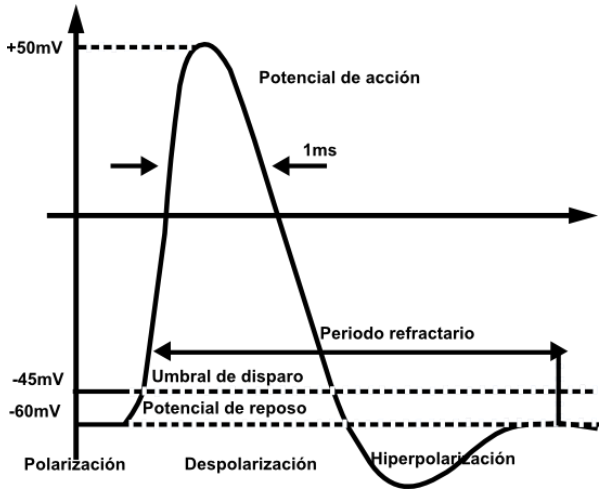

Figura 2.34. Impulso de potencial (Del Brío & Sanz, 2002).

Según las estimulaciones y el nivel de excitación varía el número de pulsos que emiten las neuronas, siendo habitualmente entre 1 y 100 por segundo, aunque algunas neuronas puedan llegar a los 500 durante pequeños periodos de tiempo (Del Brío & Sanz, 2002).

La comunicación entre las neuronas se lleva a cabo cuando el impulso llega al botón sináptico liberando neurotransmisores provenientes de los extremos del axón de la célula presináptica y que serán absorbidos por la célula postsináptica; esto sucede en la hendidura sináptica y se verá reflejado en nuevas excitaciones o inhibiciones en otras neuronas. Así es como se convierte la señal eléctrica de una neurona presináptica a un proceso químico en la sinapsis, para después reconvertirse en señal eléctrica en la neurona postsináptica (Sánchez & Alanís, 2006).

Finalmente, en el material de Del Brío & Sanz (2002) se puede encontrar una clara explicación acerca de la forma de comunicación más habitual entre dos neuronas (transmisión química):

La neurona presináptica libera sustancias químicas complejas denominadas neurotransmisores, que atraviesan el vacío sináptico. Si la neurona postsináptica

posee en las dendritas o en el soma canales sensibles a los neurotransmisores liberados, permitirán el paso de determinados iones a través de la membrana. Las corrientes iónicas que de esta manera se crean provocan pequeños potenciales postsinápticos, excitadores (positivos) o inhibidores (negativos). (p. 7)

Ya estudiados los aspectos fundamentales biológicos de las neuronas, se procede al estudio desde un punto de vista técnico en el cual se mencionarán sus ventajas, una comparativa entre el cerebro humano y las computadoras ordinarias, así como la estructura y características y funcionamiento de los principales modelos de redes neuronales artificiales.

2.3.3 Ventajas de una red neuronal

Algunas de las ventajas que presentan las redes neuronales frente a otros sistemas son:

- **No linealidad.** En este caso el paralelismo permite realizar ciertas tareas de una manera mucho más rápida que un sistema basado en algoritmos secuenciales.
- **Transformación entrada-salida o aprendizaje adaptativo.** El proceso consiste básicamente en presentar a la red un ejemplo ilustrativo y modificar sus pesos sinápticos según la salida obtenida, es decir, aprende a realizar tareas basadas en entrenamiento. La red no necesita un algoritmo sino que ella misma genera su distribución de pesos. Algunas redes continúan "aprendiendo" aún después de su periodo de entrenamiento.
- **Adaptabilidad.** Se les llama a las RNA sistemas dinámicos y auto-adaptativos ya que tienen la capacidad de adaptar sus parámetros inclusive en tiempo real.
- **Auto-organización.** Durante su etapa de aprendizaje, la red puede por sí sola crear su propia organización o representación. Mientras que el aprendizaje se lleva a cabo en cada neurona, la auto-organización toma lugar de manera global, es decir, en la red completa. La auto-organización es importante porque permite a las RN reaccionar de manera adecuada cuando se

presenten datos o situaciones a las que no había estado expuesta; esto es útil cuando se presentan datos incompletos o con ruido.

- **Tolerancia a fallos.** Debido a la interconexión masiva, la falla de un procesador no altera seriamente la operación, aunque la destrucción parcial de la red degrada su estructura y funcionamiento, aún así, ciertas capacidades de la red pueden conservarse. Debido a su redundancia el sistema puede perder rendimiento pero no fallar completamente como un sistema computacional tradicional. En los sistemas tradicionales generalmente la información se guarda en una localidad de memoria que puede sufrir cambios o daños imprevistos, mientras en una RN la información no se encuentra localizada, sino más bien se tienen valores en función de los estímulos recibidos, y se genera una salida a partir de los valores que se tienen.
- **Operación en tiempo real.** Con hardware especializado, los cómputos neuronales pueden realizarse en paralelo; para esto se diseñan y fabrican máquinas con hardware especial con la idea de obtener esta capacidad. Dada la necesidad de procesamiento de datos en forma rápida, las RN son una buena opción gracias a su implementación paralela.
- **Analogía con las redes biológicas.** Esto permite la utilización mutua del conocimiento de las dos áreas.
- **Fácil inserción dentro de la tecnología existente.** Existen herramientas computacionales diferentes a las PC (circuitos integrados especiales), en donde una red puede ser implementada rápidamente a un costo relativamente bajo. Esto permite crear redes para aplicaciones específicas dentro de estas herramientas, mejorando los sistemas de forma incremental.

Ya tratadas las ventajas que presentan las redes neuronales, se puede hacer una comparativa entre el cerebro humano y sus neuronas como pequeños procesadores de información y la manera en que la información es tratada por las computadoras.

2.3.4 *Comparación entre el cerebro y las computadoras ordinarias*

Las máquinas más “inteligentes” construidas por el hombre son capaces de realizar cálculos y razonamiento lógico a velocidades muy altas, lo cual resulta bastante complicado para el cerebro. Sin embargo, existen problemas como el de reconocimiento de patrones, datos de entrada no precisos, etc. en los que la eficiencia de los ordenadores comunes queda muy distante de la del cerebro humano.

Las computadoras cotidianas utilizan la arquitectura von Neumann, que consta de uno o más procesadores (potentes hoy en día), capaces de realizar complejas instrucciones a velocidades sorprendentes; sin embargo, estas instrucciones son ejecutadas de manera secuencial.

Por otro lado, las fibras nerviosas son “malos conductores” y cada neurona es un pequeño procesador sencillo y lento. La diferencia radica en los miles de millones de neuronas o procesadores del cerebro, claramente un número inalcanzable por cualquier máquina construida por el hombre hasta ahora.

Una de las razones de la lentitud de las neuronas se puede entender al comparar la neurona con un extenso cable (pero en mucha menor escala) que a lo largo de muchos kilómetros va presentando pérdidas de la señal debido a la resistencia del mismo cable. En este caso, la resistencia del axón es mayor a la del cobre, lo cual lo vuelve comparativamente en un “peor” conductor. Otra razón es la capacitancia[8] de la membrana celular.

[8] Impedancia ofrecida por un condensador al paso de una corriente eléctrica. (Diccionario RAE 22 ed.)

La tabla 2.3 compara algunas características entre el cerebro humano y una computadora.

Tabla 2.3 Comparación entre el cerebro y una computadora convencional.

	Cerebro	**Computadora**
Velocidad de proceso	≈ 10^{-2} seg. (100Hz)	≈ 10^{-9} seg. (1000Hz)
Estilo de procesamiento	paralelo	secuencial
Número de procesadores	10^{11} - 10^{14}	pocos
Conexiones	≈ 10,000 por neurona	pocas
Almacenamiento del conocimiento	distribuido	direcciones fijas
Tolerancia a fallos	amplia	nula
Tipo de control del proceso	auto-organizado	centralizado

(Del Brío & Sanz, 2002, p. xxx).

2.3.5 Arquitectura de las redes neuronales artificiales

Uno de los elementos claves de las redes neuronales es su estructura. Determinado número de neuronas forman niveles o capas, de los cuales se pueden distinguir tres tipos: las *capas de entrada*, que son las que reciben información del exterior por medio de señales; las *capas ocultas*, que son internas dentro de la red y no tienen contacto con el exterior, pueden existir desde 0 hasta *n* niveles ocultos y dependiendo del número de neuronas así como de su interconexión se obtienen las diferentes topologías de RNA; y las *capas de salida*, que envían información proveniente de la red hacía el exterior.

En la estructura de una red neuronal artificial se pueden distinguir de lo general a lo particular diferentes partes tales como: *el sistema neuronal, la red, la capa y la neurona*, ver Figura 2.35.

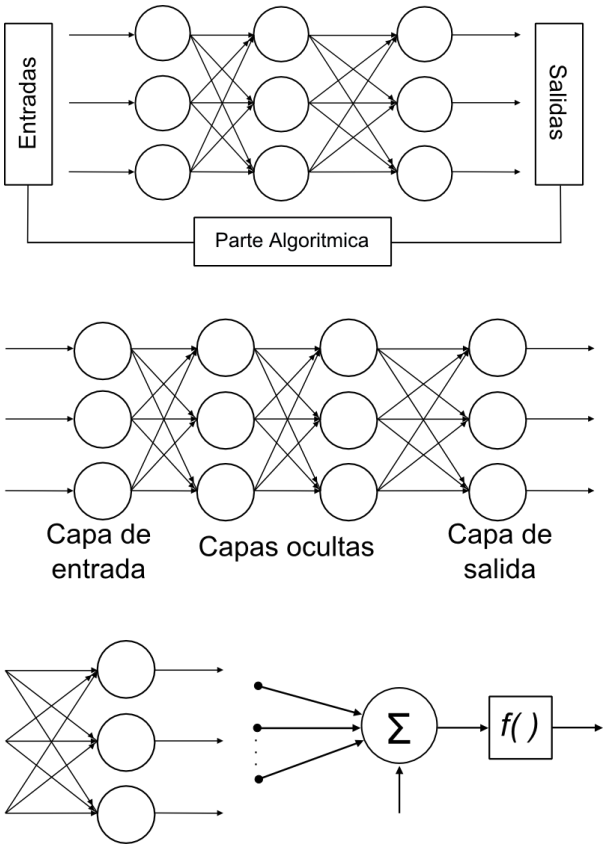

Figura 2.35. Parte superior, sistema neuronal; parte de en medio, red de neuronas; parte de abajo izquierda, capa y parte de abajo derecha, neurona.

Se comenzará en esta parte por analizar el elemento básico de una RNA, que es la *neurona*, definida como “la unidad de proceso de información fundamental en una red neuronal” (Haykin, 1999, citado en Sánchez & Alanís, 2006). La figura 2.36 muestra el modelo básico de neurona de McCulloch-Pitts.

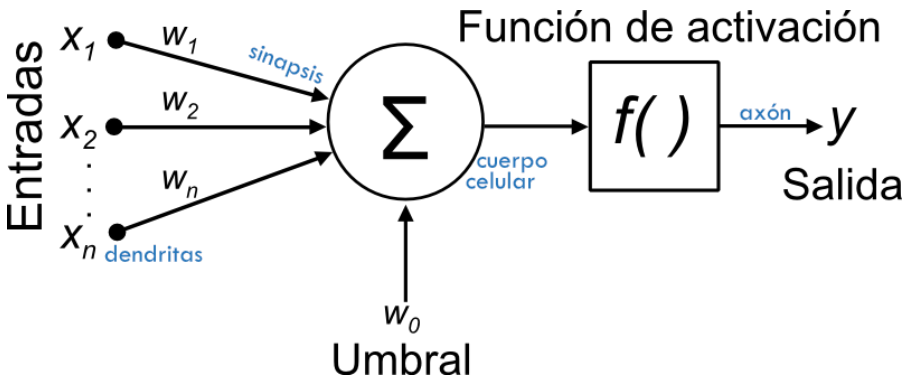

Figura 2.36. Modelo de una neurona artificial.

Donde *x* son las entradas provenientes de otras neuronas, *w* son los pesos sinápticos (intensidad de las señales de entrada), w_0 es el umbral, polarización que se utiliza como compensación a la diferencia de los valores medios de las entradas y las salidas deseadas, el valor de este umbral (w_0) es generalmente igual a uno.

Para calcular la salida, primero se calcula la activación:

$$a = \sum_{i=1}^{n} w_i x_i + w_0$$

que es la suma ponderada de las entradas. Posteriormente aplicando la función de activación se determina *y*:

$$y = f(a) = f\left(\sum_{i=1}^{n} w_i x_i + w_0\right) = f\left(\sum_{i=0}^{n} w_i x_i\right)$$

donde utilizando *w* y *x* como vectores de pesos y entradas respectivamente, se puede expresar la ecuación en notación vectorial:

$$f(w^t x)$$

2.3.6 Funciones de activación

Las neuronas tanto biológicas como artificiales tienen un *estado de activación*, es decir, pueden encontrarse en alguno de estos dos estados: *activas o excitadas* e *inactivas o no excitadas*.

La función de activación define la salida de la neurona en función del potencial de activación *x*, calculando el estado de actividad de una neurona al transformar la entrada global en un valor de activación. El rango usado generalmente va de 0 a 1 o de -1 a 1.

Existen diversas funciones de activación, siendo más comúnmente utilizadas las siguientes:

2.3.6.1 Función escalón

Definida por *a* en las ecuaciones mostradas en la sección anterior, se tiene que

$$f(a)\begin{cases}0\ si\ a<0\\1\ si\ a\geq 0\end{cases}$$

2.3.6.2 Función lineal

Es aquella en la que la *f(a)* = *a*, es decir, la salida es la misma que la entrada.

2.3.6.3 Función lineal a tramos

En este tipo de función se tiene que

$$f(a)\begin{cases}0\ si\ a\leq -\dfrac{1}{2}\\ a\ si+\dfrac{1}{2}>a>-\dfrac{1}{2}\\1\ si\ a\geq +\dfrac{1}{2}\end{cases}$$

2.3.6.4 Función sigmoidea

Es quizá la función más comúnmente utilizada en las redes neuronales y se encuentra dentro de un intervalo cerrado [0, 1] donde el parámetro *g* determina la pendiente de la función, que está definida por la siguiente función logística:

$$f(a) = \frac{1}{1 + e^{-ga}}$$

2.3.6.5 Función tangente hiperbólica

Comprende valores dentro de [-1, 1], donde la pendiente de la función de activación está determinada por *g*.

$$f(a) = \frac{e^{ga} - e^{-ga}}{e^{ga} + e^{-ga}}$$

2.3.6.6 Función gaussiana

$$f(a) = exp\left(-\frac{(a-\mu)^2}{2\sigma^2}\right)$$

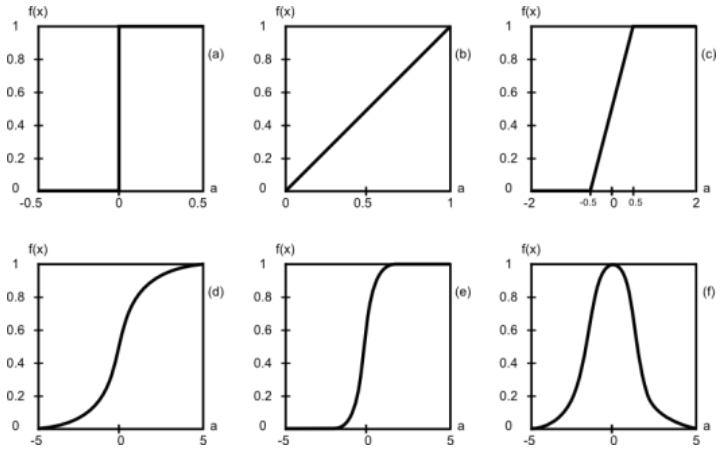

Figura 2.37. Funciones de activación: (a) Función escalón. (b) Función lineal. (c) Función lineal a tramos. (d) Función sigmoidea. (e) Función tangente hiperbólica. (f) Función gaussiana.

2.3.7 Tipos *de* neuronas

De momento se definen dos tipos de neuronas artificiales, clasificándolas en base a los valores que pueden tomar:

2.3.7.1 Neuronas binarias

Toman valores del conjunto {0, 1} ó {-1, 1}.

2.3.7.2 Neuronas reales

Toman valores dentro del rango [0, 1] ó [-1, 1].

2.3.8 Arquitecturas neuronales

La arquitectura también conocida como topología, estructura o patrón de conexionado de la red consiste en la organización y disposición de las neuronas en la red, formando unidades estructurales o agrupaciones (capas) conectadas por medio de las sinapsis, lo que determina el comportamiento de la red. Las neuronas de una capa pueden agruparse, a su vez, formando grupos neuronales (clústeres).

Como ya se mencionó, existen tres tipos de capas: de entrada, de salida y ocultas. La capa de entrada recibe señales provenientes del exterior, las capas ocultas no tienen ningún contacto con el entorno, mientras la capa de salida proporciona la respuesta de la red.

