UNIVERSIDAD NACIONAL DE LA PLATA

FACULTAD DE INFORMÁTICA

TESINA DE LICENCIATURA

Título: Controlador robótico obtenido a través una metaheurística de población variable

Autores: Franco Ronchetti

Director: Prof. Lic. Laura Lanzarini

Codirector: Prof. Lic. Franco Chichizola

Carrera: Licenciatura en Informática

Resumen

La Robótica Evolutiva tiene por objetivo central la obtención de controladores basados en Redes Neuronales adaptadas por evolución. Dado que estos controladores deben ser capaces de adaptarse a cambios en el entorno, resulta más adecuado utilizar metaheurísticas poblacionales en lugar de técnicas basadas en gradiente. Sin embargo, la mayoría de las soluciones existentes utilizan metaheurísticas de población fija y presentan dos grandes problemas: son proclives a la pérdida de diversidad y resultan computacionalmente costosas especialmente si el espacio de búsqueda es muy amplio y se intenta hacer una exploración completa.

Esta tesina propone una metaheurística de población variable que utiliza especiación para obtener un controlador robótico basado en una red neuronal de arquitectura mínima, con capacidad para resolver el problema de evasión de obstáculos y alcance de objetivos.

Para reducir el tiempo de cálculo se analizaron distintos aspectos relacionados con el algoritmo evolutivo como así también la paralelización de la solución utilizando herramientas actuales de procesamiento paralelo.

Palabras Claves

Robótica Evolutiva. Especiación. Población variable.

Trabajos Realizados

Se implementó un mecanismo de mutación específicamente diseñado para el controlador robótico.

Se realizaron diversas modificaciones al simulador utilizado para adaptarse al problema en cuestión.

Se utilizó un cluster formado por 12 máquinas, permitiendo obtener el fitness de los individuos en un tiempo mínimo. Esto redujo el costo computacional que posee la evaluación del desempeño del controlador.

Conclusiones

Se ha definido un método evolutivo rápido y eficaz para obtener un controlador robótico pequeño y veloz, que puede ser instalado fácilmente en un robot real. Su desempeño ha sido medido en un robot Khepera II.

Los experimentos realizados demostraron la eficacia y efectividad de los operadores genéticos utilizados, así como del criterio de especiación basado en el concepto de especies con tamaño mínimo.

Trabajos Futuros

Actualmente se está trabajando para mejorar el mecanismo de asignación de tiempos de vida con el objetivo de utilizar un único criterio para controlar el crecimiento de la población.

También se están analizando distintas estrategias que permitan incorporar el controlador obtenido a partir del método propuesto en una arquitectura más compleja.

Fecha de la presentación: Marzo 2011

Controlador robótico obtenido a través una metaheurística de población variable

Autor: A.P.U. Franco Ronchetti

Directora: Prof. Lic. Laura C. Lanzarini
Co-Director: Prof. Lic. Franco Chichizola

Tesina de Licenciatura en informática

Facultad de Informática

UNIVERSIDAD NACIONAL DE LA PLATA

27 de febrero de 2011

A Laura, Jorge y Franco.

A Ayelén.

A mi familia.

Resumen

La Robótica Evolutiva comprende a las estrategias capaces de determinar controladores para comandar robots autónomos basadas en algoritmos evolutivos.

En esta dirección, las redes neuronales evolutivas han demostrado ser una herramienta sumamente útil por su capacidad de adaptardse a entornos de información dinámicos. Es importante considerar que, para estas estructuras, el aprendizaje de comportamientos específicos a partir de técnicas convenionales basadas en el descenso por gradiente requiere conocer la respuesta correcta para cada movimiento que debe efectuar el robot a través del controlador. Esta información no Por tal motivo se recurre a la adapción por evolución ya que resuelve este problema permitiendo que el aprendizaje se lleve a cabo de una manera más eficiente y eficaz.

Dentro de las técnicas más utilizadas para obtener controladores neuronales, las metaheurísticas poblacionales ocupan un lugar importante. Sin embargo, la mayoría de las soluciones existentes utilizan metaheurísticas de población fija y presentan dos grandes problemas: son proclives a la pérdida de diversidad y resultan computacionalmente costosas especialmente si el espacio de búsqueda es muy amplio y se intenta hacer una exploración completa.

Esta tesina propone una nueva estrategia evolutiva que tiene por finalidad resolver ambos aspectos. Se trata de una metaheurística poblacional que permite variar el tamaño de la población durante el proceso adaptativo siendo cada individuo un controlador neuronal completo. También se controlan los aspectos relacionados con la preservación de la diversidad de la población así como los efectos adversos que la evolución presenta sobre la representación de cada solución. Para reducir el tiempo de cálculo se propone el uso de una arquitectura de tamaño mínimo y se analizan aspectos relacionados con el algoritmo evolutivo como así también la paralelización de la solución utilizando herramientas actuales de procesamiento paralelo.

El método propuesto fue utilizado para resolver el problema de evasión de obstáculos y alcance de objetivos.

Índice general

Capítulo 1

Problemas de optimización

En matemática e informática, la optimización intenta responder a una serie de problemas donde se busca la mejor solución de un conjunto posible. En este marco, y desde hace más de 30 años, se han desarrollado diversos métodos informáticos que intentan resolver estos problemas desde diferentes perspectivas.

En este capítulo se analizan de forma general los problemas de optimización así como también las diferentes técnicas para enfrentarlos.

1.1. Aspectos generales

Generalmente, diferentes algoritmos o técnicas se aplican dependiendo la naturaleza o complejidad del problema. Se puede clasificar a los problemas de optimización en dos grandes grupos: los que tienen solo variables discretas, o enteras; y los tienen variables continuas. Los primeros también son llamados problemas de optimización combinatoria, ya que se basan en problemas donde existe un conjunto finito de elementos y el objetivo es encontrar una permutación óptima de estos que cumpla con la especificación del problema.

Las técnicas de optimización originalmente fueron pensadas para resolver problemas combinatorios, aunque se aplican también de forma masiva a problemas con variables continuas.

Los problemas combinatorios presentan la particularidad de que siempre existe un algoritmo exacto que encuentra la solución óptima. El método no es más que la exploración exhaustiva de todas las posibles soluciones. Este algoritmo suele ser extremadamente ineficiente ya que en la mayoría de los problemas de interés en el ámbito de la optimización el tiempo que se emplearía en encontrar la mejor solución crece exponencialmente con el tamaño del problema, como se detalla a continuación.

Desde los años '70 se intentó caracterizar matemáticamente a los problemas combinatorios, lo que dio origen a dos grandes grupos: la clase de problemas *P* y la clase *NP* . La clase *P* incluye a todos los problemas que pueden ser resueltos mediante un algoritmo con tiempo polinómico [61]. Es decir que a medida que el tamaño del problema crece, el tiempo crece en función de algún polinomio. Se dice que estos problemas tienen una solución eficiente [54]. Sin embargo para una gran parte de los problemas de interés práctico científico no se conoce un algoritmo con complejidad polinómica que lo resuelva de forma exacta. En cambio, el tiempo de convergencia crece de manera exponencial. Este tipo de problemas pertenecen a la clase *NP* . Un ejemplo bien conocido de problema perteneciente a esta clase es el problema del viajante de comercio (Traveling Salesman Problem, TSP [48]).