Los parámetros fundamentales de la red son: el número de capas, el número de neuronas por capa, el grado de conectividad y el tipo de conexiones entre neuronas.

Respecto a la estructura de las capas, se tienen *redes monocapa*, compuestas por una sola capa de neuronas y las *redes multicapa*, cuyas neuronas se organizan en varias capas. Cuando todos los nodos de una capa están conectados a todos los nodos de la siguiente capa, entonces se habla de una red *totalmente conectada*, de otra manera, la red se encuentra *parcialmente conectada*.

De acuerdo al flujo de los datos, existen *redes unidireccionales (feedforward)*, donde la información circula en un único sentido, es decir, de las capas de entrada hacia las de salida y las *redes recurrentes (feedback)*, en las cuales la información puede circular en cualquier sentido.

En algunas ocasiones, la operación de una red neuronal se interpreta como la de una memoria asociativa, que ante un determinado patrón de entradas responde con un cierto patrón de salida. Entonces se puede hablar de redes auto-asociativas que ante un patrón A de entrada, responde con un mismo patrón A de salida (por ejemplo A' = A+*ruido*, donde la salida es A, eliminando el ruido de la señal de entrada) y redes hetero-asociativas, que arrojan como salida un patrón B a partir de la entrada de un patrón A (Del Brío & Sanz, 2002).

Las conexiones entre las neuronas pueden ser excitadoras o inhibitorias según el peso sináptico sea positivo o negativo respectivamente. También pueden distinguirse conexiones intra-capa, es decir, entre neuronas de una misma capa; y extra-capa, que se refiere a las conexiones entre neuronas de diferentes capas. Por último existen las conexiones realimentadas, en las cuales la señal fluye en sentido contrario al de entrada-salida, pudiendo existir inclusive realimentación de una neurona consigo misma (Del Brío & Sanz, 2002).

Matemáticamente una red neuronal puede ser definida como "un grafo dirigido, con las siguientes propiedades:

- A cada nodo i se le asocia una variable de estado xi.
- A cada conexión (i, j) de los nodos i y j se asocia un peso wij$\in\Re$.
- A cada nodo i se asocia un umbral θ_i.

Para cada nodo *i* se define una función $fi(x_j, w_{ij}, \theta_i)$, que depende de los pesos de sus conexiones, del umbral y de los estados de los nodos *j* a él conectados. Esta función proporciona el nuevo estado del nodo". (Müller 1990), citado en Del Brío & Sanz, 2002).

2.3.8.1 Redes monocapa

En este caso, en la más simple de las redes neuronales, la capa de entrada se conecta directamente a la capa de salida por medio de la sinapsis. Se denominan monocapa porque no se toma en cuenta la capa de los nodos de entrada. Este tipo de red es utilizada generalmente en tareas relacionadas a la autoasociación, es decir, regeneran información que es presentada de forma incompleta o con ruido.

2.3.8.2 Redes multicapa

Éstas cuentan con una o más capas de neuronas ocultas.

2.3.8.3 Redes recurrentes

Se diferencian de las anteriores por tener al menos un laso de retroalimentación. En una estructura recurrente se permite retener valores para ser usados en el siguiente paso de procesamiento, de esta manera presentan un profundo impacto en la capacidad de aprendizaje y representación de la red neuronal (Sánchez & Alanís, 2006) ya que a diferencia de las redes *feedforward*, que solo transmiten su información hacia las siguientes capas; en este caso, las las neuronas de las redes recurrentes intercambian información en ambos sentidos, dotando a la red con características predictivas.

2.3.9 Mecanismos de aprendizaje

Una red neuronal aprende por medio de la interacción continua con el ambiente y su proceso de autoajuste. El medio que una red emplea para almacenar su conocimiento acerca del problema a resolver es la variación sus pesos sinápticos y el umbral de polarización. Este proceso de aprendizaje también es conocido como entrenamiento, que no es otra cosa más que la modificación de los parámetros de la red.

Para llevar a cabo esta tarea, se hace uso de un *conjunto de entrenamiento* que consiste en utilizar diversos ejemplos. De esta forma no se pretenden memorizar las

relaciones entrada/salida del conjunto de entrenamiento, sino más bien el modelado del proceso que ha generado estos datos (Guijarro Berdiñas, Fontela Romero, & Sánchez Meroño, 2008).

Analógicamente con los sistemas biológicos se puede hablar de la creación modificación y destrucción de conexiones entre las neuronas, esto es, respectivamente, que el peso de cada conexión pase a tener un valor distinto de cero, se modifique o su peso pase a ser cero.

Cuando los valores de los pesos permanecen estables, entonces se puede afirmar que el proceso de aprendizaje ha terminado. Es importante conocer como se modifican los valores de los pesos, los criterios que se siguen para cambiar estos valores asignados.

Puede ser que una red aprenda inclusive durante su funcionamiento, para lo cual se utiliza un término que no será traducido: *online*; mientras que el término *offline*, hace referencia a una "desconexión" de la red una vez terminado el proceso de aprendizaje.

Cuando se lleva a cabo el procedimiento offline se pueden distinguir dos fases que son la de *aprendizaje* o *entrenamiento* y la de *operación* o *funcionamiento*[9].

2.3.9.1 Fase de aprendizaje

El aprendizaje consiste básicamente en el ajuste de los pesos sinápticos, de tal manera que la red realice correctamente el tipo de procesamiento deseado. Una vez que se tienen un modelo de neurona y una arquitectura de red se procede al entrenamiento de ésta, estableciendo en un principio los pesos sinápticos como nulos, o bien inicializados de manera aleatoria (Del Brío & Sanz, 2002).

[9] En otros textos puede encontrarse como el modo de recuerdo o ejecución.

El entrenamiento de la red puede llevarse a cabo en dos niveles, uno incluye la creación y destrucción de neuronas, viéndose reflejado en la modificación de la estructura de la red. El otro nivel, consiste en modificar los pesos sinápticos bajo cierta regla de aprendizaje, la cual puede tratarse de la optimización de una función de error o costo, midiendo así la eficacia de la red.

La adaptación de los pesos es el tipo de aprendizaje en el que se distinguen como eventos del proceso: el estímulo del medio ambiente, el ajuste de los parámetros y la generación de una nueva respuesta.

El mecanismo de ajuste de los pesos de una red por medio de un aprendizaje supervisado (el cual se tratará más adelante) se lleva a cabo a través de la siguiente ecuación (Regla de Hebb, 1949):

$$w_{ij}(t+1) = w_{ij}(t) + \Delta w_{ij}(t)$$

donde w_{ij} es el peso que conecta a la neurona presináptica *j* con la postsináptica *i* en la iteración t y $\Delta w_{ij}(t)$ proporciona la modificación que se deberá incorporar a dicho peso.

El proceso de aprendizaje es usualmente iterativo, en éste se actualizan los pesos sinápticos hasta que la red alcanza el rendimiento deseado (Del Brío & Sanz, 2002).

En función del conjunto de entrenamiento se distinguen dos principales paradigmas de aprendizaje: *supervisado* y *no supervisado*, que serán abordados a continuación.

2.3.9.1.1 *Aprendizaje supervisado*

En este paradigma de aprendizaje se presenta a la red un conjunto de patrones o entradas y la salida deseada con el objetivo de aprender la correspondencia entre ambas. El algoritmo optimizará los parámetros iterativamente hasta que su salida

tienda al objetivo, es decir, se parezca lo más posible al conjunto de entrenamiento, utilizando para ello información detallada del error que comete en cada peso. Dicho de otro modo, trata de encontrar el conjunto de parámetros que minimizan el error de aproximación, es por ello necesario definir una función de error E que en cada iteración indique lo cerca que se está de la solución.

2.3.9.1.2 *Aprendizaje no supervisado*

También es conocido como aprendizaje auto-supervisado o auto-organizado, en este caso no se requiere influencia externa (no recibe información del entorno que indique si la salida generada es o no correcta) para ajustar los presos de las conexiones entre neuronas. Se presentan a la red multitud de patrones sin adjuntar la respuesta que se desea. El objetivo no será encontrar un mapeo de entrada-salida, sino reconocer características, correlaciones, regularidades en el conjunto de entradas, extraer rasgos, o agrupar patrones por categorías según su similitud (clustering).

Estos patrones ocurren con distintas frecuencias y la red deberá encontrar lo que ocurre con más generalidad. Como ejemplos de posibles funciones están el tratar de encontrar una estimación de la distribución de probabilidad *x* y en otros el objetivo será inferir las clases existentes en las que agrupar los ejemplos de entrada que presenten similitudes.

2.3.9.1.3 *Aprendizaje híbrido*

Sucede cuando se hace uso de los dos modelos anteriores, los cuales tienen lugar normalmente en distintas capas de neuronas (Del Brío & Sanz, 2002).

2.3.9.1.4 *Aprendizaje por reforzamiento*

Se basa en no disponer de un ejemplo completo de la salida deseada, sino de una señal de refuerzo que indica si la salida de la red se ajusta a la salida deseada (éxito = 1, fracaso = 0). En ocasiones es conocido también como *aprendizaje por*

premio-castigo y suele ser de uno de los dos tipos siguientes: *no asociativo*, en el cual se selecciona una sola acción óptima y *asociativo,* el cual además de la señal de refuerzo, provee información que permite crear un mapa sobre el estímulo de acción (Sánchez & Alanís, 2006).

Dentro de los paradigmas descritos anteriormente, se utilizan diversos algoritmos entre los que se encuentran principalmente:

- Aprendizaje por corrección de error.
- Regla de Hebb (Delta).
- Retro-propagación.
- Aprendizaje de Boltzman.
- Aprendizaje por refuerzo.
- Aprendizaje estocástico.
- Aprendizaje competitivo.

A continuación se presenta una breve descripción de cada uno de los algoritmos de aprendizaje anteriores.

2.3.9.2 Aprendizaje por corrección de error

Un *error* es la diferencia entre los valores de salida deseados y los obtenidos a partir de la red. En función de esta diferencia es que se ajustan los pesos sinápticos con la intención de disminuir el error y obtener una salida lo más similar a los valores deseados. A partir de la siguiente representación de error:

$$e_i(t) = d_i(t) - y_i(t)$$

donde e_i es el error de la *i*-ésima neurona de la iteración *t,* d_i es la respuesta deseada para la neurona *i* y y_i la respuesta de la neurona; en este caso el objetivo es minimizar una función de costo apoyándose en el criterio de mínimos cuadrados.

Haciendo uso del valor instantáneo del error, se tiene de una solución aproximada:

$$E = \frac{1}{2}\sum_{i=1}^{l} e_i^2(t)$$

finalmente, se obtiene como regla de aprendizaje:

$$\Delta w_{ij}(t) = \sigma e_i(t) x_j(t)$$

donde $0 < \sigma < 1$ es el factor de aprendizaje y $x(t)$ es el vector estímulo del ambiente y $\{ x(t, d_i(t)\}$ el conjunto de entrenamiento.

Como ejemplos pertenecientes a esta clasificación se encuentran el algoritmo de aprendizaje del perceptrón, la regla de aprendizaje delta y la propagación hacia atrás (backpropagation).

2.3.9.3 Aprendizaje por refuerzo

Se trata de un aprendizaje más lento que el anterior, en el cual no se dispone de un ejemplo completo de la salida deseada. En este caso el supervisor indica mediante una señal de refuerzo si la salida obtenida se ajusta a la salida deseada, ajustándose en base a ello los pesos sinápticos.

2.3.9.4 Regla de Hebb

Se trata del primer postulado en 1949 de cómo aprenden las neuronas. El de postulado Hebb (1949, citado en Sánchez & Alanís, 2006) expresa que:

> Cuando el axón de una neurona *A* está lo suficientemente cerca de una neurona *B*, y repetida y persistentemente ayuda a su disparo, se produce un proceso de crecimiento o cambio de metabolismos en una o ambas neuronas, de tal manera

que la eficiencia de *A*, como una de las neuronas que disparan a *B*, se incrementa (p. 24).

En otras palabras, se puede replantear el fundamento de la regla de Hebb diciendo que si dos neuronas N_i y N_j toman el mismo estado simultáneamente (ambas activas o inactivas), el peso sináptico que las une se incrementa; en caso de ser activadas asincrónicamente, el peso se decrementa o desaparece.

La regla de aprendizaje de Hebb puede expresarse en términos matemáticos de la siguiente manera:

$$\Delta w_{ij}(t) = \eta y_i x_j$$

que se refiere a aquella forma de aprendizaje en la cual se modifican los pesos Δw_{ij} proporcionalmente al producto de una entrada *j* por la salida *i* de la neurona, donde x_j y y_i son las señales pre y postsinápticas respectivamente de la neurona en cuestión, mientras que η es el parámetro del ritmo de aprendizaje.

2.3.9.5 Retro-propagación

Basa el aprendizaje de sus pesos en una regla similar a la de corrección de error. El ajuste se realiza comenzando en la capa de salida, propagando el error (calculado mediante una función) hacia las capas anteriores hasta llegar a la capa de entrada. El aprendizaje por retro-propagación cuenta con dos etapas:

2.3.9.5.1 *Hacia adelante*

Se fijan los parámetros de la red (generalmente valores pequeños aleatorios) y se presenta una entrada, ésta se propaga hacia adelante hasta producir la salida.

2.3.9.5.2 *Hacia atrás*

El error obtenido se propaga hacia atrás, los parámetros tienen que ser ajustados con la intención de minimizar el cuadrado de dicho error.

2.3.9.6 Aprendizaje de Boltzmann

Tiene como característica ser un aprendizaje estocástico. Consiste básicamente en realizar cambios aleatorios en los pesos sinápticos y evaluar su efecto a partir del objetivo deseado y ciertas distribuciones de probabilidad.

Las RNA con este tipo de regla de aprendizaje poseen una estructura recurrente de operación binaria (Sánchez & Alanís, 2006):

- Neurona *j* disparada → sj = 1
- Neurona *j* apagada → sj = -1

caracterizadas por la siguiente función de "energía":

$$E = -\frac{1}{2}\sum_{i=1}^{p}\sum_{j=i}^{m} w_{ji} s_j s_i$$

con $i \neq j$; es decir, no hay realimentación.

Su operación se lleva a cabo seleccionando una neurona *j* de forma aleatoria y cambiando su estado de s_j a - s_j, +1 a -1 o vice versa, en función de una probabilidad:

$$P(s_j \rightarrow s_{-j}) = \frac{1}{1 + e^{-\Delta E_j / T}}$$

donde ΔE_j es el incremento en *E*, resultado de tal cambio y *T* el parámetro de pseudo-temperatura.

Las neuronas se dividen en *visibles,* que interactúan con el medio y *ocultas* que operan libremente. Sus modos de operación pueden ser *fijo*, asignándole a las neuronas libres un estado determinado por el ambiente y *libre,* donde todas las neuronas operan libremente (Sánchez & Alanís, 2006).

Se definen también ρ_{ji}^{+} como la correlación entre los estados de las neuronas *i* y *j* condicionada a que la red opere fija y ρ_{ji}^{-} como la correlación entre los estado de las neuronas *i* y *j*.

$$\rho_{ji}^{+} = \sum_{\alpha}\sum_{\beta} P_{\alpha\beta}^{+} S_{j/\alpha\beta} S_{i/\alpha\beta}$$

$$\rho_{ji}^{-} = \sum_{\alpha}\sum_{\beta} P_{\alpha\beta}^{-} S_{j/\alpha\beta} S_{i/\alpha\beta}$$

donde $S_{(\bullet)/\alpha\beta}$ es el estado de la neurona (•) visible en α y oculta en β, $P_{(\bullet)/\alpha\beta}$ es la probabilidad condicional visible en α y oculta en β con la red operando fija en (+) y (-) (Sánchez & Alanís, 2006).

En cuanto a su aprendizaje, se emplea la siguiente función:

$$\Delta w_{ij} = \eta(\rho_{ij}^{+} - \rho_{ij}^{-})$$

que arroja el ajuste Δw_{ij} aplicado al peso sináptico w_{ij} que conecta la neurona *j* con la neurona *i*.

2.3.9.7 Aprendizaje competitivo y comparativo

En este caso las neuronas de la capa de salida compiten entre sí para ser la única ganadora. Resulta ser un esquema útil para descubrir características importantes en el reconocimiento de patrones y clasificación de los datos de entrada (agrupamientos o "clústeres").

Si se determina que cierto patrón nuevo pertenece a una clase previamente reconocida, se lleva a cabo la inclusión de este nuevo patrón. En caso contrario, se reajustarán los pesos para poder reconocer la nueva clase.

A cada neurona se le asigna una cantidad de pesos, generalmente:

$$\sum_{i=1}^{p} w_{ji} = 1 \qquad \forall j = 1 \ldots n$$

utilizando la siguiente regla de aprendizaje (Sánchez & Alanís, 2006):

$$\Delta w_{ji} = \begin{cases} \eta(x - w_{ji}) \text{ si la neurona } j \text{ gana} \\ 0 \text{ si pierde} \end{cases}$$

2.3.10 Modelos de redes neuronales artificiales

Actualmente existe una vasta cantidad de modelos de redes neuronales, algunos más conocidos que otros, algunos que sirven para desempeñar tareas un tanto más generales y algunos otros resultan ser más especializados. Con el tiempo, ciertos modelos clásicos han servido de base para el desarrollo de una infinidad de nuevos sistemas.

La descripción y el análisis de cada uno de estos modelos van más allá del alcance de esta tesis, por lo que únicamente se mencionarán a continuación los modelos más comúnmente utilizados y que han servido como base para muchos otros desarrollos.

2.3.10.1 Perceptrón

Este fue el primer modelo de red de neuronas artificiales introducido por Rosenblatt a finales de los 50, además fue el primer sistema del método de aprendizaje supervisado. Se trata de un modelo construido por una capa de entrada y una capa de salida.

El perceptrón puede utilizarse tanto como clasificador de patrones, como para la representación de funciones booleanas, pues su neurona es esencialmente de tipo MacCulloch-Pitts, de salida binaria. El perceptrón utiliza como regla de aprendizaje una modificación de la regla propuesta por Hebb.

La capa de entrada no realiza ningún computo, solo recibe entradas. La principal aportación del perceptrón es que la adaptación de sus pesos se realiza teniendo en cuenta el error entre la salida que obtiene la red y la salida deseada.

Este modelo sufre de varias limitaciones, entre ellas el hecho de que sus valores de salida solo puedan ser binarios, debido a la función de activación empleada; otro aspecto es que solo pueden clasificar vectores linealmente separables, es decir, zonas de clasificación separadas por una condición lineal o hiperplano. La clase de funciones no separables linealmente no pueden ser representadas por un perceptrón simple, pues se requieren al menos dos líneas para separar las clases.

Otro modelo similar es la ADALINE (ADAptative LInear NEtwork), cuya principal diferencia radica en el algoritmo de aprendizaje utilizado.

2.3.10.2 Perceptrón multicapa

Los problemas presentados por el perceptrón simple al no poder resolver problemas de clasificación no linealmente separables (limitaciones demostradas por Minsky y Papert en 1969), quedaron atrás con el desarrollo del perceptrón multicapa, obra de P. Werbos, 1974.

En su estructura posee al menos una capa oculta y su algoritmo de entrenamiento utilizado es del tipo corrección de error. Se basa en el cálculo del gradiente descendente.

Con este tipo de RNA se han podido resolver diversos y complicados problemas por medio del algoritmo de retro-propagación. Muchas veces la arquitectura de un

perceptrón multicapa (MLP) más un aprendizaje por retro-propagación (BP) suele denominarse *red de retro-propagación o BP*.

El perceptrón multicapa es capaz de presentar mapeos complejos, por lo que puede abordar grandes problemas de clasificación. Varios estudios han demostrado que el perceptrón de una y dos capas resulta ser un excelente aproximador universal de funciones.

2.3.10.3 Redes neuronales de base radial

Fueron desarrolladas en un principio por Powell en 1985, y Broomhead y Lowe en 1988.

Los modelos anteriores producen la salida del producto escalar de un vector de entradas y un vector de peso; en este modelo la activación de las unidades ocultas está determinada por la distancia entre el vector de entrada y el vector de pesos.

Este tipo de redes consta de tres capas: una capa de entrada; una capa oculta, con funciones de activación de base radial (distancia respecto a un punto central) en sus neuronas y la capa de salida con función de activación de tipo lineal. "Su aprendizaje puede ser visto como el ajuste de un conjunto de datos dados por una curva" (Sánchez & Alanís, 2006, p. 87).

Su estructura es de tipo feedforward y los nodos de su capa de entrada son completamente conectados. Su entrenamiento resulta más rápido que el del perceptrón multicapa cuando se cuenta con un mayor número de neuronas y puede utilizar un aprendizaje híbrido (no supervisado en la capa de entrada y supervisado en la capa de salida).

Este tipo de modelo resulta útil para problemas de interpolación, problemas de aproximación, teniendo deficiencias en problemas de extrapolación, para lo cual el perceptrón multicapa responde mejor. También son útiles para problemas de

clasificación, mapeo y poseen capacidades de aproximación universal, control, procesamiento del lenguaje, visión y procesamiento de imágenes, reconocimiento de patrones, estadística, predicción, aplicación en series de tiempo, etc. En comparación con el MLP, este modelo tiene una sola capa oculta, mientras el MLP puede tener más de una.

2.3.10.4 Máquinas de vector soporte

Al igual que el perceptrón multicapa y las redes de base radial, también tienen la capacidad de aproximadores universales. Son útiles para la clasificación de patrones y regresión no lineal.

"La máquina de vector soporte SVM (por sus siglas en inglés) es una implementación aproximada del método de la minimización estructural de riesgos" (Sánchez & Alanís, 2006, p. 107).

Este tipo de redes proporcionan una buena generalización para problemas de clasificación a pesar de no incorporar un dominio del conocimiento del problema. La idea en este caso es construir "modelos fiables" por encima de modelos que cometan "pocos errores".

Entre sus aplicaciones no solo se encuentran los problemas de clasificación binaria, también pueden llevar a cabo tareas de regresión, multi-clasificación, agrupamiento, etc.

2.3.10.5 Mapas auto-organizados

En este tipo de modelo, al no presentarse salidas deseadas, durante el proceso de aprendizaje, la red debe descubrir por sí misma rasgos comunes, regularidades, correlaciones o categorías en los datos de entrada e incorporarlos a sus pesos. En otras palabras estas redes deben auto-organizarse en función de los estímulos requiriendo cierto grado de redundancia (Del Brío & Sanz, 2002).

En general este tipo de redes suelen clasificarse en dos grandes grupos: *redes no supervisadas hebbianas* y *redes no supervisadas competitivas.* En el primer grupo su aprendizaje es de tipo Hebb y varias neuronas de salida pueden ser activadas simultáneamente, mientras en el segundo tipo se llevan a cabo inhibiciones laterales entre las neuronas de salida de tal forma que solo la ganadora permanezca activa; la ganadora en este caso obtiene como premio el refuerzo de sus conexiones sinápticas.

El tipo de procesamiento de una RNA no supervisada depende en gran medida de su arquitectura. Entre sus principales aplicaciones se encuentran el agrupamiento de patrones (clustering), análisis exploratorio, visualización, análisis de componentes principales, memoria asociativa, mapas de riesgos, minería de datos, aproximación funcional, cuantificación vectorial, etc.

2.3.10.5.1 *Modelo de mapas auto-organizados de Kohonen*

Es un sistema utilizado para visualizar e interpretar conjuntos de datos con un espacio de entrada de alta dimensionaldiad. Este sistema convierte las relaciones estadísticas complejas entre los datos de entrada en relaciones geométricas simples en otro espacio de baja dimensión (Guijarro Berdiñas, Fontela Romero, & Sánchez Meroño, 2008).

En este modelo no se actualizan únicamente los pesos de la neurona ganadora, también los pesos del resto de las neuronas, esta es una de las principales diferencias respecto a otros sistemas de aprendizaje competitivos.

Los mapas de Kohonen son útiles en clasificación de patrones, cuantificación vectorial, reducción de dimensiones, extracción de rangos, monitorización de procesos, análisis exploratorio, reconocimiento del habla, visualización y minería de datos (Del Brío & Sanz, 2002).

2.3.10.6 Redes neuronales recurrentes

El tipo de redes pertenecientes a esta clasificación son de gran importancia pues presentan una gran cantidad de arquitecturas y son capaces de llevar a cabo aplicaciones que les resultan imposibles a las redes estáticas, como la predicción no lineal, modelado, control, representaciones en espacio de estado.

Como ya se mencionó anteriormente, las RNA recurrentes cuentan con uno o más lazos de retroalimentación local o global. Su estudio es más complicado que las redes feedforward, que además son más estables y se usan en mayor proporción en aplicaciones prácticas dada su fiabilidad. Las redes recurrentes en ocasiones se estabilizarán después de cierto número de iteraciones, aunque en otros casos la convergencia puede no llegar.

2.3.10.6.1 *Modelo de Hopfield*

Como ejemplo de redes recurrentes, se tiene el modelo de Hopfield que básicamente se trata de una única capa de neuronas totalmente conectada, el modelo es similar a un perceptrón. Las redes de Hopfield son binarias y pueden ser utilizadas además como memoria asociativa.

Algunas de las limitaciones con las que cuenta este modelo es la cantidad de datos que se pueden almacenar y la necesidad de que los datos sean ortogonales entre sí, es decir, lo suficientemente diferentes para que la salida no sea errónea debido a la similitud de las entradas.

Entre las aplicaciones del modelo de Hopfield se pueden mencionar el reconocimiento de imágenes y voz, control de motores, resolución de problemas de optimización, etc.

2.3.10.6.2 *Máquina de Boltzmann*

Cuando el modelo de neurona de Hopfield se interpreta desde un punto de vista probabilístico se obtiene la Máquina de Boltzmann (Del Brío & Sanz, 2002). Este modelo utiliza estados binarios, conexiones bidireccionales, transiciones probabilísticas y puede tener neuronas ocultas. Cada neurona es una unidad estocástica que genera un resultado de acuerdo con la distribución de Boltzmann de la mecánica estadística. Sirven para la clasificación de patrones.

2.3.11 Limitaciones de las RNA

Una de las principales desventajas es la falta de hardware especializado, aunque ya existan dispositivos especiales con funcionamiento en paralelo.

Al utilizar un ordenador común, el tiempo juega un papel fundamental, teniendo como desventaja el hecho de que este tipo de ordenadores son secuenciales y únicamente pueden ejecutar una instrucción a la vez. Además dependiendo de la cantidad de patrones a identificar o clasificar, serán las necesidades de tiempo requeridas.

Otro problema en ocasiones es la falta de reglas definidas que ayuden a la construcción de las redes, pues existen diversidad de algoritmos de aprendizaje, arquitecturas y representaciones de datos posibles que pueden afectar si no se seleccionan y modelan de manera correcta.

Concluida la sección correspondiente a las redes neuronales; como parte final del marco teórico, resta el estudio de la lógica difusa, tratado a continuación.

2.4 Lógica Difusa

Las RN (tal como se estudió antes) emulan de una manera sencilla la estructura del cerebro, intentando reproducir sus capacidades; en cambio, en la lógica difusa (o lógica borrosa) lo que se pretende es emular la manera en que el cerebro razona.

A diferencia de la lógica clásica en la que únicamente se reconocen dos valores de verdad (cierto o falso), la lógica borrosa, permite el trato de información imprecisa o *predicados vagos*, como el decir que una persona es “alta” o “baja”. Dicho de otra manera, la lógica borrosa no maneja valores totalmente ciertos o falsos, sino con determinado grado de verdad o falsedad, lo que permite tratar con modelos del mundo real para los cuales un modelo formal basado en técnicas tradicionales sería muy complejo.

En cuanto a sus aplicaciones, resulta útil su uso en problemas en los cuales no es necesaria precisamente la exactitud; por ejemplo, problemas más “humanos”, como el hecho de estacionar un auto, poder decir si el clima es caluroso, templado o frío, considerar un objeto o alguna acción como buena o mala.

2.4.1 Antecedentes

Históricamente en las ciencias, la incertidumbre se intentaba evitar en medida de lo posible, hasta que comenzó a tomar más auge la teoría de la probabilidad, en la cual se puede encontrar cierto grado de incertidumbre aleatoria.

Para 1965, Lofti Zadeh introdujo ciertas ideas a las cuales llamó *teoría de conjuntos borrosos.*

2.4.2 Utilidad de los sistemas borrosos

En muchas ocasiones se ha demostrado que los sistemas borrosos son aproximadores universales (Kosko, 19994; Ying et al., 1999, citado en Timothy, 2010). Sin embargo, la aproximación de funciones algebraicas no es una de las cualidades principales de un sistema basado en lógica borrosa, más bien su potencial se encuentra en la capacidad de aproximación al comportamiento de

sistemas sin relación numérica o que carecen formulaciones analíticas (sistemas complejos[10]) (Timothy, 2010).

Los sistemas borrosos permiten modelar procesos no lineales y pueden trabajar junto a las redes neuronales y los algoritmos genéticos para aprender de los datos.

Además, gracias a la simplicidad de sus operaciones, resultan ser sistemas relativamente baratos en cuanto al costo y al tiempo empleado en su diseño y realización. Así, cuando se requiera una solución aproximada pero rápida, se puede utilizar un sistema borroso por ejemplo, como estimador inicial para alguna técnica o método más exacto que requiera de una inversión mayor y a largo plazo.

Los sistemas basados en lógica borrosa son útiles también en métodos de control en los que los modelos matemáticos significativos son muy complejos. Estos sistemas de inferencia borrosa consisten en reglas SI-ENTONCES. Los valores ahora dejan de ser exactamente 0 o 1 para pasar a tomar parte de cualquier valor entre este intervalo. Sus principales componentes son las reglas y el razonamiento borroso.

También responden de manera adecuada a aplicaciones como el control automático, clasificación de información, análisis de decisión, sistemas expertos, robótica, reconocimiento de patrones (Jamshidi, 1997, citado en Castillo & Melin, 2008).

[10] Pueden ser sistemas que no han sido probados y que están relacionados con la condición humana, como pueden serlo sistemas biológicos o médicos, sociales, económicos, políticos; en los cuales las entradas y salidas no pueden ser capturadas, analizadas y controladas por completo. Por otro lado, generalmente las causas y los efectos de estos sistemas no son comprendidos, sin embargo, pueden ser observados (Timothy, 2010, traducido por autor).

Una desventaja de estos sistemas es el no poder adaptarse por sí solos a situaciones que cambian, por eso es recomendable, para brindarles adaptabilidad, combinarlos con otras técnicas como RN o AG.

2.4.3 Conjuntos difusos (o borrosos)

Timothy (2010) establece que "los conjuntos borrosos proveen una forma matemática de representar la vaguedad y la impresición en sistemas humanísticos" (p. 13).

En lógica clásica, teniéndo un conjunto de datos X = {1, 2, 3, 4, 5, 6, 7, 8, 9} y para el predicado P = "pares del conjunto X" se tiene N = {2, 4, 6 ,8} es cierto, mientras que I = {1, 3, 5, 7, 9} es falso; la función que asigna estos valores toma el nombre de *función de inclución o pertenencia (membership function, MF)*.

$$\mu_P(x) = \begin{cases} 0, & x \in \mathrm{P} \\ 1, & x \notin \mathrm{P} \end{cases}$$

donde $\mu_P(x)$ denota un miembro ambiguo del elemento *x* en el conjunto P.

En otro ejemplo, teniendo $1.7 \leq x \leq 1.9$ metros, el valor de un individuo x_1 con estatura de 1.8 metros es igual a 1, mientras que el valor de un individuo x_2 con estatura de 1.69 metros es igual a 0. En este sentido, Zadeh extendió tal tipo de conjuntos binarios incorporando valores reales dentro del intervalo [0, 1], obteniendo grados de modo que:

$$\mu_A: U \rightarrow [0,1]$$

donde $\mu_A(x)$ representa el grado en el que el elemento $x \in \mathrm{U}$ pertenece al conjunto borroso A.

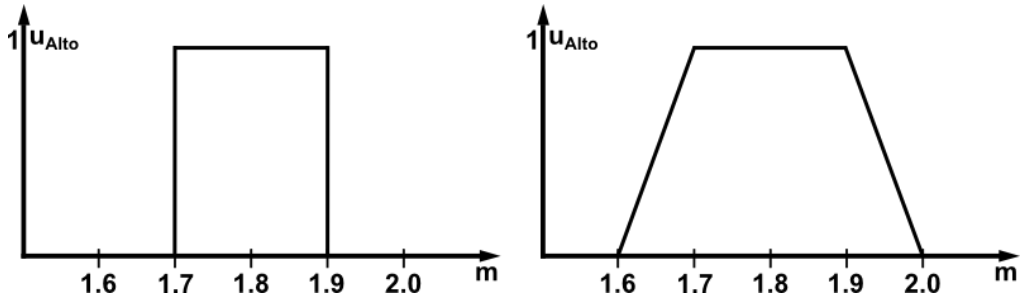

Figura 2.38. Función de inclusión del predicado "alto" en un conjunto clásico (izquierda) y un conjunto borroso (derecha).

"Los elementos en un conjunto difuso, dada su tipo de pertenencia, no necesitan encajar completamente en dicho conjunto, sino que pueden ser miembros también de otros conjuntos difusos dentro del mismo universo" (Timothy, 2010, traducido por autor).

A continuación se presentan brevemente algunos conceptos asociados a una MF tal como se describen en Castillo & Melin (2008):

2.4.3.1 Soporte

El soporte de un conjunto borroso A es el conjunto de todos los puntos *x* en *U* tal que $\mu_A(x) > 0$.

2.4.3.2 Núcleo

El núcleo de un conjunto borroso A es el conjunto de todos los puntos *x* en *U* tal que $\mu_A(x) = 0$.