Desde hace más de 30 años, se han desarrollado diversas técnicas y algoritmos para enfrentar los problemas de optimización, tanto para problemas combinatorios, como para todo tipo de problemas de variables continuas. Estos algoritmos se pueden dividir en dos tipos principales: exactos y aproximados.

Los algoritmos exactos garantizan encontrar la solución óptima a un problema en un tiempo acotado [54][56]. Sin embargo para muchos problemas pertenecientes a la clase *NP* , y sobre todo a *NP-hard*[22], utilizar un algoritmo exacto implicaría un tiempo computacional tan elevado que la propia vida de una persona no sería suficiente.

Dado el interés práctico que tiene la resolución de muchos problemas pertenecientes a la clase *NP* y *NP-hard* y la dificultad que existe (en cuestión de tiempo) para resolverlos, se presentan nuevas estrategias basadas en encontrar soluciones de alta calidad en un tiempo razonable. Esto es lo que hacen las técnicas aproximadas.

1.2. Heurísticas

Las heurísticas nacen con el fin de resolver problemas matemáticos pertenecientes a la clase *NP* en un tiempo notablemente inferior que el consumido con un algoritmo exacto. En pro de conseguir esta reducción de tiempo se sacrifica la solución óptima por otra considerada de buena calidad. Uno de los principales inconvenientes de las heurísticas es que al finalizar el algoritmo es muy difícil saber qué tan lejos se encuentra del óptimo la solución obtenida.

El término *heurística* proviene del vocablo griego *heuriskein* que podría traducirse como *encontrar, descubrir o hallar*. En la actualidad existen diversas definiciones del término. Una de las más precisas es la establecida por Reeves[59], que resume lo aquí planteado:

"Una heurística es una técnica que busca soluciones buenas (es decir, casi óptimas) a un costo computacional razonable, aunque sin garantizar factibilidad u optimalidad de las mismas. En algunos casos, ni siquiera puede determinarse qué tan cerca del óptimo

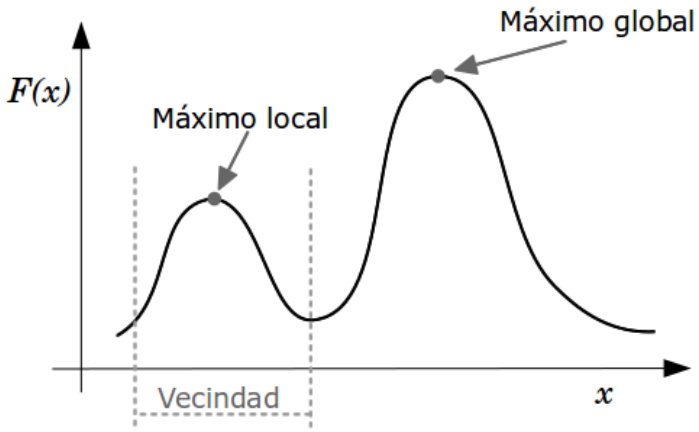

Figura 1.1: Una función objetivo con un máximo local dentro de una vecindad.

se encuentra una solución factible en particular."

Los métodos heurísticos tienen su principal limitación en su incapacidad para escapar de óptimos locales. Esto se debe a que no tienen un mecanismo que les permita seguir en la búsqueda del óptimo global en caso de quedar atrapados en uno local[1]. La figura 1.1 es un ejemplo de una función continua que posee un óptimo local cercano al óptimo global. En este caso los óptimos son el valor máximo de la función[2]. Esta función presenta una sola variable, pero en general, en problemas de optimización suele haber muchas variables (incluso varias decenas) lo cual genera una cardinalidad del espacio de búsqueda enorme.

Existen muchos métodos heurísticos de naturaleza diferente, lo que hace difícil hacer una clasificación taxativa de los mismos. No obstante, se suele distinguir entre tres tipos principales de heurísticas:

- **Heurísticas constructivas.** Suelen ser los métodos más rápidos. Parten de una solución vacía e iterativamente van creando una solución completa añadiendo componentes. Las soluciones ofrecidas suelen ser de muy baja calidad, y resulta muy difícil encontrar una buena solución, aunque siempre depende del problema particular.
- **Heurísticas de búsqueda local.** A diferencia de las anteriores, estas heurísticas comienzan con una solución completa e iterativamente van cambiando hacia otra

[1] El término *local* hace referencia a un cierto rango dentro del dominio de las variables de una función.
[2] En general un problema de optimización se basa en la búsqueda del máximo o el mínimo de una función establecida.

mejor, explorando dentro de un vecindario apropiado. El vecindario es el conjunto de soluciones a las que el proceso puede acceder utilizando un operador de movimiento. Dependiendo de este operador el vecindario puede cambiar, al igual que el espacio de búsqueda.

- **Metaheurísticas.** Estas son técnicas que utilizan a las heurísticas desde un nivel superior, como se detalla en la siguiente sección.

1.3. Metaheurísticas

Las metaheurísticas son algoritmos más inteligentes que los presentados recientemente. Intentan solventar las deficiencias de estos con técnicas más avanzadas que guían el proceso de búsqueda. El término *metaheurística* proviene de la composición de dos vocablos griegos: *heuriskein* explicado en la sección anterior y *meta* que puede ser traducido como *más allá, en un nivel superior*. Es difícil hacer una descripción unificada de las metaheurísticas, sobre todo porque muchas fueron diseñadas para un propósito específico. Sin embargo, las descripciones encontradas en la literatura hacen posible obtener ciertas características en común [24][1]:

- Las metaheurísticas son técnicas que orientan el proceso de búsqueda hacia el óptimo.
- Las metaheurísticas tienen como objetivo encontrar soluciones buenas y nunca garantizan encontrar la solución óptima.
- Los algoritmos metaheurísticos son inmensamente más rápidos que sus equivalentes exactos.
- Incorporan mecanismos “inteligentes” para escapar de máximos locales.
- Las técnicas son genéricas, y no dependen del problema específico.

Existen diversas formas de clasificar a las metaheurísticas [11]. Dependiendo de la característica observada pueden ser clasificadas en metaheurísticas inspiradas o no en procesos naturales, metaheurísticas con o sin memoria, etc. En este capítulo se detalla la clasificación más empleada, que distingue las metaheurísticas que operan sobre una única solución de las que lo hacen sobre un conjunto de soluciones posibles, o población. Según esta clasificación las metaheurísticas se dividen en basadas en trayectoria y basadas en población respectivamente [1]. La figura 1.2 muestra una clasificación de todas las técnicas para resolver problemas de optimización presentadas en este capítulo así como también las que se verán en las próximas secciones.