2.4.3.3 Normalización.

Un conjunto borroso A se encuentra normalizado si su núcleo es no vacío, es decir, que se puede encontrar siempre algún punto *x* en *U* tal que $\mu_A(x) = 1$.

2.4.3.4 Puntos de cruce

Los puntos de cruce de un conjunto borroso V es un punto *x* en *U* en el cual $\mu_A(x) = 0.5$.

2.4.3.5 Singleton

Es un conjunto borroso que soporta un simple punto en *U* con $\mu_A(x) = 1$.

2.4.3.6 Conjunto α-corte

El conjunto α-corte de un conjunto borroso A es el conjunto clásico de todos los puntos *x* en *U* para los que se cumple $\mu_A(x) \geq \alpha$, mientras que para los que se cumple $\mu_A(x) > \alpha$, se les denomina *α-corte fuerte* (p. 7).

Así como existen ciertas operaciones propias de los conjuntos (clásicos), también existen operaciones que pueden llevarse a cabo con los conjuntos difusos. Primeramente será necesario proporcionar una definición de *pertenencia* para conjuntos difusos.

En el texto de Castillo & Melin (2008) se tiene que un conjunto borroso A se encuentra contenido en un conjunto borroso B (A es un subconjunto de B) sí y sólo sí $\mu_A(x) \leq \mu_B(x)$ para toda *x*. Matemáticamente:

$$\boldsymbol{A \subseteq B \Leftrightarrow \mu_A(x) \leq \mu_B(x)}$$

Teniendo los conjuntos borrosos A y B en un universo U. Para un elemento x del universo, se definen

- la unión como $\mu_{A\cup B}(x) = \max(\mu_A(x), \mu_B(x)) = \mu_A(x) \vee \mu_B(x)$,
- la intersección como $\mu_{A\cup B}(x) = \min(\mu_A(x),\ \mu_B(x)) = \mu_A(x) \wedge \mu_B(x)$
- y el complemento o negación como $\mu_{cA}(x) = 1 - \mu_A(x)$;

conocidas como operaciones borrosas estándar.

También los principios de De Morgan pueden ser utilizados en los conjuntos difusos, inclusive todas las demás operaciones aplicables a los conjuntos clásicos pueden ser aplicadas en los conjuntos difusos, a excepción de los axiomas de contradicción y exclusión:

$\overline{A \cap B} = \bar{A} \cup \bar{B}$ y $\overline{A \cup B} = \bar{A} \cap \bar{B}$ - Principios de De Morgan.

$A \cup \bar{A} \neq U$ y $A \cap \bar{A} \neq \emptyset$ - Principios de contradicción y exclusión respectivamente.

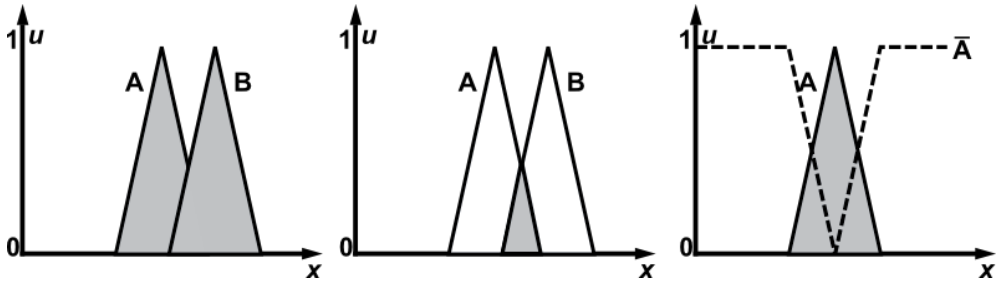

Figura 2.39. Operaciones estándar sobre conjuntos borrosos. De izquierda a derecha: unión, intersección y complemento. (Timothy, 2010, pp. 35-36)

2.4.4 Principales MF en los conjuntos borrosos

Existen diversas funciones de pertenencia comúnmente utilizadas, en seguida se describen de manera breve algunas de ellas.

2.4.4.1 Función triangular

Es definida por tres parámetros {a, b, c} de la siguiente manera:

$$T(x; a, b, c) = \begin{cases} 0, & x \leq a \\ \dfrac{x-a}{b-a}, & a \leq x \leq b \\ \dfrac{c-x}{c-b}, & b \leq x \leq c \\ 0, & c \leq x \end{cases}$$

2.4.4.2 Función trapezoidal

Está definida por cuatro parámetros {a, b, c, d} de la siguiente manera:

$$S(x;a,b,c,d) = \begin{cases} 0, & x \le a \\ \dfrac{x-a}{b-a}, & a \le x \le b \\ 1, & b \le x \le c \\ \dfrac{d-x}{d-c}, & c \le x \le d \\ 0, & d \le x \end{cases}$$

Esta función se utiliza generalmente en sistemas borrosos sencillos, pues permite definir conjuntos con pocos datos.

Dada la simplicidad en sus fórmulas y eficiencia, las dos funciones son ampliamente empleadas, sobre todo en implementaciones en tiempo real (Castillo & Melin, 2008).

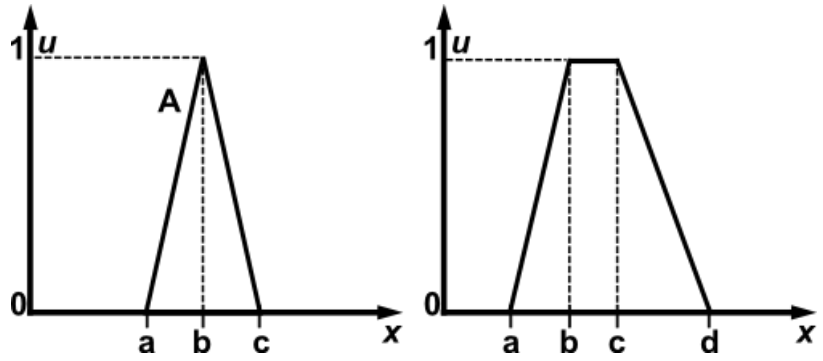

Figura 2.40. Funciones triangular y trapezoidal.

2.4.4.3 Función sigmoidea

La función sigmoidea se define por los parámetros {a, b, c} como se muestra a continuación:

$$S(x;a,b,c)=\begin{cases}0, & x\le a\\ 2\left(\dfrac{x-a}{c-a}\right)^2, & a\le x\le b\\ 1-2\left(\dfrac{x-a}{c-a}\right)^2, & b\le x\le c\\ 1, & x\ge c\end{cases}$$

siendo su cruce igual a 0.5 y $b=(a+c)/2$ para esta función se utiliza también la siguiente ecuación:

$$S(x;a,c)=\frac{1}{1+\exp\left[-a(x-c)\right]}$$

2.4.4.4 Función gaussiana

La función gaussiana es definida por dos parámetros {c, σ} de la siguiente manera:

$$G(x;c,\sigma)=e^{-\frac{1}{2}\left(\frac{x-c}{\sigma}\right)^2}$$

donde *c* representa el centro y σ determina el ancho. Otra forma de definirse esta función es por medio de la ecuación siguiente:

$$G(x;b,c)=\begin{cases}S\left(x;c-b,c-\dfrac{b}{2},c\right) & x\le c\\ 1-S\left(x;c-b,c-\dfrac{b}{2},c\right) & x\ge c\end{cases}$$

2.4.4.5 Función campana generalizada

Es definida por tres parámetros {a, b, c} de la siguiente manera:

$$B(x;a,b,c)=\frac{1}{1+|(x-c)/a|^{2b}}$$

La Figura 2.41 muestra los tres tipos de funciones recién introducidos.

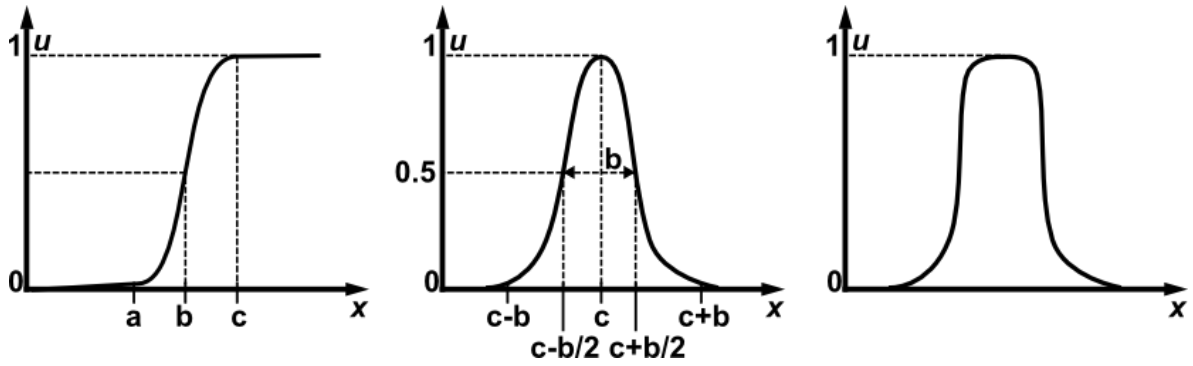

Figura 2.41. Funciones sigmoidea, gaussiana y de “campana generalizada”.

2.4.5 Variable lingüística

La noción de variable lingüística fue introducida por Zadeh en 1975 refiriéndose a aquella variable V que puede tomar por valor términos del lenguaje natural que realizan una descripción cualitativa de un conjunto de referencia U, por ejemplo, sea una variable V que toma valores de un conjunto $L = \{L_1, L_2,..., L_n\}$, si U es un conjunto de temperaturas, se puede considerar L_T = {frío, templado, caliente}.

Formalmente una variable lingüística es una tupla (V, T(V), U, G, M), donde V es el nombre de la variable; T(V) es el conjunto de términos que nombran los *valores* o *términos lingüísticos* “*x*” que puede tomar V, estos valores con conjuntos borrosos en U, que es el universo del discurso; G es una regla sintáctica que genera los términos en T(V); finalmente, M es una regla semántica que asocia un significado a cada valor.

Dentro de una variable V se pueden llevar a cabo diversas particiones. La partición es un subconjunto de los conjuntos borrosos definidos para la variable V. En este sentido, para la variable “temperatura”, la Figura 2.42 muestra un ejemplo de participación borrosa.

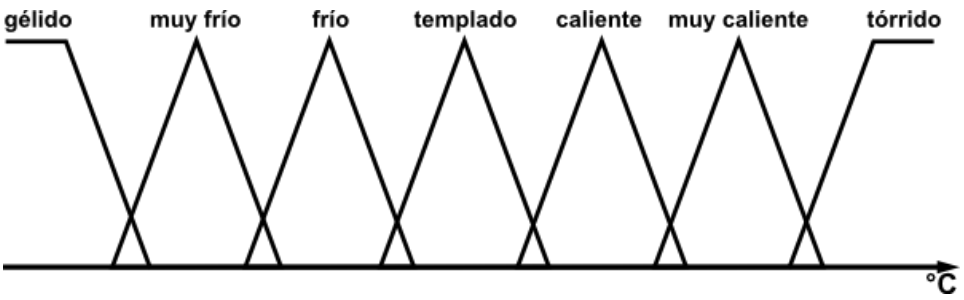

Figura 2.42. Ejemplo de partición borrosa en el conjunto de las temperaturas (Félix Lamas & Bugarín Diz, p. 274)

2.4.6 Inferencia borrosa

Al igual que en la lógica clásica, la borrosa trata el razonamiento formal con proposiciones, teniendo que estas proposiciones pueden tomar valores intermedios entre verdadero y falso.

2.4.6.1 Principio de extensión

El principio de extensión de Zadeh, es un concepto básico en la teoría de conjuntos difusos que permite convertir conceptos no borrosos en borrosos, realizando un mapeo *uno a uno*. Se trata de la base de inferencia de los sistemas borrosos. Por ejemplo, sean U y V dos universos de discurso y *f* una función de U a V. En general, para un conjunto borroso A en U, el principio de extensión define un conjunto borroso B en V dado por:

$$\mu_B(v) = sup_{u \in f^{-1}(v)}[\mu_A(u)]$$

donde $\mu_B(v)$ es el máximo de $\mu_A(u)$ para todos los $u \in U$ que cumplen que $f(u) = v$, donde $v \in V$ y suponemos que $f^{-1}(v)$ no es vacío. Si $f^{-1}(v)$ es vacío para algún $v \in V$ definiremos $\mu_B(v) = 0$ (Del Brío & Sanz, 2002).

2.4.6.2 Relaciones borrosas

Las relaciones borrosas permiten una pertenencia parcial describiendo la relación entre dos o más objetos. Estas relaciones son un subconjunto borroso del producto cartesiano entre dos conjuntos.

Siendo $\mu_{A_1}(x)$, $\mu_{A_2}(x)$,..., $\mu_{A_n}(x)$ funciones de pertenencia de A_1, A_2,...,A_n su producto cartesiano se define como la función de pertenencia siguiente:

$$\mu_{A_1 \text{ x } A_2 \text{ x } \dots \text{ x } A_n}(x_1, x_2, \dots, x_n) = min\left[\mu_{A_1}(x_1), \mu_{A_2}(x_2) \dots, \mu_{A_n}(x_n)\right]$$

así, teniendo A={(2, 0.4), (4, 2), (8, 0.6)} y B={(3, 2), (0.5, 2)} el resultado de A x B sería igual a:

A x B = {[(2, 3), min(0.4, 2)], [(4, 3), min(2, 2)], [(8, 3), min(0.6, 2)], [(2, 0.5), min(0.4, 2)], [(4, 0.5), min(2, 2)] , [(8, 0.5), min(0.6, 2)]}

A x B = {[(2, 3), 0.4], [(4, 3), 2], [(8, 3), 0.6], [(2, 0.5), 0.4], [(4, 0.5), 2] , [(8, 0.5), 0.6]}

De esta manera, las relaciones borrosas son un mapeo de los elementos de dos universos a través de su producto cartesiano. Teniendo dos conjuntos A, B y mediante una función de pertenencia una relación borrosa R, ésta es un mapeo del espacio cartesiano X x Y en el intervalo [0, 1]:

$$\mu_R(x, y) = \mu_{A \text{ x } B}(x, y) = \min\left(\mu_A(x), \mu_B(y)\right)$$

$$\mu_R : A \text{ x } B \rightarrow [0,1]$$

$$R = \{((x, y), \mu_R(x, y)) | \mu_R(x, y) \geq 0, x \in A, y \in B\}$$

A manera de ejemplo, teniendo los siguientes conjuntos difusos A y B, definidos en un universo de temperaturas $\mathrm{X} = \{x_1, x_2, x_3\}$ y un universo de presiones $\mathrm{Y} = \{y_1, y_2\}$ respectivamente, se obtiene como producto cartesiano:

$$\mu_A(x) = 0.3, 0.5, 1 \text{ y } \mu_B(y) = 0.4, 0.7$$

$$\mu_R(x,y) = \begin{bmatrix} 0.3 & 0.3 \\ 0.4 & 0.5 \\ 0.4 & 0.7 \end{bmatrix}$$

Por otro lado, teniendo X = {1, 3, 6}, Y = {1, 2, 5, 7} y R = "*y* es un poco más grande que *x*". La función de pertenencia de la relación difusa R puede ser expresada (por ejemplo) como se muestra a continuación:

$$\mu_R(x,y) = \begin{cases} \dfrac{y-x}{y+x}, & si\ y > x \\ 0, & si\ y \leq x \end{cases}$$

lo que daría como resultado:

$$\mu_R(x,y) = \begin{bmatrix} 0 & 0.3 & 0.6 & 0.75 \\ 0 & 0 & 0.25 & 0.4 \\ 0 & 0 & 0 & 0.07 \end{bmatrix}$$

Por medio de las relaciones borrosas se pueden obtener medidas como la *borrosidad*, grado que mide la distancia existente entre un conjunto borroso y un conjunto discreto; la *distancia entre dos conjuntos borrosos; la similitud*, que debe contener como propiedades la *reflexividad, simetría y transitividad* para poder indicar el parecido entre dos conjuntos, etc. De estas posibles mediciones se pueden obtener sentencias como "es mucho más frío que", "es algo similar a", "es mucho menor que", etcétera.

2.4.6.3 Reglas borrosas

Las reglas borrosas combinan uno o más conjuntos de entrada llamados *antecedentes* o *premisas*, asociándolos a un conjunto borroso de salida. También conocidas como *implicaciones borrosas* o *sentencias condicionales borrosas*, son generalmente del tipo SI-ENTONCES

si *x* es A entonces *y* es B

donde "*x* es A" es la premisa y "*y* es B" es la *consecuencia* o *conclusión*, en algunas ocasiones abreviadas como A → B y que generalmente denota implicación, tal como en la lógica clásica.

La regla básica de inferencia es el *modus ponens*, en el cual se puede inferir la verdad de la preposición B a partir de la verdad de la preposición A. El concepto anterior se ilustra en la Tabla 2.4.

Tabla 2.4 Implicación borrosa

	A → B
Premisa 1 (hecho)	x es A
Premisa 2 (regla)	si *x* es A entonces *y* es B
Conclusión	y es B

Hay ocasiones en que el modus ponens es utilizado de manera aproximada, teniendo que A' es similar a A y B' a B. En esta caso el procedimiento se llama *razonamiento aproximado* o *razonamiento borroso*, dado el modus ponens como caso especial, se le suele llamar también *modus ponens generalizado* (GMP Generalized Modus Ponens). Lo anterior puede verse con más claridad en la Tabla 2.5.

Tabla 2.5 Modus Ponens Generalizado

	A' → B'
premisa 1 (hecho)	x es A'
Premisa 2 (regla)	si *x* es A entonces *y* es B
conclusión	y es B'

Las reglas borrosas permiten denotar conocimiento a partir de reglas base que expresan las relaciones conocidas entre antecedentes y consecuentes. Estas reglas

pueden representarse como una tabla de las reglas que la forman, o como una memoria asociativa borrosa (FAM, Fuzzy Associative Memory), siendo éstas, matrices que representan la consecuencia de cada regla para cada combinación de dos entradas (Del Brío & Sanz, 2002).

2.4.7 Dispositivos de inferencia borrosa

Se les llama así a los sistemas que interpretan las reglas del tipo SI-ENTONCES, con la finalidad de obtener una salida a partir de las variables lingüísticas de entrada de un sistema.

2.4.7.1 Borrosificador (Fuzzifier)

Puede definirse la b*orrosificación* (fuzzification) como el proceso de convertir datos de entrada no borrosos a datos borrosos. Así, un *borrosificador* es aquel que establece una relación entre puntos de entrada no borrosos al sistema y sus correspondientes conjuntos borrosos.

Acorde con Del Brío & Sanz (2002) se pueden utilizar diversas estratégias de borrosificación, tales como:

2.4.7.1.1 *Borrosificador "singleton"*

Es el método de borrosificación más utilizado, principalmente en sistemas de control; consiste en considerar los propios valores discretos como conjuntos borrosos. De otra forma, para cada valor de entrada **x** se define un conjunto A' que lo soporta, con función de pertenencia $\mu_A(\mathbf{x}')$, de modo que $\mu_A(\mathbf{x}) = 1, (\mathbf{x}' = \mathbf{x})$ y $\mu_A(\mathbf{x}') = 0$, para todos los otros $\mathbf{x}' \in U$ en los que $\mathbf{x}' \neq \mathbf{x}$.

2.4.7.1.2 *Borrosificador "no singleton"*

En este médoto de borrosificación se utiliza una función exponencial del tipo siguiente

$$\mu_{A'}(x') = a.exp\left[-\left(\frac{x'-x}{\sigma}\right)^2\right]$$

función con forma de campana, centrada en el valor de *x* de entrada, de anchura σ y amplitud *a* (p. 264).

2.4.7.2 Desborrosificador (defuzzifier)

El desborrosificador se refiere a una función que transforma un conjunto borroso en uno no borroso. Entre los diferentes desborrosificadores existentes destacan los que se muestran a continuación:

2.4.7.2.1 *Desborrosificador por máximo*

A este tipo de desborrosificador corresponde la siguiente ecuación

$$y = argsup_{y\in V}(\mu_{B'}(y))$$

donde *y* es el punto en V en que $\mu_{G^l}(y)$ alcanza su valor máximo y donde $\mu_{B'}(y)$ está definido por la unión de los M conjuntos borrosos B^l, teniendo cada *l* como una regla de la base de reglas.

$$\mu_{B'}(y) = \mu_{B^1}(y) + \cdots + \mu_{B^m}(y)$$
$$\mu_{B^l}(y) = sup_{\mathrm{x}\in U}\left[\mu_{F_1^l \mathrm{x}\ldots\mathrm{x}\, F_n^l \to G'}(\mathbf{x}, y) * \mu_{A'}(\mathbf{x})\right]$$

2.4.7.2.2 *Desborrosificador por centro de área*

Este tipo de desborrosificador está definido por la ecuación:

$$\boldsymbol{y = \frac{\sum_{l=1}^{M} M^l\left(\mu_{B'}(y^{-l})\right)}{\sum_{l=1}^{M} A^l(\mu_{B'}(y^{-l}))} = \frac{\sum_{l=1}^{M}\int_v \mu_{B'}(y^{-l})^2 dy M^l\left(\mu_{B'}(y^{-l})\right)}{\sum_{l=1}^{M}\int_v \mu_{B'}(y^{-l})dy}}$$

donde $\boldsymbol{M^l}$ es el momento (en torno al eje y del universo de discurso de la salida V) de la función de inclusión del conjunto borroso $\boldsymbol{G^l}$ y $\boldsymbol{A^l}$ es el área.

2.4.7.2.3 *Desborrosificador por media de centros*

Definido como:

$$\boldsymbol{y} = \frac{\sum_{l=1}^{M} \boldsymbol{y^{-l}} \left(\boldsymbol{\mu_{B'}}(\boldsymbol{y^{-l}}) \right)}{\sum_{l=1}^{M} \left(\boldsymbol{\mu_{B'}}(\boldsymbol{y^{-l}}) \right)}$$

donde $\boldsymbol{y^{-l}}$ representa el centro del conjunto borroso $\boldsymbol{G^l}$.

2.4.8 Modelado de sistemas borrosos

En general, los sistemas de inferencia borrosa se diseñan basados en el conocimiento que se tiene sobre el comportamiento del sistema objetivo. Como ejemplo, un sistema de inferencia borrosa podría convertirse en el operador que regule y controle algún proceso industrial que hasta antes de su implementación se llevaba a cabo por un operador humano.

De acuerdo con Castillo & Melin (2008), el proceso de modelado generalmente se compone de las siguientes características:

- La estructura de reglas de un sistema de inferencia borrosa hace que sea fácil de incorporar la experiencia humana sobre el sistema objetivo directamente en el proceso de modelado.
- El modelado borroso aprovecha el "conocimiento del dominio" que no puede ser fácil o directamente empleado en otros enfoques de modelado.
- Cuando los datos de entrada-salida de un sistema objetivo están disponibles, se pueden utilizar técnicas convencionales de identificación de sistemas para el modelado difuso. En otras palabras, el uso de "datos numéricos" también juega un papel importante en el "modelado difuso", al igual que en otros métodos de modelado matemático" (traducido por autor, p. 27).

Castillo & Melin (2008) también expone que el modelado difuso puede llevarse a cabo en dos fases. Primeramente la "estructura de la superficie", que incluye las siguientes tareas:

- La selección de variables relevantes de entrada y salida.
- Escoger el tipo específico de sistema de inferencia borrosa.
- Determinar el número de términos lingüísticos asociados a cada variable de entrada y salida.
- El diseño de una colección de reglas borrosas SI-ENTONCES.

Llevándose a cabo en base al propio conocimiento, tal como el sentido común, leyes físicas, etc., conocimiento proporcionado por expertos humanos y/o prueba y error.

Al concluir la primera fase se obtienen reglas que describen el comportamiento del sistema objetivo por medio de términos lingüísticos. El significado de éstos últimos se determina en la segunda fase, la identificación de una "estructura profunda", que determina la función de pertenencia de cada término lingüístico. Este proceso se ejecuta por medio de las siguientes acciones:

- Se escoge una familia apropiada de MFs.
- Se entrevista a expertos humanos en el tema para determinar los parámetros de las MFs utilizadas en la regla base.
- Se refinan los parámetros de las MFs utilizando técnicas de regresión y optimización.

En este caso, los dos primeros puntos asumen la disponibilidad de expertos humanos, mientras que el tercer punto asume la disponibilidad de un conjunto de datos de entrada-salida deseados (p.28).

Habiendo expuesto lo anterior, queda mencionar que los sistemas de inferencia borrosa (FIS, Fuzzy Inference Systems) o controladores borrosos (FLC, Fuzzy Logic

Controllers), son sistemas expertos de control borroso basados en reglas. Son según Del Brío & Sanz (2002) la aplicación más extendida de la lógica difusa.

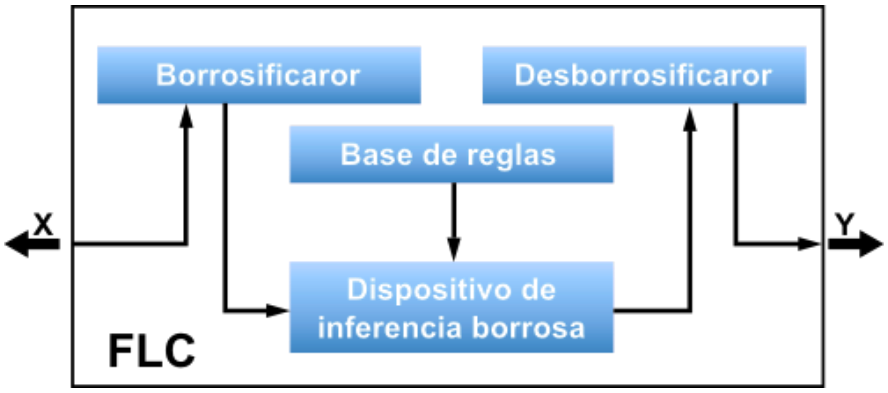

Figura 2.43. Estructura de un controlador borroso o FLC (Del Brío & Sanz, 2002).

A través del tiempo, estos sistemas han pasado de ser estáticos, a sistemas que pueden optimizar su funcionamiento por medio de entrenamiento, haciendo uso, por ejemplo, de sistemas neuronales (sistemas neuro-borrosos) o algoritmos genéticos.

Entre el vasto número de algoritmos de aprendizaje que se pueden aplicar a los sistemas borrosos, destacan los siguientes:

- Retropropagación.
- Algoritmos genéticos.
- Mínimos cuadrados ortogonales.
- Tablas de búsqueda.
- Agrupamiento (clustering) por vecino más cercano.

En general, la mayoría de los métodos de aprendizaje utilizados en las redes neuronales pueden ser aplicables a este tipo de sistemas, pudiendo obtener sistemas neuro-borrosos que aprovechen la capacidad de aprendizaje de las redes neuronales y el trato de información incierta y su traducción a reglas de los sistemas borrosos.

2.4.9 Aplicaciones y conclusiones

Entre las diversas aplicaciones que se pueden dar a los sistemas basados en lógica borra se encuentran:

- Control industrial.
- Aplicaciones en áreas médicas, tales como:
  - diagnósticos,
  - análisis de ritmos cardiacos,
  - etc.
- Apoyo a la toma de decisiones.
- Asesoramientos financieros y de inversión.
- Procesamiento de imágenes.
- Reconocimiento de caracteres.

La lógica borrosa es un paradigma cada vez más estudiado debido a su capacidad de resolver problemas con alto grado de incertidumbre, además del trato con grandes cantidades de datos y la posible incorporación de conocimiento.

Este tipo de sistemas puede funcionar como una extensión de sistemas expertos, disminuyendo su costo y yendo más allá de las limitaciones de éstos, al tratar con datos rodeados de incertidumbre.

Una ventaja importante de la lógica borrosa radica en la simplicidad de los cálculos requeridos (sumas, comparaciones), lo que se ve reflejado en sistemas rápidos y económicos, pero su verdadero potencial se encuentra en la combinación con otros paradigmas como los AG y las RN, aprovechando sus capacidades de aproximación, aprendizaje, logrando construir así sistemas complejos que traten bien con cantidades considerables de información en las que los datos vengan acompañados de ruido y redundancia, obteniendo mejores resultados.

3 MARCO METODOLÓGICO

3.1 Introducción

En este capítulo se presenta la metodología empleada para el desarrollo de los algoritmos antes mencionados. Se muestran aspectos como el uso de marcos de trabajo y el por qué de su elección, así como la de los algoritmos presentados.

3.2 Metodología

La metodología bajo la cual es desarrollado el presente proyecto es ***cuantitativa*** dado que se va a trabajar con datos numéricos, se van a realizar experimentos bajo las diferentes técnicas ya mencionadas, para poder realizar una adecuada comparación entre éstas mediante los datos arrojados por cada uno de los algoritmos que se van a desarrollar.

3.3 Diseño de la investigación

En función de los objetivos definidos en el presente estudio, se decidió adecuar éste a los propósitos de la investigación experimental.

3.4 Procedimientos

Para obtener una aproximación al óptimo del problema del agente viajero se desarrollaron dos programas en el lenguaje Java. Las quince ciudades entre las que se pretende obtener una ruta óptima se muestran en la Tabla 3.1, proporcionando la distancia en kilómetros entre cada una de las ciudades, en línea recta.

Tabla 3.1 Tabla de ciudades y distancias utilizadas

	Mlm	**DF**	**Gto**	**Qro**	**Gdl**	**Ags**	**Mty**	**Cun**	**Pue**	**SLP**	**Zac**	**Chi**	**Mzt**	**Cabos**	**Tij**
Mlm	0	314	180	190	293	323	890	1896	425	398	444	1269	762	*1391*	2518
DF	314	0	348	218	543	510	910	1608	137	418	603	1428	1012	1641	2768
Gto	180	348	0	124	296	196	700	1918	447	208	317	1142	765	1394	2522
Qro	190	218	124	0	364	294	703	1808	337	211	396	1221	833	1462	2598
Gdl	293	543	296	364	0	216	765	2141	670	330	316	1116	475	1104	2232
Ags	323	510	196	294	216	0	573	2101	630	166	120	945	690	1319	2356
Mty	890	910	700	703	765	573	0	2298	1032	515	463	801	882	1511	2355
Cun	1896	1608	1918	1808	2141	2101	2298	0	1495	2016	2201	3027	2623	3252	4379
Pue	425	137	447	337	670	630	1032	1495	0	531	716	1541	1137	1766	2894
SLP	398	418	208	211	330	166	515	2016	531	0	190	1015	802	1431	2427
Zac	444	603	317	396	316	120	463	2201	716	190	0	831	595	1224	2242
Chi	1269	1428	1142	1221	1116	945	801	3027	1541	1015	831	0	908	1259	1409
Mzt	762	1012	765	833	475	690	882	2623	1137	802	595	908	0	634	1759
Cabos	1391	1641	1394	1462	1104	1319	1511	3252	1766	1431	1224	1259	634	0	1631
Tij	2518	2768	2522	2598	2232	2356	2355	4379	2894	2427	2242	1409	1759	1631	0

El primero de los programas consistió en utilizar algoritmos genéticos haciendo uso de un marco de trabajo *open source* llamado "WatchMaker", definido en su página web como *un framework de algoritmos evolutivos/genéticos extensible, de alto rendimiento y orientado a objetos*.

Entre los principales frameworks de computación evolutiva se encuentran ECJ, JGAP y WatchMaker. Sin embargo, se optó por utilizar éste último (en su versión 0.7.1) por ser un framework de propósito general, moderno, flexible y que ha sido creado tomando las mejores características de otros frameworks, además de contar con una buena documentación, estar bien estructurado y ser sencillo y rápido de aprender e implementar.

Al ser el más reciente de los tres y ser el único en utilizar Java JDK 5, aprovecha las ventajas de utilizar tipos genéricos ("generics") que ayudan a tener un mejor control sobre tipos de datos en tiempo de compilación, permitiendo corregir errores en tiempo de compilación y no de ejecución, lo cual resulta más complicado; los tipos genéricos aportan la eliminación de "casts" y la ventaja de implementar algoritmos genéricos que puedan ser personalizados y fáciles de leer[11].

Entre las principales características de este framework, tomadas de la página web oficial del proyecto[12], se encuentran:

- Un motor de evolución multi-hilo,
- además de varias estrategias de selección, la posibilidad e implementar una selección personalizada,
- esquemas de evolución flexibles,
- operadores re-utilizables para tipos comunes,
- modelo de evolución de isla y estado estable,

[11] http://docs.oracle.com/javase/tutorial/java/generics/why.html

[12] http://watchmaker.uncommons.org - 2006-2010 Daniel W. Dyer

- estrategias de evolución,
- procesamiento distribuido,
- componentes GUI que ayudan a simplificar la construcción de ambientes gráficos,
- clara y buena documentación y ejemplos,
- entre otros.

El segundo programa consistió en implementar un mapa auto-organizado de Kohonen, que como ya se estudió anteriormente, se trata del tipo de red neuronal de aprendizaje no supervisado más ampliamente utilizado. Para las redes neuronales existen diversas herramientas tales como: Encog, Joone, Neuroph, Matlab, etc., sin embargo, ninguna de ellas permite implementar un SOFM para optimización, en su lugar lo hacen para reconocimiento de patrones. Por lo tanto el programa fue desarrollado completamente desde cero.

Ambos programas consisten en encontrar el camino más corto para recorrer hasta 15 ciudades de México, sin la necesidad de regresar a la ciudad de origen.

Para el caso de los algoritmos genéticos y dadas las características y ventajas del framework, se tiene la posibilidad de seleccionar el tamaño de la población (número de cromosomas o posibles soluciones), elitismo (el número de "mejores individuos" que serán conservados hacia la siguiente generación), número de generaciones (iteraciones de evolución), operadores de mutación y cruzamiento, así como diferentes estrategias de selección (rango, ruleta, sigma, estocástica, torneo, truncamiento). Se deja a consideración del lector el estudio de la estructura y código del framework[13].