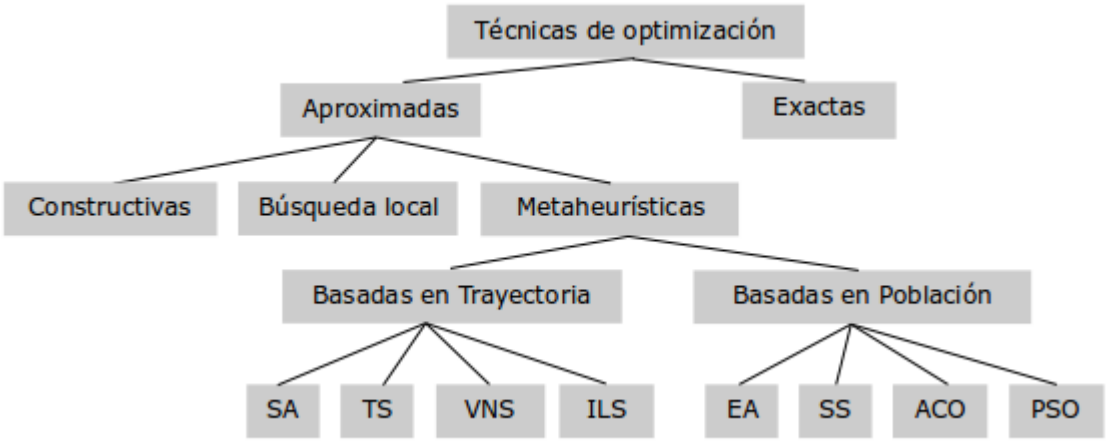

Figura 1.2: Clasificación de las diferentes técnicas de optimización.

1.3.1. Metaheurísticas basadas en trayectoria

Estas técnicas parten de un punto inicial y actualizan una única solución mediante la exploración del vecindario, formando una trayectoria en su camino de búsqueda. En general casi todos estos métodos son extensiones de procesos iterativos simples mejorados, los cuales no tienen buen desempeño por sí solos, con el agregado de técnicas para escapar de máximos locales. Generalmente el criterio de terminación es más complejo que la simple búsqueda de un máximo local. El proceso puede finalizar por diversos motivos: una cantidad de tiempo establecido; al encontrar una solución aceptable; al detectar un estancamiento del proceso; etc.

A continuación se detallan algunos de los algoritmos más importantes dentro de esta clasificación.

Enfriamiento simulado (*Simulated Annealing, SA*)

Es una de las primeras metaheurísticas [41]. Se basa en el proceso térmico natural de algunos metales y cristales, donde el material se calienta hasta alcanzar ciertos límites para luego enfriarse. En cada iteración se analiza una solución *S1* elegida al azar dentro de un vecindario apropiado a partir de la actual solución *S0*. Si *S1* es mejor que *S0*, la primera sustituye a la segunda. En caso contrario se sustituye en base a una probabilidad. Esta técnica permite escapar de máximos locales ya que se aceptan, ocacionalmente, movimientos hacia soluciones peores que la actual [1].

Búsqueda Tabú (*Tabu Search, TS*)

Esta técnica ha sido muy utilizada y reconocida para resolver problemas de optimización combinatoria, introducida por primera vez en [24]. Utiliza una estrategia exploratoria a través de un historial de búsqueda (memoria a corto plazo). La implementación del historial es mediante una *lista tabú* de soluciones, donde el método guarda las soluciones recientemente encontradas que no podrán ser usadas para los próximos movimientos (de allí su nombre). Esta memoria de búsqueda no solo le permite realizar una búsqueda eficiente evitando zonas ya reconocidas, sino que también le permite escapar de óptimos locales. En cada iteración se encuentra la mejor solución entre las permitidas y se la agrega en la *lista tabú*.

Búsqueda en vecindario variable (*Variable Neighborhood Search, VNS*)

Esta estrategia propone la búsqueda a través de diferentes estructuras de vecindarios. El algoritmo es muy genérico, con muchos grados de libertad, permitiendo variaciones y modificaciones particulares del mismo. Al comenzar el programa se define un conjunto de estructuras de vecindario y se crea una solución inicial. Luego se itera hasta conseguir una condición de fin [31][30].

Búsqueda local iterada (*Iterated Local Search, ILS*)

En cada iteración esta técnica aplica un cambio o modificación a la solución actual, lo que da lugar a una solución intermedia. A esta nueva solución se le aplica una heurística de búsqueda local para mejorarla. Por último, este nuevo óptimo puede ser aceptado o no dependiendo que pase un test de aceptación [65].

1.3.2. Metaheurísticas basadas en población

A diferencia de las metaheurísticas basadas en trayectoria, estas técnicas analizan en cada iteración un conjunto de soluciones, representadas por los individuos de la población. Si bien la forma de explorar el espacio de búsqueda resulta mucho más natural, el desempeño final del algoritmo depende en gran parte de la manipulación que se haga de la población.

A continuación se describen de forma resumida los principales algoritmos basados en población.

Algoritmos evolutivos (*Evolutionary Algorithms, EA*)

Estos algoritmos están inspirados en la evolución biológica (teoría de las especies) donde los individuos se adaptan al entorno por medio de la selección natural, o supervivencia del más apto [13]. Generalmente se comienza con una población aleatoria que es modificada y recombinada iterativamente. Análogamente a lo ocurrido en la evolución biológica, la forma de explorar el espacio de búsqueda en estos algoritmos es a través de tres operadores principales: selección, mutación y recombinación. La selección establece una estrategia para elegir los individuos y se utiliza en dos instantes diferentes del algoritmo: en el momento de decidir cuales serán las soluciones que participarán en el proceso de generación de nuevos individuos y cuando es preciso determinar cuales son las soluciones que formarán la población al pasar de una generación a la otra. La mutación permite modificar de forma aleatoria una solución actual para moverse hacia otra ligeramente desplazada. La recombinación (o cruza) genera intercambio genético de diferentes individuos para generar uno nuevo. El proceso evolutivo tiene su eje central en la selección de los individuos más aptos. Para esto, utiliza una función de aptitud (o fitness), que es la encargada de medir el desempeño del individuo en la resolución del problema. Estos algoritmos deben establecer un equilibro entre la fase de selección (aceptación de buenas soluciones) y la fase de reproducción (generación de nuevas soluciones). Existen diferentes políticas de reemplazo de los individuos, que no siempre permiten que soluciones mejores reemplacen a soluciones peores [8].

Una característica importante de estos algoritmos es cómo trata a las soluciones inviables (soluciones que no son válidas en el dominio de la función objetivo). Estas soluciones pueden ser producidas tanto por el operador de cruce como por el operador de mutación. Existen diferentes alternativas a tomar [1], que suelen recaer en: la eliminación, la penalización o la reparación de la solución inviable.

En el transcurso de los años, se han desarrollado diferentes extensiones dentro de los algoritmos evolutivos (o computación evolutiva). Estas son: la Programación Evolutiva (Evolutionary Programming, EP) [21], las Estrategias Evolutivas (Evolutionary Estrategies, ES) [58] y los Algoritmos Genéticos (Genetic Algorithms, GA) [32].

Optimización mediante cúmulos de particulas (*Particle Swarm Optimization, PSO*)

Esta técnica se basa en el comportamiento en grupo de ciertos animales, como las bandadas de aves o los cardúmenes de peces, donde cada agente toma una decisión del próximo movimiento a realizar basándose en parámetros sociales e individuales. Esta estrategia es tomada por el algoritmo para explorar el espacio de búsqueda, donde cada individuo de la población se mueve siguiendo una corriente social hacia el óptimo global [39].

El algoritmo se basa en la idea de que los individuos que conviven en una sociedad

poseen un conjunto de creencias estrechamente relacionadas [40]. En este contexto el algoritmo utiliza tres parámetros principales para realizar el próximo movimiento en cada individuo:

- El conocimiento de su entorno. Es decir su valor de aptitud, que se mide, al igual que en otras metaheurísticas, con la función objetivo o algún tipo de prueba experimental que sirva como métrica de la calidad de la solución obtenida.
- La memoria individual. Este conocimiento le indica al individuo en qué dirección se encuentra la mejor solución de las ya exploradas.
- La memoria social. Esto resulta análogo al conocimiento individual pero evaluando la respuesta del mejor individuo dentro de un vecindario. Eventualmente puede utilizarse la población completa lo que llevará a considerar al mejor global.

Optimización mediante colonias de hormigas (*Ant Colony Optimization, ACO*)

Esta metaheurística está inspirada en la forma en que las hormigas buscan su comida alrededor del hormiguero. Cada hormiga sale a buscar comida recorriendo lugares de forma aleatoria. Al encontrarla, regresa al hormiguero por el mismo camino que llegó, depositando un rastro de feromonas (componente químico). La calidad de esta sustancia, que depende en parte de la calidad del alimento o del trayecto recorrido, ayuda a las demás hormigas a que encuentren esta comida y decidan entre diferentes caminos cuál es el más adecuado. La habilidad de una hormiga individual es muy pobre, sin embargo la colonia entera de hormigas logra un comportamiento inteligente [16].

Las técnicas ACO simulan las habilidades de las hormigas para resolver diversos problemas de optimización, tales como problemas de ruteo, ordenamiento secuenciales, problemas de scheduling, etc. [18]. El comportamiento del rastro de feromonas de las hormigas es simulado artificialmente mediante un modelo probabilístico.

Búsqueda dispersa (*Scatter Search, SS*)

Esta técnica se basa en el principio de que la información sobre la calidad de un conjunto de reglas o soluciones puede ser utilizado mediante la combinación de éstas. El método consiste en la combinación de soluciones, generando nuevas con una gran diferencia respecto de las anteriores. Para esto, el método posee un *conjunto de referencia* que contiene las soluciones consideradas buenas y que pueden recombinarse. Cabe destacar que el término *buena* no se restringe necesariamente a la calidad de la solución, sino también a la diversidad que esta genera sobre las demás soluciones. Para una explicación más detallada ver [23][25].

Capítulo 2

Redes Neuronales Artificiales

Dentro del estudio de la Inteligencia Artificial pueden encontrarse dos grandes áreas de investigación: la simbólica, y la subsimbólica.

La primera se ocupa de la construcción de sistemas que tienen capacidad para realizar inferencias nuevas a partir de hechos que se conocen y afirman como verdaderos. Modelan la actividad racional mediante sistemas formales de reglas y manipulación simbólica. Un ejemplo de aplicación de esta área son los Sistemas Expertos.

Por otro lado, la perspectiva subsimbólica basa su funcionamiento en la emulación de procesos biológicos. Se trata de sistemas que extraen la información necesaria para resolver un problema de un conjunto de ejemplos, sin necesidad de indicarle las reglas necesarias para resolverlo. Ejemplos clásicos de esta rama son: las redes neuronales, los algoritmos evolutivos, las técnicas de optimización basadas en cúmulos de partículas, etc. Cada una de ellas emula un sistema biológico diferente.

Por ejemplo, las Redes Neuronales Artificiales (RNAs) buscan imitar la manera en que funciona el cerebro humano[1] [34].

Las RNAs han sido utilizadas en infinidad de aplicaciones exitosas gracias a su capacidad para aprender, reconocer patrones, generalizar y generar relaciones entre objetos propios del mundo real. Entre los usos más frecuentes pueden nombrarse: el reconocimiento de patrones en imágenes, controladores robóticos, analizadores del habla, aplicaciones médicas, explotación de bases de datos, predicción del tiempo, juegos electrónicos, etc.

[1]Desde otro punto de vista, esta clasificación puede ser análoga a la clasificación de técnicas de optimización en exactas y aproximadas, donde las RNAs son una técnica metaheurística de trayectoria.

2.1. Introducción y características generales

Las RNAs se pueden definir como sistemas de procesadores paralelos interconectados entre sí en forma de grafo dirigido, donde cada elemento del grafo es una neurona artificial. Debido a su constitución y fundamentos, las RNAs presentan diversas características semejantes al cerebro humano, lo que hace que estos sistemas ofrezcan numerosas ventajas, entre las cuales se pueden nombrar [62]:

- **Aprendizaje adaptativo**: Esta es la capacidad de aprender de la experiencia y adaptarse a los cambios que se producen en la información disponible. Es decir que la red puede modificar el conocimiento adquirido en caso de que se produzcan modificaciones en los ejemplos iniciales.
- **Auto-organización**: Consiste en la modificación de la red completa con el fin de llevar a cabo un objetivo específico. Esto ayuda a la generalización de la red.
- **Tolerancia a fallos**: A diferencia de un sistema de cómputo tradicional, un pequeño error en una red neuronal no provoca la inutilización del sistema completo, sino que puede seguir funcionando con una pequeña degradación. Las RNAs también son tolerantes a entradas de datos con ruido o incompletos.
- **Operación en tiempo real**: Con el hardware adecuado, las RNAs son el método más adecuado para procesamiento en tiempo real, debido a su modo de operar distribuidamente.
- **Fácil inserción en las tecnologías existentes**: Es sencillo obtener controladores neuronales especializados que pueden ser usados para generar sistemas de forma incremental, donde cada paso puede ser evaluado cuidadosamente corroborando que la red sirva para un propósito bien definido [45].

En el resto de esta sección se detallan los aspectos principales de las redes neuronales, así como también las arquitecturas más utilizadas.

2.1.1. La neurona artificial

Las redes neuronales artificiales pueden diferir en su arquitectura pero siempre están compuestas por la misma unidad de procesamiento: la neurona artificial. Una neurona artificial es un autómata que tiene sus fundamentos en las neuronas biológicas de los animales.