Por otro lado, en lo que al SOFM se refiere, se detallará la forma en que fue implementado ya que no se utilizó ningún framework. Para este caso, la red

[13] http://watchmaker.uncommons.org/api/index.html

consta de una capa de entrada y una de salida; la capa de entrada es de dos neuronas, las cuales indican las coordenadas de cada ciudad, mientras que la capa de salida puede ser una matriz de una o dos dimensiones.

En este desarrollo se utilizaron dos dimensiones para representar cada una de las neuronas, sin embargo, en lugar de utilizar una matriz de $m * n$, se utilizo un *anillo elástico* de tamaño $2n$, donde n es el número de ciudades para las cuales se pretende encontrar la ruta más corta.

Se utiliza el doble de neuronas respecto al número de ciudades con la finalidad de evitar que una neurona oscile entre dos ciudades cercanas, y aunque esto no es necesario, si es de ayuda para obtener mejores resultados. Lo que si será siempre necesario es tener un mínimo de neuronas igual al número de ciudades a recorrer.

En (Kohonen, 2001), como en otros autores, se lleva a cabo un estudio más detallado acerca del número de neuronas que se sugiere deben ser utilizadas para un óptimo desempeño.

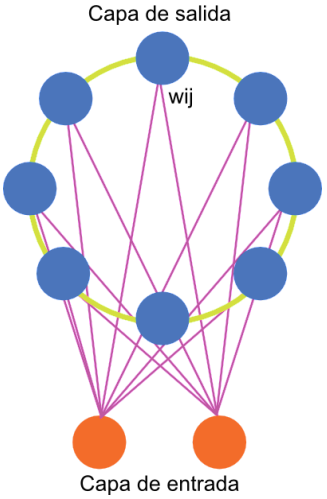

Figura 3.1. Red elástica.

En la figura 3.1 se pueden apreciar 2 neuronas en la capa de entrada y 8 neuronas en la capa de salida. Cada neurona de la capa de salida tiene un peso w_{ij}, donde i es su coordenada en x y j su coordenada en y.

Lo que se hace en el algoritmo es para cada una de las ciudades del recorrido (conjunto de entrenamiento), actualizar la capa de entrada en la que la primera neurona será la coordenada x de la ciudad en cuestión, y la segunda neurona será la coordenada y.

Como se trata de un método competitivo, para cada neurona de la capa de salida se obtiene su distancia euclídea respecto a las coordenadas de la capa de entrada, resultando ganadora, la neurona más cercana a la ciudad actual del conjunto de entrenamiento. Para el vector de entrada **x**, se tiene la siguiente fórmula con la que se obtiene la distancia euclídea:

$$d^2(\mathbf{w}_{ij}, \mathbf{x}) = \sum_{k=1}^{n} (w_{ij} - x_k)^2$$

Una ventaja introducida en los Mapas de Kohonen es que no únicamente se actualizan los pesos sinápticos de las neuronas ganadoras, sino también de las neuronas vecinas. Con esto, las neuronas próximas a la ganadora se sintonizan con patrones similares, una característica muy útil, por ejemplo, en el reconocimiento de patrones.

Se cuenta con diferentes funciones de vecindad que de acuerdo a un radio determinan si cierta neurona actualizará o no sus pesos sinápticos. Para determinar las neuronas vecinas se obtiene su distancia respecto a la neurona ganadora y de acuerdo a alguna de las funciones de la Figura 3.2 se sabe si será actualizada o no.

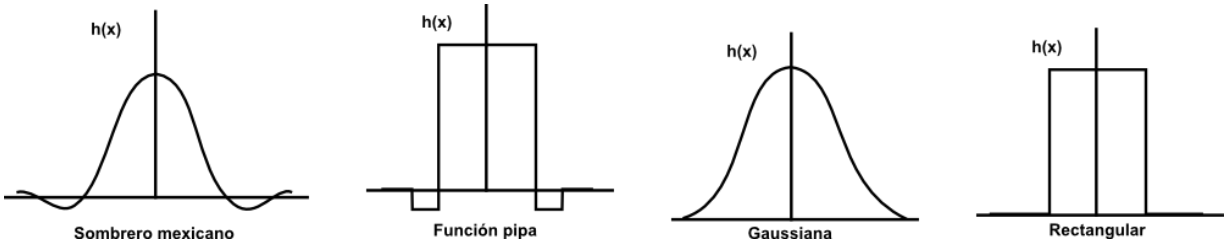

Figura 3.2. Funciones de vecindad.

En este caso la función utilizada fue la función rectangular, en la cual si una neurona no pertenece a la vecindad, el valor es 0 y no será actualizada. Una vez encontradas tanto la neurona ganadora como sus vecinas, se actualizan los pesos sinápticos de cada una de ellas con la intención de que estos pesos se parezcan cada vez más a las coordenadas de cada ciudad. Para la actualización de los pesos se utiliza la siguiente regla:

$$w_{ij}(t+1) = w_{ijk}(t) + \alpha(t) \cdot h(|\mathbf{i} - \mathbf{g}|, t) \cdot (x_k(t) - w_{ijk}(t))$$

donde $w_{ij}(t+1)$ representa el nuevo peso sináptico de la neurona actual; $w_{ijk}(t)$, el peso sináptico de la neurona actual antes del cambio; $\alpha(t)$, es el ritmo de aprendizaje, que de acuerdo a cada problema deberá ser ajustado de manera correcta para evitar una convergencia prematura o un sobre-entrenamiento de la red; $h(|\mathbf{i} - \mathbf{g}|, t)$, es la función de vecindad que determina que neuronas (**i**) pertenecen a la vecindad de la ganadora (**g**) y serán actualizadas; $(x_k(t) - w_{ijk}(t))$, determina finalmente la distancia que se va a recorrer desde la neurona actual hacia las coordenadas de la capa de entrada.

El procedimiento anterior se realiza hasta que se cumplan cierto número de iteraciones (*t*), y para cada iteración, tanto el ritmo de aprendizaje como el radio de vecindad irán disminuyendo hasta aproximarse a cero.

La función utilizada para el nuevo ritmo de aprendizaje es igual al aprendizaje al que queremos llegar al final del entrenamiento (cerca de cero), entre el

aprendizaje inicial, elevado a la iteración actual entre el número de iteraciones totales (o epochs - del inglés "época") que se llevarán a cabo. Todo esto multiplicado por el aprendizaje inicial, como lo muestra la siguiente fórmula:

$$\alpha(t) = \alpha_0 \left(\frac{\alpha_f}{\alpha_0}\right)^{\frac{t}{t_\alpha}}$$

mientras la función utilizada para el nuevo radio de vecindad es igual al radio que se quiere alcanzar al final (igual a 1, para actualizar solo las neuronas adyacentes), menos el radio de vecindad original; esto multiplicado por la iteración actual entre el número de iteraciones máximas y finalmente el producto sumado por el radio de vecindad original, esto es:

$$R(t) = R_0 + \left(R_f - R_0\right)\frac{t}{t_R}$$

Hay ocasiones en que los valores que se proporcionan en el conjunto de entrenamiento deben ser normalizados (escalados a un rango entre [0-1]) antes de ser utilizados, sin embargo, para este ejemplo se utilizaron pixeles, sin la necesidad de normalizar las coordenadas actuales.

4 IMPLEMENTACION, PRUEBAS Y RESULTADOS

4.1 Programa 1 - Algoritmos Genéticos

Primeramente, para el programa que realiza la búsqueda por medio de algoritmos genéticos, se llevan a cabo diversas ejecuciones, modificando ciertos parámetros, tales como: el número de cromosomas, el número de generaciones y el elitismo; todo esto con el objetivo de determinar hasta qué punto el algoritmo puede mejorar el resultado arrojado, y en que límite, los resultados no mejoran, requerido por otro lado, un mayor tiempo de ejecución.

Después de determinar a prueba y error un número mínimo adecuado para el tamaño de la población y número de generaciones, se obtuvieron con una población de 150 cromosomas, un elitismo de 15 individuos y 150 generaciones, los datos mostrados en la tabla 4.1 para diez ejecuciones con cada estrategia de selección.

Tal número de ejecuciones se determinó después de numerosas pruebas, en las cuales, para los parámetros descritos anteriormente, sería difícil obtener un mejor resultado para el máximo de ciudades a visitar, en este caso, 15.

Finalmente, dado que es una de las más utilizadas y proporciona una mayor variación, se utilizó la recombinación PMX propuesta por Goldberg y descrita anteriormente; así como una mutación con distribución de Poisson en lugar de un número fijo de mutaciones, ya que de esta manera la mutación se vuelve aleatoria, es decir, más parecida a la naturaleza.

La tabla siguiente, muestra los resultados de los experimentos, de los cuales la ruta más corta entre las 15 ciudades propuestas fue iniciando en la ciudad de Cancún y terminando en Tijuana, de la siguiente manera: Cancún -> Querétaro -> Puebla -> D.F -> Aguascalientes -> Zacatecas -> S.L.P. -> Guanajuato -> Morelia -> Guadalajara -> Mazatlán -> Los Cabos -> Chihuahua -> Monterrey -> Tijuana.

Tabla 4.1 Resultados de arrojados por el programa

Estrategía de selección	Mejor recorrido/Tiempo ejecución	Peor reocorrido/Tiempo ejecución
Rango	6,027 k.m. / 2.371 seg.	6,926 k.m. / 2.449 seg.
Ruleta	6,027 k.m. / 2.433 seg.	6,735 k.m. / 2.387 seg.
Sigma	6,068 k.m. / 2.34 seg.	6,778 k.m. / 2.465 seg.
Estocástica	6,027 k.m. / 2.496 seg.	7,088 k.m. / 2.59 seg.
Torneo	6,027 k.m. / 2.496 seg.	6,629 k.m. / 2.418 seg.
Truncamiento	6,027 k.m. / 2.418 seg.	6,943 k.m. / 2.277 seg.

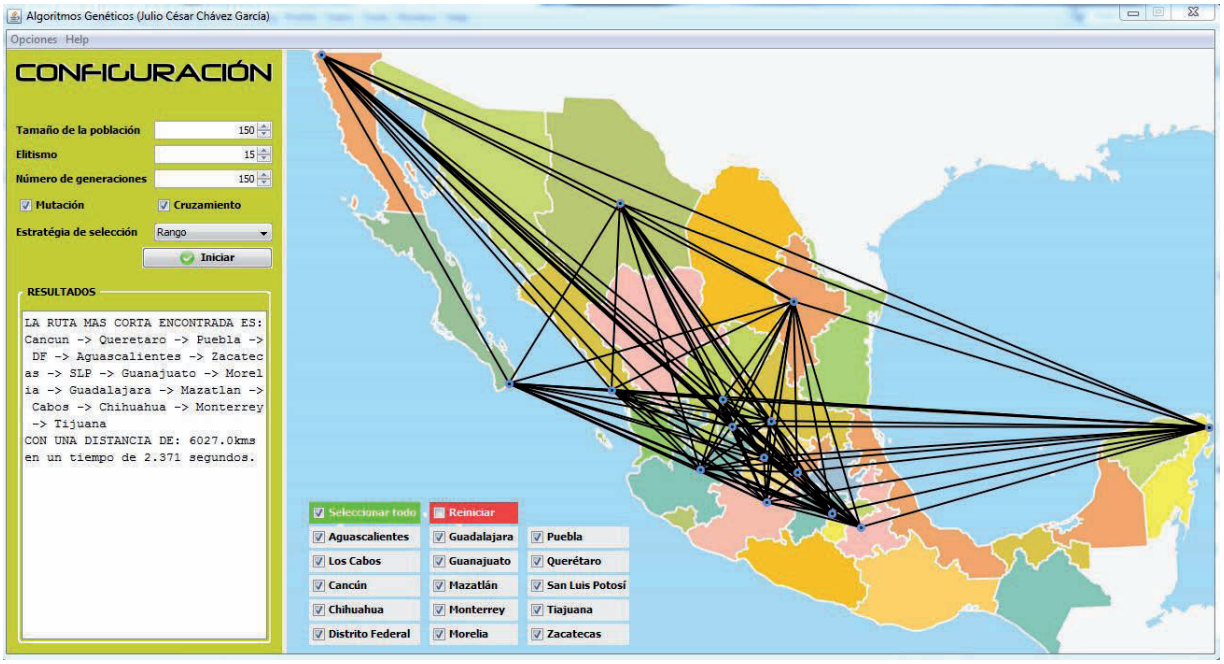

Figura 4.1. Resultados Agente Viajero en Java.

Para concluir con los algoritmos genéticos, si se tiene que un año cuenta con 365 días, 1 día está formado por 24 horas y 1 hora por 3600 segundos (sin tomar en cuenta los desfases de horas que llevan a formar años bisiestos), se tienen 31'536,000 segundos al año. Considerando un ordenador que en promedio arroje 6,000 permutaciones por segundo, se obtienen $1.89216x10^{11}$ permutaciones al año.

Así, dividiendo el número total de permutaciones posibles (15! o $1.307674368x10^{12}$) se tiene que la solución óptima se encontraría aproximadamente después de 6.911 años empleando un método de comparación por fuerza bruta que explore la totalidad del espacio de búsqueda.

La diferencia entre estos casi 7 años y los 2.5 segundos que en promedio resultaron de las pruebas realizadas, habla de la ventaja que proporcionan los algoritmos genéticos al ir acercándose a los individuos más aptos en cada región del espacio de búsqueda, lo que les permite no explorar éste último por completo y además encontrar una buena solución en un tiempo aceptable.

Como ya se estudió, los algoritmos genéticos tienen la ventaja de al ser estocásticos y no seguir una estrategia con conocimientos previos, evitar los problemas que ésta conlleva, como descartar caminos buenos y explorar lugares que la inteligencia humana omitiría.

Además de la información mostrada en la tabla 4.1, los datos arrojados por las pruebas indican que un número insuficiente de generaciones o una población muy pequeña, no explora por completo el espacio de búsqueda, proporcionando soluciones más alejadas del óptimo global.

Por otro lado, si el tamaño de la población y el número de generaciones es muy grande, la exploración se vuelve muy extensa, encontrando los mismos resultados, pero en un mucho mayor periodo de tiempo. Por ejemplo, para las mismas quince ciudades, con una población de 900 individuos, 10 como valor de elitismo y 900 generaciones, el tiempo de ejecución ascendió a 8.47 segundos en promedio, sin embargo, subiendo el tamaño de la población a 5,000 individuos, 20 como valor de elitismo y 50,000 generaciones, el tiempo de ejecución fue de 3.38 minutos, lo cual comprueba, como un mal diseño del algoritmo y/o elección de parámetros puede llevar a un alto e innecesario costo de recursos, tanto computacionales, como de tiempo o hasta dinero, dependiendo del problema que se esté tratando.

4.2 Programa 2 - Redes Neuronales

Para el segundo programa se trataron las mismas 15 ciudades. El tipo de red que se utilizó fue un Mapa Auto-organizado de Kohonen, para el cual primeramente se inicializa el mapa con las ciudades que se pueden llegar a seleccionar, como lo muestra la figura 4.2.

Figura 4.2. Ciudades seleccionadas para el recorrido.

El segundo paso consistió inicializar la red de neuronas, que en este caso serían *2n=10*. Los anillos se inician dividiendo los 360 grados (radianes) de una circunferencia entre el número de neuronas. Esto indica la separación en grados de cada neurona. La red se fijó de un radio de 50 pixeles, así, se obtienen la ubicación de cada neurona por medio de coordenadas polares. Lo anterior se ilustra en la Figura 4.3.

Figura 4.3. Red inicial.

De acuerdo al número de ciudades, los parámetros descritos anteriormente deben ser ajustados. Después de varios experimentos, apoyado en las fórmulas descritas y por medio de prueba y error, se determinaron como ritmo de aprendizaje inicial, final, radio de vecindad y número de epochs, las cantidades de: 0.8, 0.01, 10 y 5000 respectivamente. Con estos valores, el algoritmo es capaz de encontrar la mayoría de las veces rutas óptimas o cercanas al óptimo, para un mínimo de 4 ciudades y el máximo definido de 15.

Una vez obtenidas las distancias entre las ciudades, iniciada la red con dos neuronas en la capa de entrada y *2n* en la capa de salida, las iteraciones o epochs máximos, se llevan a cabo tal como lo ilustra la siguiente figura:

```
COMIENZO
   HACER
      ACTUALIZAR capa de entrada seleccionando una ciudad al azar del conjunto de entrenamiento
      ENCONTRAR neurona ganadora
      ENCONTRAR neuronas vecinas
      ACTUALIZAR pesos sinápticos de estas neuronas
      ACTUALIZAR ritmo de aprendizaje
      ACTUALIZAR radio de vecindad
      SUMAR epoch actual + 1
   MIENTRAS (epoch sea menor que el número de epochs totales)
   FIN HACER
FIN
```

Figura 4.4. Pseudo-código de iteraciones.

Después de correr el programa, se obtiene una salida como la siguiente:

Figura 4.5. Mejor ruta encontrada por la red.

Durante la ejecución del algoritmo, cuando una neurona se encuentra bastante cerca de una ciudad, se crea una relación 1-a-1. Al final del algoritmo las neuronas que no se encuentren asociadas a una ciudad son eliminadas, de esta manera en el mapa aparecen únicamente las neuronas que se encuentren relacionadas a las ciudades seleccionadas.

La siguiente figura muestra una posible ruta obtenida por la red para un total de 15 ciudades seleccionadas:

Figura 4.6. Una posible ruta encontrada para 15 ciudades.

Después de 50 experimentos, cantidad en la cual empiezan a arrojarse resultados repetidos sin prácticamente ninguna mejora, la ruta más corta encontrada fue de 10,372 km en un tiempo de 0.049 segundos, mientras la ruta más larga fue de 14,442 km en un tiempo de 0.056 segundos. A pesar de no ser la distancia más larga

ni la más corta, el algoritmo tardó en encontrar una ruta en 0.168 segundos, que fue el mayor tiempo necesitado por la red para arrojar una posible solución.

4.3 Análisis de resultados

Como se describió a lo largo de este trabajo, ambas técnicas (AG y RN), son de gran utilidad en problemas complicados de analizar matemáticamente, o para los cuales encontrar la respuesta requiere de un tiempo considerable.

Para el problema del agente viajero, los algoritmos genéticos superan a las redes neuronales en cuanto a que su respuesta es más cercana al óptimo conforme el espacio de búsqueda crece. Las redes neuronales (en especial SOFM), alcanzan muchas de las ocasiones buenos resultados, aunque estos dependerán de los parámetros de entrenamiento bajo los cuales se configure la red.

Una ventaja de las RA sobre los AG es su alta velocidad de procesamiento aún cuando el número de ciudades y/o *epochs* sea muy grande, esto gracias a la sencillez de su algoritmo respecto a otros que resultan ser más exhaustivos; sin embargo, cuando el número de ciudades aumenta en demasía o existen dos o más ciudades muy cercanas entre sí, hay ocasiones en que las neuronas oscilan entre una ciudad y otra, provocando que se caiga en óptimos locales.

Se ha mencionado que las RN trabajan bien en presencia de ruido y son capaces de eliminarlo, sin embargo, cuando en un problema de optimización se llega a caer en un óptimo local, la presencia de ruido puede resultar útil para escapar de éste y continuar buscando una ruta hacia el óptimo global.

Por otra parte, a partir de los resultados obtenidos durante la experimentación, se puede concluir que los AG resultan más eficientes para salir de los óptimos locales. La causa de esta ventaja son los operadores de cruzamiento y mutación que permiten explorar de mejor manera el espacio de búsqueda.

Muchas de las veces en que las RN arrojan soluciones un tanto distantes del óptimo, se pueden emplear diversos métodos o maneras de contrarrestar estos inconvenientes, entre los que se pueden mencionar:

- Utilizar un algoritmo que optimice la ruta final (por ejemplo, 2-opt),
- eliminar durante la ejecución neuronas que después de varias iteraciones no han sido ganadoras o que no se hayan movido de su posición original,
- cuando una neurona es ganadora para varias ciudades, crear una neurona cercana, para evitar oscilaciones,
- cuando una neurona ha sido establecida en una relación 1-a-1 con una ciudad, evitar que dicha neurona participe en la competencia, o bien, la ciudad ya no sea considerada en el conjunto de entrenamiento.

En cuanto a la lógica difusa se refiere, ésta es utilizada principalmente para control y es de gran utilidad en ambientes donde se cuenta con incertidumbre, al igual que con datos incompletos o con ruido. Su potencial radica en el uso de conocimiento previo, el trato con incertidumbre, la simplicidad de sus operaciones y su incorporación con otras técnicas.

5 CONCLUSIONES Y TRABAJO FUTURO

La presente investiga ción ha servido para introducir tres técnicas para la resolución de problemas, éstas técnicas que resultan ser sencillos modelos de la naturaleza, sirven ayudar a resolver, o en su defecto, arrojar soluciones aproximadas a problemas que resultan tornarse muy difíciles de tratar por medio de algoritmos comunes.

La teoría plasmada en este proyecto puede servir como una iniciación para aquellos que se encuentren interesados en el campo de la inteligencia computacional y deseen conocer la teoría y fundamentos de las técnicas aquí descritas a fin de que puedan ser empleadas en situaciones reales.

Sobre las técnicas descritas en los capítulos anteriores, a pesar de sus ventajas e inconvenientes, no se puede hablar de un solucionador universal de problemas, sin embargo, se tienen tres herramientas las cuales pueden formar sistemas híbridos de mayor capacidad. Como ejemplo, un sistema de control industrial que funcione en base a los resultados proporcionados por una red neuronal entrenada por medio de algoritmos genéticos.

Si bien, el uso de estas técnicas individualmente o en sistemas híbridos no resolverán todos los problemas complejos del mundo real, contribuirán con un acercamiento que permita emplear otras técnicas y conocimientos en una dirección acertada, sirviendo de apoyo al hombre para resolver y solucionar sus problemas y alcanzar sus objetivos de una manera más rápida, eficiente y menos costosa.

Es por eso, que con el conocimiento adquirido a lo largo de la realización de este proyecto, se pretende en un futuro desarrollar sistemas híbridos que contribuyan al desarrollo de la ciencia, principalmente en el área de la bio-informática y/o medicina.

De esta manera, el objetivo es trabajar en la construcción de estructuras proteínicas y predicción en la evolución de células cancerígenas con la intención de aportar beneficios a la salud y avances a la ciencia.

BIBLIOGRAFÍA

[1] Anderson, J. A. (2007). Redes Neuronales. México, D.F.: Alfaomega.

[2] Asadi, A. A., Naserasadi, A., & Asado, Z. A. (n.d.). A New Hybrid Algorithm for Traveler Salesman Problem based on Genetic Algorithms and Artificial Neural Networks. Irán

[3] Barrera, H. A. (1992). Información genética: Estructura, función y manipulación. CONACYT.

[4] Barrionuevo, F. J. (2008). Computación Evolutiva. In J. T. Palma, & R. Marín, Inteligencia Artificial: Técnicas, métodos y aplicaciones (pp. 433-469). Madrid: McGraw Hill.

[5] Bindu, & Tanwar, P. (2012 йил Junio). International Journal of Computer Science Y Communication Networks. From http://www.ijcscn.com/Documents/Volumes/vol2issue3/ijcscn2012020322.pdf

[6] Brocki, L. (n.d.). Kohonen Self-Organizing Map for the Traveling Salesperson Problem. Varsovia, Polonia.

[7] Castillo, O., & Melin, P. (2008). Type-2 Fuzzy Logic: Theory and Applications. Springer-Verlag.

[8] Cvalcarcel. (2009 йил 13-Septiembre). From http://cvalcarcel.wordpress.com/category/software-development/genetic-programming/

[9] Del Brío, M., & Sanz, A. (2002). Redes Neuronales y Sistemas Difusos. México D.F.: Alfaomega.

[10] Delgado, M. L., & Martín, Q. M. (2003 йил Abril). DISEÑO DE UNA RED NEURONAL AUTOASOCIATIVA PARA RESOLVER EL PROBLEMA DE VIAJANTE DE COMERCIO (TRAVELING SALESMAN PROBLEM, TSP).

[11] Eiben, A. E., & Smith, J. E. (2003). Introduction to Evolutionary Computing. Springer.

[12] El Huron Inteligente. (2012). Retrieved 2012 йил Mayo from Blog de inteligencia artificial y algoritmos avanzados: http://elhuroninteligente.blogspot.mx/2012/01/computacion-evolutiva-programacion.html

[13] Félix Lamas, P., & Bugarín Diz, A. Conjuntos borrosos. En L. NEGRO.

[14] George Mason University. (2008 йил 10-Septiembre). EClab. From http://cs.gmu.edu/~eclab/projects/ecj/

[15] Guijarro Berdiñas, B., Fontela Romero, O., & Sánchez Meroño, N. (2008). Redes Neuronales. In E. K. Burke, & G. Kendall, Inteligencia Artificial: Técnica, métodos y aplicaciones (pp. 649-689). Madrid: McGraw Hill.

[16] Jin, H.-d., Leung, K.-S., & Wong, M.-L. (n.d.). An Integrated Self-Organizing Map for the Traveling Salesman Problem.

[17] Kohonen, T. (1990). The Self-Organizing Map.

[18] Markovic, D., Madic, M., & Vojislav Tomic, S. S. (2012). SOLVIND TRAVELLING SALESMAN PROBLEM BY USE OF KOHONEN SELF-ORGANIZING MAPS. Rumania.

[19] Martikainen, J., & Ovaska, S. J. (2005). USING FUZZY EVOLUTIONARY PROGRAMMING TO SOLVE TRAVELING SALESMAN PROBLEMS. Finlandia.

[20] Martikainen, J., & Ovaska, S. J. (2005). USING FUZZY EVOLUTIONARY PROGRAMMING TO SOLVE TRAVELING SALESMAN PROBLEMS. España.

[21] Martikainen, J., & S., O. (2005). Aalto University Library. From http://lib.tkk.fi/Diss/2006/isbn951228524X/article6.pdf

[22] Michalewicz, Z. (1998). Genetic Algorithms + Data Structures = Evolution Programs. Springer.

[23] Nilsson, N. (2001). Inteligencia Artificial: Una nueva síntesis. Madrid: McGraw-Hill.

[24] Orable. (2013). Oracle. From http://docs.oracle.com/javase/tutorial/java/generics/why.html

[25] Paton, R. C. (1997). Principles of genetics. In T. Baeck, D. Fogel, & Z. Michalewicz, Handbook of Evolutionary Computation. United Kingdom: Oxford University Press.

[26] s.r.l., I. (2010). Jenes 2.0. From https://sites.google.com/a/intelligentia.it/jenes/

[27] Sánchez, N., & Alanís, Y. (2006). Redes Neuronales: Conceptos fundamentales y aplicaciones a control automático. Pearson Education.

[28] Sastry, K., Goldberg, D., & Kendall, G. (2005). Genetic Algorithms. In E. K. Burke, & G. Kendall, Search Methodologies: Introductory Tutorials in Optimization and Decision Support Techniques (pp. 97-115). Springer.

[29] Sazonov, E. (2004 йил 06-Junio). Fuzzy Relations and Composition of Fuzzy Relations. Retrieved 2010 йил 19-Octubre from http://www.intelligent-systems.info/classes/ee509/6.pdf

[30] Timothy, R. (2010). Fuzzy Logic with engineering applications. United Kingdom: Wiley & Sons, Ltd.

ÍNDICE DE FIGURAS

ÍNDICE DE TABLAS

APÉNDICE 1

Programa Algoritmos Genéticos AG.java

Clase que muestra la creación y ejecución de un algoritmo genético, usando el framework Watchmaker.

```
package Genetic;
import java.util.ArrayList;
import java.util.LinkedList;
import java.util.List;
import java.util.Random;
import org.uncommons.maths.number.NumberGenerator;
import org.uncommons.maths.random.MersenneTwisterRNG;
import org.uncommons.maths.random.PoissonGenerator;
import org.uncommons.watchmaker.framework.CandidateFactory;
import org.uncommons.watchmaker.framework.EvolutionaryOperator;
import org.uncommons.watchmaker.framework.GenerationalEvolutionEngine;
import org.uncommons.watchmaker.framework.operators.EvolutionPipeline;
import org.uncommons.maths.random.Probability;
import org.uncommons.watchmaker.framework.EvolutionObserver;
import org.uncommons.watchmaker.framework.FitnessEvaluator;
import org.uncommons.watchmaker.framework.PopulationData;
import org.uncommons.watchmaker.framework.SelectionStrategy;
import org.uncommons.watchmaker.framework.factories.ListPermutationFactory;
import org.uncommons.watchmaker.framework.operators.ListOrderCrossover;
import org.uncommons.watchmaker.framework.operators.ListOrderMutation;
import org.uncommons.watchmaker.framework.selection.*;
import org.uncommons.watchmaker.framework.termination.GenerationCount;
import tesisga.Mapa;

/**
 *
 * @author Jules
 */
public class AG {
    private int poblacion, elitismo, generaciones;
    GenerationalEvolutionEngine<List<String>> engine;
    private boolean mutacion, cruzamiento;
    private String seleccion;
    char [] chromosome;
    private Mapa objMapa;
    private Random rng;
    private SelectionStrategy<Object> selection;
    private FitnessEvaluator<List<String>> fitnessEvaluator;
    private CandidateFactory<List<String>> factory;
    private EvolutionaryOperator<List<String>> operator;
    private List<String> result;
    private String [][] distancias;
    private String ciudades, opSelec;
    public double distancia;
    public String lineasCiudades[][];
```

```
   AG(String ciudades, String[][] distancias, String poblacion, String elitismo, String generaciones, boolean mutacion, boolean cruzamiento, String opSelec) {
      this.ciudades = ciudades;
      this.distancias = distancias;
      this.poblacion = Integer.parseInt(poblacion);
      this.elitismo = Integer.parseInt(elitismo);
      this.generaciones = Integer.parseInt(generaciones);
      this.mutacion = mutacion;
      this.cruzamiento = cruzamiento;
      this.opSelec = opSelec;
      prepareThings();
   }
  // Preparamos todo lo necesario para echar a andar el algoritmo
   private void prepareThings() {

      // Elige que tipo de selección va a utilizar
      if(opSelec.equalsIgnoreCase("rango"))
         selection = new RankSelection();

      else if(opSelec.equalsIgnoreCase("ruleta"))
         selection = new RouletteWheelSelection();

      else if(opSelec.equalsIgnoreCase("sigma"))
         selection = new SigmaScaling();

      else if(opSelec.equalsIgnoreCase("estocástica"))
         selection = new StochasticUniversalSampling();

      /*
       * Probabilidad de 75% de que sean seleccionados los elementos más adaptados. El 75% para que tengan una oportunidad considerable los
       * elementos menos adaptados y no se pierda tan fácil la diversidad.
       */
      else if(opSelec.equalsIgnoreCase("torneo")) {
         selection = new TournamentSelection(new NumberGenerator<Probability>() {
            @Override
            public Probability nextValue() {
               return new Probability(0.75);
            }
         });
      }

      else if(opSelec.equalsIgnoreCase("truncamiento"))
         selection = new TruncationSelection(new NumberGenerator<Double>() {
         @Override
         public Double nextValue() {
            return 0.5;
         }
      });

      // Esta línea necesita ir arriba de las tres siguientes, porque sino marca excepción por no poder generar los números aleatorios
      rng = new MersenneTwisterRNG();
      // Clase creada para evaluar individuos y regresar de cada generación el mejor
      fitnessEvaluator = new StringEvaluator();
      // Inicia soluciones candidatas
      factory = initChromosomes();
      // Inicia operadores (mutación & cruzamiento)
      operator = initOperators();
```

```
        /*
         * Ya que se crearon la forma de evaluar las soluciones, se iniciaron los primeros candidatos y se determinó que tipo de operadores se van a
         * usar; se crea el motor, se agregar el observador para ir checando e imprimiendo los mejores cromosomas y se inicia la búsqueda con el
         * método envolve.
         */

        engine = new GenerationalEvolutionEngine<List<String>>(
                factory,
                operator,
                fitnessEvaluator,
                selection,
                rng);

        engine.addEvolutionObserver(new EvolutionObserver<List<String>>() {
            public void populationUpdate(PopulationData<? extends List<String>> data) {
                //System.out.println("Generación: " + (data.getGenerationNumber()+1) + ", mejor candidato: " + data.getBestCandidate());
            }
        });

        // tamaño poblacion, num elitismo, # de generaciones
        //result = engine.evolve(15, 1, new GenerationCount(20));
        result = engine.evolve(poblacion, elitismo, new GenerationCount(generaciones));
    }

    private CandidateFactory<List<String>> initChromosomes() {
        List lista = new LinkedList<String>();
        // Los genes del cromosoma, son las letras de las ciudades seleccionadas
        chromosome = ciudades.toCharArray();

        for(int i = 0; i< chromosome.length; i++)
            lista.add(chromosome[i]);
        //CandidateFactory<List<String>> candidateFactory
            //= new ListPermutationFactory<String>(new LinkedList<String>(cities));
        //CandidateFactory<String> factory = new StringFactory(chromosome, chromosome.length);

        /*
         * La permutación se usa en el agente viajero, porque SI importa el orden de las ciudades.
         */
        CandidateFactory<List<String>> factory = new ListPermutationFactory<String>(lista);
        return factory;
    }

    private EvolutionaryOperator<List<String>> initOperators() {
        // Se crea de tamaño dos porque en la clase Opciones verifica que por fuerza se usen mutación y cruzamiento
        List<EvolutionaryOperator<List<String>>> operators = new ArrayList<EvolutionaryOperator<List<String>>>(2);

        /*
         * También se puede usar StringCrossover que puede poner uno o varios puntos de cruce fijos o aleatorios, pero todos los cromosomas deben
         * ser del mismo tamaño o arroja excepción. Mejor se usó ListOrderCrossover, que es básicamente el PMX (partially mapped x-over), propuesto
         * por Goldberg, de los más usados. La mutación es mayor y resulta más útil para cadenas de caracteres
         */
        operators.add(new ListOrderCrossover<String>());
```

```
    /*
     * Se utiliza el poisson generator porque para switchear los elementos se puede hacer con un número fijo de mutaciones, o aleatoriamente,
     * el poisson determina cuantas mutaciones aplicar... (más aleatorio, más parecido a la naturaleza)
     */
    operators.add(new ListOrderMutation<String>(new PoissonGenerator(1.5, rng), new PoissonGenerator(1.5, rng)));

    EvolutionaryOperator<List<String>> pipeline = new EvolutionPipeline<List<String>>(operators);

    return pipeline;
  }

  public List<String> getResult(){
    return this.result;
  }

  private String[][] obtenerLineasCiudades(String ciudades) {
    String[][] lineasCiudades = new String[ciudades.length()][2];

    for(int i = 0; i < ciudades.length(); i++) {
      /*
       * En cada iteración obtendo todas las letras de las ciudades seleccionadas en un orden que me da el cromosoma, lo paso a un array
       * de chars, para cada elemento del array, si obtengo el nombre de la ciudad, y como las ciudades no tienen un número, ese se lo
       * asigno aca con los if else, para después mandar pintar las líneas que ocupan el nombre y número de la ciudad
       */

      char[] letras = ciudades.toCharArray();

      if(Mapa.ciudades[i].getChar() == letras[i]) {
        lineasCiudades[i][0] = Mapa.ciudades[i].getName();
        if(letras[i] == 'A') lineasCiudades[i][1] = "0";
        else if(letras[i] == 'B') lineasCiudades[i][1] = "1";
        else if(letras[i] == 'C') lineasCiudades[i][1] = "2";
        else if(letras[i] == 'D') lineasCiudades[i][1] = "3";
        else if(letras[i] == 'E') lineasCiudades[i][1] = "4";
        else if(letras[i] == 'F') lineasCiudades[i][1] = "5";
        else if(letras[i] == 'G') lineasCiudades[i][1] = "6";
        else if(letras[i] == 'H') lineasCiudades[i][1] = "7";
        else if(letras[i] == 'I') lineasCiudades[i][1] = "8";
        else if(letras[i] == 'J') lineasCiudades[i][1] = "9";
        else if(letras[i] == 'K') lineasCiudades[i][1] = "10";
        else if(letras[i] == 'L') lineasCiudades[i][1] = "11";
        else if(letras[i] == 'M') lineasCiudades[i][1] = "12";
        else if(letras[i] == 'N') lineasCiudades[i][1] = "13";
        else if(letras[i] == 'O') lineasCiudades[i][1] = "14";
      }

      for(int j = 0; j < lineasCiudades.length; j++) {

      }
    }

    return lineasCiudades;
  }

  private class StringEvaluator implements FitnessEvaluator<List<String>> {
```

```
    public double getFitness(List<String> candidate, List<? extends List<String>> population) {

        int distance = 0;
        char from, to;

        for(int i = 0; i < candidate.size()-1 ; i++) {
            from = String.valueOf(candidate.get(i)).charAt(0);
            to = String.valueOf(candidate.get(i+1)).charAt(0);

            if(from != to) {
                int count = 0, count2 = 0;
                salir: for(char s = 'A'; s <= 'O' ; s++) {
                    count++;
                    if(s == from) {
                        for(char t = 'A'; t <= 'O' ; t++) {
                            count2++;
                            if(t == to) {
                                distance += Integer.parseInt(distancias[count][count2]);
                                break salir;
                            }
                        }
                    }
                }
            }
        }

        return distance;
    }

    // Si natural TRUE, regresa el valor mayor. En este caso queremos la ruta más corta, entonces lo seteamos FALSE
    public boolean isNatural(){
        return false;
    }
}

    }
```