Una célula neuronal consta de tres componentes principales: las dendritas, el cuerpo neuronal y el axón. Las dendritas son las encargadas de recibir estímulos eletro-químicos desde el exterior, o desde otras neuronas y transmitirlos al interior de la célula. El cuerpo

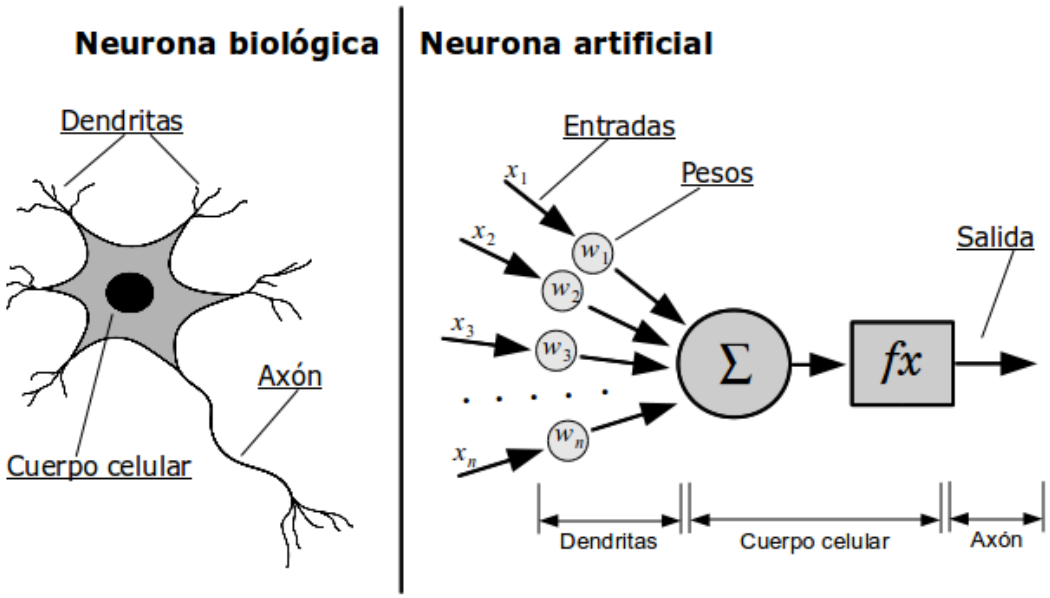

Figura 2.1: Comparación de una neurona biológica con una artificial.

celular se encarga de procesar esos estímulos y generar un impulso de salida a través del axón, que luego se conecta con otras neuronas. Las neuronas no son elementos lineales, sino que funcionan por saturación; emiten un estímulo de salida siempre y cuando la suma de todos los estímulos recibidos supere un cierto umbral [34]. El sistema nervioso humano posee miles de millones de neuronas, con un grado de interconexión muy elevado.

La neurona artificial se basa en estos conceptos. La figura 2.1 muestra una comparación entre una célula neuronal animal y una neurona artificial clásica. En la figura se pueden apreciar los principales componentes de la neurona artificial y su analogía con la biológica. La neurona artificial funciona del siguiente modo:

1. Las entradas de la red, generalmente nombradas $x_1, x_2...x_n$ son ponderadas por una serie de pesos asociados a estas, generalmente nombrados $w_1, w_2...w_n$. Estos pesos harán que cada entrada se *excite* o *inhiba* dependiendo su valor.

2. Se calcula el total del estímulo que recibe evaluando la suma de todas las entradas ponderadas a través de los pesos de los arcos correspondientes.

3. Se aplica una función de activación (o transferencia) al resultado del paso 2 y se genera un valor de salida.

En muchos modelos neuronales se utiliza un parámetro extra a las entradas llamado *umbral de activación*. Este umbral no se ve reflejado en la figura, sin embargo existe en

las neuronas biológicas, permitiendo establecer un valor de energía que debe superarse para que la neurona se active.

La función de activación a utilizar queda determinada por el tipo de problema que debe resolver la neurona. Algunas de las funciones más utilizadas son: identidad, escalón, sigmoidea y gaussiana[20].

Al cambiar los pesos de la red, ésta se comporta de diferentes formas, siendo este cambio el principal eje del aprendizaje de la neurona. Sin embargo, al considerar una red entera, este cambio debe estar equilibrado con los cambios en los pesos de otras neuronas, e incluso quizá con cambios en la arquitectura de la red.

La siguiente ecuación muestra cómo se calcula la salida de una neurona típica:

$$salida = f(\sum_{i=1}^{n} w_i x_i) \tag{2.1}$$

Donde f es la función de activación, y la sumatoria podría ser reemplazada por alguna otra función de propagación. La función de activación, y por tanto la salida de la neurona, puede ser un valor discreto o continuo, dependiendo del problema a resolver. Por ejemplo, si se está construyendo una neurona que actúe como clasificador la función de activación podría tomar los valores $0, 1$ indicando que un patrón de entrada pertenece o no a dicha clasificación.

2.1.2. Arquitectura básica de una red

Una red neuronal es un conjunto de neuronas interconectadas con algún criterio particular. Esta estructura puede verse, desde un punto de vista matemático, como un grafo dirigido y ponderado donde cada nodo es una neurona artificial y los arcos son las conexiones sinápticas[2] entre las neuronas. Esto quiere decir que la información siempre viaja en un sentido, desde la neurona que origina el impulso, hacia la que lo recibe.

Las redes neuronales suelen dividirse en capas, como lo muestra la figura 2.2. Se suelen identificar tres capas principales:

- Capa de entrada. las neuronas de esta capa reciben los datos que se quieren procesar.
- Capa oculta. También llamada capa intermedia, recibe estímulos desde la capa de entrada y los propaga hacia la siguiente capa. Puede haber varias capas ocultas.
- Capa de salida. Su función es proporcionar la respuesta de toda la red.

La figura muestra una red con solo 3 capas, las cuales están totalmente interconectadas, es decir que existe un arco desde cada neurona de una capa hacia toda neurona de la capa siguiente.

[2]En biología, la sinapsis es una relación de contacto entre dos neuronas.

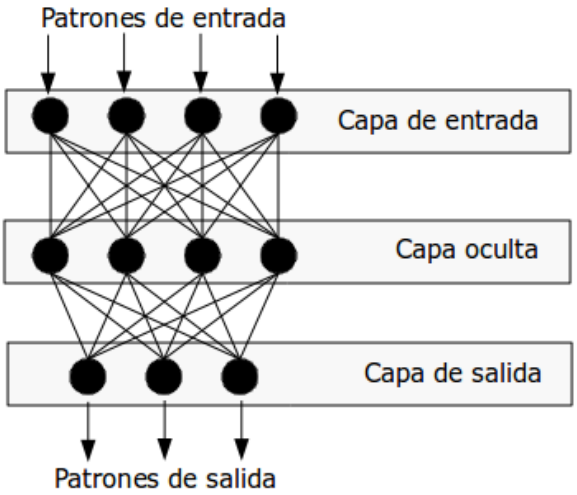

Figura 2.2: Una red neuronal con tres capas totalmente interconectadas.