APÉNDICE 2

Mapa.java

Código que muestra como se configura y ejecuta un Mapa Auto-organizado de Kohonen.

```
private void jB_nnActionPerformed(java.awt.event.ActionEvent evt) {
    ArrayList<int[][]> vectorEntrada = getVectorEntrada();
    CapaEntrada capaEntrada;
    CapaSalida capaSalida;

    if(numCBSelec > 3) {
      // Permite pintar los círculos en el mapa.
      nnSelected = true;

      // Obtengo las coordenadas iniciales del círculo y repinto; a la vez que obtengo los datos de entrada/entrenamiento.
      alCoordsNeuronas = getCoordsIniciales(numCBSelec);

      // Crea la capa de entrada con sólo dos neuronas que serán las coordenadas (x,y).
      capaEntrada = new CapaEntrada(new Neurona(), new Neurona());

      // Crea la capa de salida con las coordenadas iniciales, para cada par (x,y) va a crear una neurona perteneciente a esta
capa.
      capaSalida = new CapaSalida(alCoordsNeuronas);

      // Creo la red con un ritmo de aprendizaje de 0.1 y una vecindad de 100 (pixeles a la redonda en los que las neuronas
son modificadas).
      // Una vecindad muy grande mueve todas las neuronas al mismo lado
      som = new SOM(capaEntrada, capaSalida, 0.8, 0.01, 10);

      // Empieza el entrenamiento con 1000 epochs.
      int epoch = 0, maxEpochs = 5000;
      int i = 0;
      // Para obtener el tiempo
      long tiempoInicio = java.util.Calendar.getInstance().getTimeInMillis();
      do {
        if(i == (vectorEntrada.size()-1)) i = 0;
        // Obtengo un número aleatorio, para seleccionar las coordenadas para la capa de entrada en cada iteración
        int ciudadRand = (int)(Math.random() * vectorEntrada.size());

        // Checa si esa ciudad ya tiene una neurona asignada, sino, entra
        //if(!som.hasNeuron(vectorEntrada.get(ciudadRand)[0][0], vectorEntrada.get(ciudadRand)[0][1])) {
          som.actualizaCapaEntrada(vectorEntrada.get(ciudadRand)[0][0], vectorEntrada.get(ciudadRand)[0][1]);
          som.encontrarNeuronaGanadora();
          som.encontrarNeuronasVecinas();
          som.actualizarPesos();
          som.actualizarAprendizaje(epoch, maxEpochs);
          som.actualizarRadioVecindad(epoch, maxEpochs);;
          epoch++;
          i++;
        //}

      } while(epoch < maxEpochs);
```

```
        // Comentar la siguiente línea y quitar el ciclo de arriba si se quiere obtener el anillo inicial
        alCoordsNeuronas = som.getFinalCoords();

        long tiempoFinal = java.util.Calendar.getInstance().getTimeInMillis();
        double diferenciaTiempo = (tiempoFinal*1.0 - tiempoInicio*1.0)/1000;
        System.out.println("LA DISTANCIA FUE: " + getDistance() + " en un tiempo de: " + diferenciaTiempo);

        repaint();
        jB_nn.setSelected(false);
      } else { JOptionPane.showMessageDialog(this, "Al menos deben seleccionarse 4 ciudad"); }
    }
```

SOM.java (self-organizing map)

Código del Mapa Auto-organizado de Kohonen

```
package nn;

import java.util.ArrayList;
import tesisnn.Mapa;

/**
 *
 * @author Jules
 */
public class SOM {
    private ArrayList<int[][]> alCoordsIniciales;
    private double ritmoAprendizaje, ritmoAprendizaje0, aprendizajeFinal;
    private CapaEntrada capaEntrada;
    private CapaSalida capaSalida;
    private double distanciaMenor;
    private int vecindad, vecindad0, winner, posicion;

    public SOM(CapaEntrada capaEntrada, CapaSalida capaSalida, double ritmoAprendizaje, double aprendizajeFinal, int
vecindad) {
        this.capaEntrada = capaEntrada;
        this.capaSalida = capaSalida;
        this.ritmoAprendizaje = this.ritmoAprendizaje0 = ritmoAprendizaje;
        this.vecindad = this.vecindad0 = vecindad;
        this.aprendizajeFinal = aprendizajeFinal;
    }

    public void actualizaCapaEntrada(int x, int y) {
        capaEntrada.setCoordNeuronaX(x);
        capaEntrada.setCoordNeuronaY(y);
    }

    public void encontrarNeuronaGanadora() {
        // Se reinician los valores para cada iteración
        distanciaMenor = 999999999;
        winner = -1;

        int i = 0;
        while(i < capaSalida.neuronas.size()) {
            // Recorro el todo el ArrayList eliminando la neurona ganadora anterior.
```

```
        capaSalida.neuronas.get(i).setWinner(false);
        capaSalida.neuronas.get(i).setNeighbour(false);

        // Si la neurona no se encuentra ya posicionada en una ciudad, entonces entra a participar
        if(!capaSalida.neuronas.get(i).isPlaced()) {
          // Obtengo todas las distancias Euclideas de cada neurona, respecto a las coordenadas de la capa de entrada
          Neurona neuronaActual = capaSalida.neuronas.get(i);
          int x2_x1 = capaEntrada.nX.getCoord() - neuronaActual.getX();
          int y2_y1 = capaEntrada.nY.getCoord() - neuronaActual.getY();
          double distanciaActual = Math.sqrt(Math.pow(x2_x1,2) + Math.pow(y2_y1,2));

          // Si la distanacia actual es menor que la anterior, se actualiza y también el índice de la neurona ganadora
          if(distanciaActual < distanciaMenor) {
            winner = i;
            distanciaMenor = distanciaActual;
          }
        }

        i++;
      }

      capaSalida.neuronas.get(winner).setWinner(true);
    }

    public void encontrarNeuronasVecinas() {
      // Coordenadas de la neurona ganadora
      int ganadoraX = capaSalida.neuronas.get(winner).getX();
      int ganadoraY = capaSalida.neuronas.get(winner).getY();

      int i = 0;
      while(i < capaSalida.neuronas.size()) {
        // Obtengo las distancias Euclideas de cada neurona, respecto a las coordenadas de la neurona ganadora.
        Neurona neuronaActual = capaSalida.neuronas.get(i);

        // Si la neurona no es la ganadora ni está ya establecida
        if(!neuronaActual.isWinner() && !capaSalida.neuronas.get(i).isPlaced()) {
          int x2_x1 = ganadoraX - neuronaActual.getX();
          int y2_y1 = ganadoraY - neuronaActual.getY();
          double distancia = Math.sqrt(Math.pow(x2_x1,2) + Math.pow(y2_y1,2));

          // Si la distancia de la neurona se encuentra dentro del radio de vecindad, se agrega al ArrayList de neuronas vecinas.
          if(distancia <= vecindad)
            capaSalida.neuronas.get(i).setNeighbour(true);
        }

        i++;
      }
    }

    /*
     * Se actualizan los pesos de las neuronas (primero la ganadora y después las vecinas) por medio de la siguiente fórmula:
     *  w(t+1)    = w(t) + lr(t) * (v(t)-w(t)), donde:
     *  w(t+1)   -> los nuevos (x,y) de la neurona en cuestión
     *  w(t)     -> los pesos actuales (x,y) de la neurona en cuestión
     *  lr(t)    -> learming rate o ritmo de aprendizaje en la iteración t
     *  v(t)-w(t) -> distancia euclidea del vector de entrada, es decir la ciudad que se está evaluando, respecto a la neurona actual
     *
```

```
    * Es decir, que lo que se hace es obtener la distancia de cada neurona (ganadora y vecinas) respecto a las coordenadas de entrada (alguna ciudad)
    * multiplicarla por el ritmo de aprendizaje, y ese pequeño resultado va a ser la distancia que se moverá cada neurona. Para ello, hay que sumar
    * la pequeña distancia a moverse y la posición actual.
    */
   public void actualizarPesos() {
      // Primero actualizamos la neurona ganadora.
      int i = 0;
      while(i < capaSalida.neuronas.size()) {
         Neurona neuronaActual = capaSalida.neuronas.get(i);
         if(neuronaActual.isWinner() || neuronaActual.isNeighbour()) {
            int x2_x1 = capaEntrada.nX.getCoord() - neuronaActual.getX();
            int y2_y1 = capaEntrada.nY.getCoord() - neuronaActual.getY();

            // Multiplico la distancia de la neurona hacia la ciudad, por el ritmo de aprendizaje
            double distancia = Math.sqrt(Math.pow(x2_x1,2) + Math.pow(y2_y1,2)) * ritmoAprendizaje;

            // A las coordenadas (x,y) de la neurona actual, le sumo la distancia que va a recorrer. Math.round acepta float, entonces casteo.
            int nuevoX, nuevoY;

            if(Math.min(capaEntrada.nX.getCoord(),neuronaActual.getX()) == capaEntrada.nX.getCoord())
               nuevoX = Math.round((float)(neuronaActual.getX() - distancia));
            else
               nuevoX = Math.round((float)(neuronaActual.getX() + distancia));

            if(Math.min(capaEntrada.nY.getCoord(),neuronaActual.getY()) == capaEntrada.nY.getCoord())
               nuevoY = Math.round((float)(neuronaActual.getY() - distancia));
            else
               nuevoY = Math.round((float)(neuronaActual.getY() + distancia));

            capaSalida.neuronas.get(i).setX(nuevoX);
            capaSalida.neuronas.get(i).setY(nuevoY);
            capaSalida.neuronas.get(i).setMoved(true);
         }

         i++;
      }
   }

   public void actualizarAprendizaje(int t, int maxEpochs) {
      /* El nuevo ritmo de aprendizaje es igual al aprendizaje al que queremos llegar al final (cerca de cero), entre el aprendizaje inicial
       * elevado a la iteración actual entre el número de iteraciones totales o epochs que se llevarán a cabo. Todo esto multiplicado por
       * el aprendizaje inicial.
       */
      ritmoAprendizaje = ritmoAprendizaje0 * Math.pow(((aprendizajeFinal+0.0)/ritmoAprendizaje0), ((t+0.0)/maxEpochs));
   }

   public void actualizarRadioVecindad(int t, int maxEpochs) {
      /* El nuevo radio de vecindad es igual al radio que se quiere alcanzar al final (igual a 1, para actualizar solo las neuronas adyacentes)
       * menos el radio de vecindad original; esto multiplicado por la iteraciónn actual entre el número de iteraciones máximas, y finalmente el
       * producto sumado por el radio de vecindad original.
       */
      int radioFinal = 1;
```

```
    vecindad = (int) Math.round(vecindad0 + ((radioFinal - vecindad0) * ((t+0.0)/maxEpochs)));
}

public ArrayList<int[][]> getNuevasCoords() {
    ArrayList<int[][]> al = new ArrayList<int[][]>();

    int i = 0;
    while(i < capaSalida.neuronas.size()) {
        int[][] arr = { {capaSalida.neuronas.get(i).getX(), capaSalida.neuronas.get(i).getY()} };
        al.add(arr);
        i++;
    }

    return al;
}

public ArrayList<int[][]> getFinalCoords() {
    ArrayList<int[][]> al = new ArrayList<int[][]>();
    int i = 0;
    while(i < capaSalida.neuronas.size()) {
        if(capaSalida.neuronas.get(i).isMoved()) {

            for(int r = 0; r < Mapa.ciudades.length; r++) {
                int xTotal = Math.abs(Mapa.ciudades[r].getX() - capaSalida.neuronas.get(i).getX());
                int yTotal = Math.abs(Mapa.ciudades[r].getY() - capaSalida.neuronas.get(i).getY());

                // Si la distancia de la neurona a la ciudad actual es menos a 5 pixeles a la redonda se setea
                if(xTotal <= 5 && yTotal <= 5) {
                    capaSalida.neuronas.get(i).setX(Mapa.ciudades[r].getX());
                    capaSalida.neuronas.get(i).setY(Mapa.ciudades[r].getY());
                    int[][] arr = { {Mapa.ciudades[r].getX(), Mapa.ciudades[r].getY()} };
                    if(!al.contains(arr))
                        al.add(arr);
                }
            }
        }
        i++;
    }

    return al;
}

public double getRitmoAprendizaje() {
    return ritmoAprendizaje;
}

public int setVecindad() {
    return vecindad;
}

/* Checa todas las neuronas establecidas en alguna ciudad, si alguna concuerda con las coordenadas de entrada, regresa
 * true, sino false. Al regresar false, permite que se actualice la capa de entrada. Esto para que algunas neuronas
 * no empiecen a acercarse a una ciudad que ya tengan neurona y entonces se acerquen a otra. Esto ayuda a evitar
 * que una neurona se quede justo en medio de dos ciudades.
 */
public boolean hasNeuron(int x, int y) {
    boolean has = false;
    int i = 0;
    while(i < capaSalida.neuronas.size()) {
```

```
        Neurona neuronaActual = capaSalida.neuronas.get(i);
        if(
              neuronaActual.isMoved() &&
              neuronaActual.getX() == x &&
              neuronaActual.getY() == y
        ) {
           has = true;
           break;
        }

        i++;
      }
      return has;
   }
}
```