Diferentes clasificaciones de las RNAs pueden realizarse dependiendo el criterio elegido. A continuación se listan algunos de los aspectos más utilizados para clasificarlas [50]:

- **Topología de la red.** De acuerdo al número y disposición de las neuronas de la red, esta puede ser monocapa o multicapa. Las redes multicapa pueden resolver problemas más complejos, a cambio de una adaptación más costosa, y mayor dificultad para escapar de óptimos locales. De acuerdo a si las redes presentan ciclos en sus conexiones, estas pueden ser redes *feedforward* (unidireccionales) o recurrentes (ver sección 2.4).

- **Tipo de adaptación.** Según la forma en que la red es adaptada puede distinguirse entre adaptación por aprendizaje o adaptación por evolución. Dentro de los algoritmos de aprendizaje (o entrenamiento) se pueden encontrar los supervisados y los no supervisados (ver sección 2.2).

- **Tipo de comportamiento.** Luego de ser entrenadas, las redes sin memoria se comportan como combinadores no lineales. En cambio, las redes que poseen algún tipo de memoria logran comportarse como un sistema dinámico no lineal, lo que agrega mucha complejidad a su comportamiento. En este sentido, las redes pueden ser clasificadas como estáticas o dinámicas respectivamente.

2.2. Técnicas de adaptación

A diferencia de los algoritmos convencionales en computación, las RNAs no se programan sino que necesitan ser adaptadas. Una de las etapas más importantes además de la decisión de la arquitectura de la red, es la adaptación por parte de esta para poder llevar a cabo la función que se desee.

Existen diferentes métodos de adaptación tanto desde un punto de vista estructural como desde la forma de aprender que tiene la red [20]. En general la arquitectura de la red se define fija y el aprendizaje se basa en la modificación gradual de los pesos de la red. Sin embargo, muchos algoritmos cambian la estructura de la red generando nuevas neuronas, eliminando otras, y creando nuevas conexiones[3]. En cualquier caso, al comenzar el proceso de adaptación se crean los pesos de la red con valores aleatorios o nulos.

Una clasificación que se puede hacer de los métodos de aprendizaje consiste en considerar si la red puede aprender (modificarse) durante su funcionamiento habitual o solo puede aprender durante una etapa de entrenamiento específica. De este modo, puede diferenciarse entre aprendizaje *on-line* o aprendizaje *off-line* respectivamente. En general, la mayoría de las técnicas de entrenamiento utilizan aprendizaje *off-line*, en donde la red es entrenada en una etapa previa a su uso, que puede llevar mucho tiempo.

Si bien la mayor parte de la bibliografía encontrada hoy en día ([34][20][50]) sobre RNAs indica que estas solo se adaptan mediante el entrenamiento por patrones de ejemplo, desde hace varios años se han utilizado algoritmos evolutivos para la adaptación de redes neuronales, siendo un método de gran interés para la comunidad científica [45][68].

En el resto de la sección se detallan estos métodos de adaptación de las RNAs.

2.2.1. Adaptación por entrenamiento

La adaptación por entrenamiento se basa en presentar a la red una serie de ejemplos con el fin de que esta logre aproximar algún tipo de función, y de esta forma "aprenda", por ejemplo, a caracterizar cierto tipo de patrones. El proceso se basa en presentar los ejemplo a la red e iterativamente modificar de modo gradual los pesos (como ya se dijo, también existen algoritmos que modifican la estructura de la red, pero son menos utilizados). Desde el punto de vista de los patrones de ejemplo, estos deben tener dos características principales: deben ser significativos y representativos [34]. Es decir que deben ser tan numerosos como para que la red logre adaptarse, y deben representar adecuadamente pero de forma acotada la totalidad de los patrones. De

[3]Mediante la modificación los pesos de la red también pueden crearse y destruirse conexiones, ya que en general, se considera que una conexión no existe cuando su peso es cero.

ser así, la red logrará generalizar y clasificar adecuadamente patrones futuros, que no estaban en los de muestra. Por ejemplo, si se quiere entrenar una red para que reconozca rostros de personas, las imágenes de muestra deberán contemplar diversas personas, con diferentes rasgos y diferentes posturas. De no utilizar patrones de cierto grupo de rostros, posiblemente la red no se adapte correctamente.

En términos matemáticos, las técnicas de aprendizaje tratan de estimar una función multivariante desconocida mediante el empleo de patrones de ejemplo. Para esto utilizan un cierto algoritmo de aproximación.

Existen diversos algoritmos de aprendizaje, pero en general se basan en la misma idea: presentar los ejemplos a la red, observar la salida de la red, modificar los pesos, y verificar si se cumplió algún criterio de convergencia. Este criterio puede ser un número fijo de ciclos, un nivel mínimo de error alcanzado, una modificación de pesos irrelevante, etc.

Existen dos métodos básicos de aprendizaje: el supervisado y el no supervisado. La diferencia principal entre ambos radica en la presencia o no de un agente externo (maestro) que regula el entrenamiento.

Aprendizaje supervisado

En este tipo de entrenamiento existe un supervisor o maestro que conoce la salida que debería tener la red para un cierto patrón. De esta forma se ajustan los pesos iterativamente en base a dicha información. Existen tres tipos principales de aprendizaje supervisado [20]:

- **Aprendizaje por corrección de error.** Este es el tipo de aprendizaje más utilizado en la práctica. Se basa en ajustar los pesos de la red en base a la diferencia entre los valores que se esperan y los valores retornados por la red. Algunos de los usos más conocidos son la regla *Delta*, y el algoritmo *backpropagation*. Estos se ven con más detalle en la sección 2.3.

- **Aprendizaje por refuerzo.** A diferencia de la técnica anterior, en este caso no se conocen los valores exactos de la salida deseada para un patrón. En cambio, sólo se conoce si la red respondió correctamente o no, convirtiéndose el maestro en un crítico que simplemente responde *sí* o *no*. Los pesos, entonces, son ajustados en función de esta información y en base a un mecanismo de probabilidades.

- **Aprendizaje estocástico.** Se basa en la modificación aleatoria de los pesos de la red, evaluando su efecto en base a la salida deseada y a una distribución de probabilidad.

Aprendizaje no supervisado

En el aprendizaje no supervisado, o también llamado auto-organizado, la red es entrenada únicamente con los patrones de ejemplo y sin información o control externo sobre la salida esperada para cada patrón. En este caso la red modifica los pesos a partir de información interna que el algoritmo genera. Aquí, la red intenta reconocer rasgos significativos, regularidades o cualquier tipo de patrón en los datos de entrada.

2.2.2. Adaptación por neuroevolución

La neuroevolución es una técnica relativamente nueva, que se basa en la adaptación de las redes neuronales a través del uso de algoritmos evolutivos. Existen diferentes trabajos que utilizan distintos tipos de algoritmos [64][68][45], entre los que se pueden encontrar: Algoritmos Genéticos, Estrategias Evolutivas, etc. Sin importar el algoritmo utilizado, el proceso se basa en la adaptación iterativa de la red neuronal para resolver un problema particular. Aquí, no se presenta una serie de patrones de ejemplo a la red, sino que esta cambia estocásticamente y es evaluada con alguna función de aptitud que indique qué tan buena es la salida otorgada. Luego, el proceso es guiado hacia la búsqueda del óptimo global con algún mecanismo propio de los algoritmos evolutivos.

Esta técnica es muy útil para problemas donde no es fácil utilizar un mecanismo de aprendizaje supervisado, o no se sabe con exactitud la salida esperada para un cierto patrón. Algunas de las áreas de aplicación más usuales son los videojuegos y la realización de controladores robóticos, como el desarrollado en esta tesina.

2.3. Modelos de redes neuronales

Existen diversos modelos de redes neuronales que se fueron desarrollando a lo largo de la historia. Se presentan a continuación algunos de los modelos más utilizados en la práctica, dejando de lado las redes recurrentes, las cuales se analizan con más detalle en la sección 2.4.

Uno de los primeros modelos que pueden citarse como red neuronal son las células de McCulloch-Pitts [34]. Estas datan del año 1943 donde se modelizaba una estructura simplificada de la neurona del cerebro humano, considerándolas como dispositivos con sólo dos estados posibles: encendido (1) o apagado (0). Ya que las células están conectadas a otras como ellas, sus entradas también son binarias. El modelo resultó muy potente, pues cualquier estructura se podía modelizar con él. Es sencillo construir operadores lógicos elementales como el NOT, el AND y el OR, y luego construir modularmente un sistema como lo hace un computador convencional. El problema radica en que para sistemas grandes y sofisticados el número de células necesarias es tan elevado

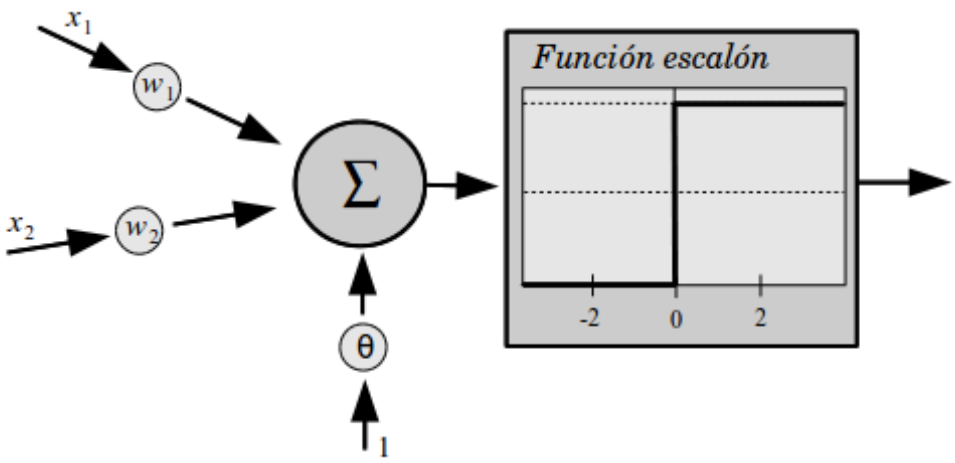

Figura 2.3: Esquema de un Perceptrón típico con dos entradas.

que resulta imposible en la práctica.

2.3.1. Perceptrón

Este modelo data de la década de 1950. Fue pensado como un clasificador lineal, que a partir de una serie de patrones de ejemplo de dos clases diferentes es capaz de encontrar el hiperplano que hace de frontera entre ellas. Una vez entrenado puede reconocer de forma automática a qué clase pertenece un patrón particular. El sistema posee sólo dos capas: una de entrada que sólo repite los patrones de ejemplo[4], y una de salida que procesa la información.

La figura 2.3 muestra un esquema de un Perceptrón típico con solo dos entradas. La neurona contiene otra entrada, llamada umbral denotado por θ que ayuda como factor de combinación para la salida. Para calcular la salida, primero se suman todos los pesos ponderados y se agrega el umbral, de la siguiente forma:

$$y' = \sum_{i=1}^{n} w_i x_i + \theta \tag{2.2}$$

El umbral puede verse como una entrada extra que siempre tiene el valor 1, y donde su peso es θ. Siendo así la ecuación puede reescribirse como:

$$y' = \sum_{i=0}^{n} w_i x_i \tag{2.3}$$

[4]Debido a esto, es habitual también que se diga que el Perceptrón en monocapa.

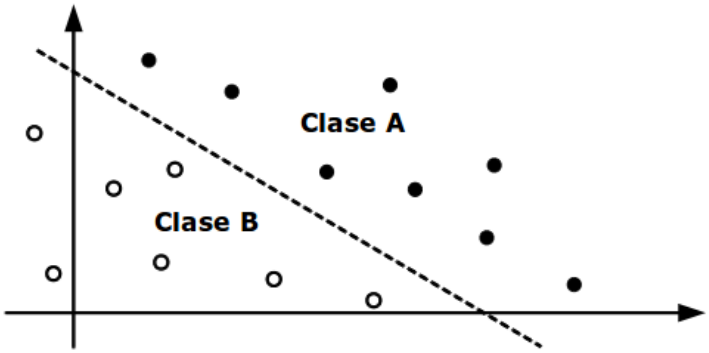

Figura 2.4: Clasificación de dos clases con un perceptrón de dos entradas.

Notar que ahora i comienza desde 0, ya que se ha agregado θ como una entrada más.

En segundo lugar se aplica la función de transferencia que es una función escalón de la siguiente forma:

$$y = f(y') = \begin{cases} 1 & \text{Si } y' > 0 \\ 0 & \text{caso contrario} \end{cases} \tag{2.4}$$

Esta función es binaria y resulta de gran utilidad a la hora de clasificar. Suponiendo solo dos clases (ya que ese es el poder de un solo Perceptrón) el algoritmo que clasifica es el siguiente:

- Si la salida de la red es 1, entonces el patrón pertenece a la clase A.
- Si la salida de la red es 0, entonces el patrón pertenece a la clase B.

En la figura 2.4 se muestra un ejemplo de cómo podría llevar a cabo esta clasificación el perceptrón de la figura 2.3 antes descrito. Ya que esta neurona tiene solo dos entradas la función a aproximar es de dos dimensiones. Los puntos sólidos pertenecen a una clase, y los puntos huecos pertenecen a la otra.

El algoritmo de aprendizaje utilizado es supervisado (ver sección 2.2.1). Iterativamente se presentan los ejemplos a la red y todos los pesos de cada neurona son ajustados de la siguiente forma:

$$w_i' = w_i + \alpha(\delta - y)x_i \qquad i = 1..n \tag{2.5}$$

donde:

- w_i' y w_i representan el valor del peso para la entrada i en el instante $t + 1$ y t respectivamente.
- x_i es el valor de la entrada i.

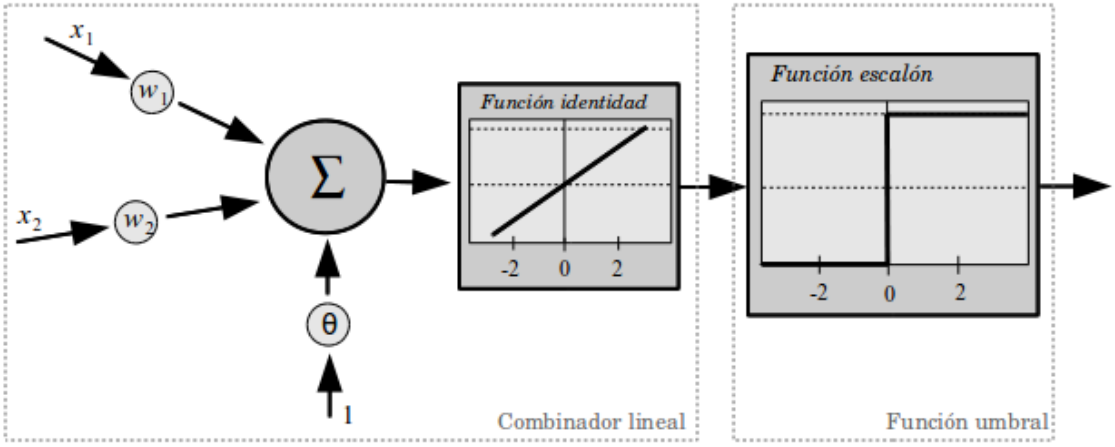

Figura 2.5: Esquema de un Adaline típico con dos entradas.

- α es una constante que se define al comienzo del algoritmo y representa la velocidad de aprendizaje. Su valor pertenece al intervalo $(0, 1]$ e indica la magnitud de las modificaciones que se realizan sobre el vector de pesos w. Si toma un valor muy pequeño, las modicaciones son chicas y la cantidad de iteraciones, necesarias para que la neurona se adapte, se incrementan considerablemente. Por el contrario, si toma un valor muy cercano a 1, las modificaciones son muy grandes y puede ocasionar que el vector w nunca se estabilice.
- δ es la salida esperada para ese patrón.
- y es la salida de la neurona para ese patrón.

El proceso de aprendizaje continua hasta que se alcanza algún criterio de terminación.

2.3.2. ADALINE

El ADALINE (ADAptative LInear NEuron[5]) es un modelo neuronal que data de 1960, muy parecido al Perceptrón pero con algunas notables diferencias [20][34]. La figura 2.5 muestra un esquema de una neurona con dos entradas. Aquí se aprecian los dos componentes principales de este sistema: un combinador lineal, que se detalla a continuación, y una función umbral similar a la utilizada en el Perceptrón.

Una de las principales diferencias con el Perceptrón radica en la técnica de entrenamiento de la neurona. Este entrenamiento se aplica sólo al combinador lineal, con

[5]Tiempo después, su autor cambió este nombre a ADAptative LINear Element.

lo cual la salida del sistema es real. Dicha salida puede calcularse como:

$$y = f(\sum_{i=0}^{n} w_i x_i) = \sum_{i=0}^{n} w_i x_i \tag{2.6}$$

En el Perceptrón, a medida que se van presentando los patrones se va ajustando cada vez mejor la función discriminante. Esto produce que este sistema sea muy dependiente del orden en que se presentan los patrones a la red. A diferencia de esto, el Combinador Lineal evalúa el error cometido de forma global y utiliza para esto una función específica. Esta función es el error cuadrático medio definido como:

$$E = \frac{1}{L} \sum_{j=1}^{L} \left(\delta^j - y^j\right)^2 \tag{2.7}$$

donde L es la cantidad de patrones de entrenamiento.

Al ser lineal la salida de la neurona, esta función es cuadrática con respecto a los pesos de la red y define una superficie en forma de paraboloide que posee un único mínimo global, pudiéndose demostrar que siempre es posible alcanzarlo sin importar la configuración inicial de los pesos de la red.

De esta forma, el combinador lineal intenta minimizar este error mediante la técnica del descenso del gradiente. Iterativamente se actualizan los pesos de la red de una forma similar a la hecha por el Perceptrón:

$$w_i' = w_i + \alpha(\delta - y)x_i \tag{2.8}$$

Si bien esta ecuación resulta muy similar a la ecuación 2.5 del Perceptrón, aquí y es la entrada neta (es decir la suma ponderada de todas las entradas). Esta regla es conocida con el nombre de *regla Delta* que no es más que una generalización de la regla del Perceptrón para variables reales.

La neurona ADALINE surge al aplicar una función umbral bipolar a la salida del combinador lineal. Para esto podría utilizarse el signo de la salida de la red en la ecuación 2.6. De esta forma la red, formada por una única neurona, se convierte en un clasificador lineal.

Si bien el Combinador Lineal fue muy utilizado para filtrado de ruidos en líneas telefónicas, la aplicación del ADALINE para resolver problemas de clasificación en dos clases no brindó resultados superiores al Perceptrón.

Es importante considerar que ambas neuronas, el Perceptrón y el ADALINE, sólo son capaces de determinar, a través de los pesos w, una función discriminante lineal. Luego, la ubicación de cada patrón con respecto al hiperplano identifica la clase a la cual pertenece.

Este enfoque presenta dos problemas. En primer lugar, existen problemas de clasificación en dos clases que no son linealmente separables como por ejemplo el problema del XOR. En segundo lugar, habitualmente deben resolverse problemas de clasificación de más de dos clases que suelen ser no lineales.

2.3.3. Perceptrón multicapa

El perceptrón multicapa surge como una solución a la limitación que presentan los modelos de RNAs antes vistos. De hecho, diferentes autores [12][33] han demostrado que el Perceptrón multicapa puede aproximar cualquier función continua sobre un espacio compacto R^n como una nueva clase de función para aproximar o interpolar relaciones no lineales entre datos de entrada y salida. La capacidad de este modelo para aprender a partir de un conjunto de patrones iniciales hace que sea un modelo muy versátil y adecuado para la resolución de problemas reales (aunque no implique que sean los mejores aproximadores universales).

Arquitectura

La arquitectura del Perceptrón multicapa está organizada del mismo modo que el presentado en la figura 2.2 en la sección 2.1.2. Se basa en un conjunto de capas con diversas neuronas cada una. Como se ha mencionado con anterioridad, se dintiguen tres tipos de capas: capa de entrada, capas ocultas y capa de salida. En general la capa de entrada solo recibe los datos de los patrones de ejemplo y los replica a las siguientes capas sin realizar ningún procesamiento con estos. Las capas ocultas por lo general realizan un procesamiento no lineal de los datos enviados por la capa de entrada y propagan los resultados a las siguientes capas, que pueden ser otras capas ocultas. Por último, la capa de salida es la última en procesar datos emitiendo en su salida los valores de salida de la red.

La arquitectura del Perceptrón multicapa es una arquitectura *feedforward*, o "alimentada hacia adelante", ya que todas sus conexiones siempre se dirigen desde una capa hacia la siguiente, siguiendo un orden de prioridad desde la capa de entrada hasta la de salida.

Un problema importante que existe en la actualidad, y que sigue siendo material de diversos estudios, es que no se conoce un método que establezca la arquitectura específica a utilizar dado un determinado problema. Es decir, no se sabe con exactitud cuántas capas son necesarias, ni cuantas neuronas por capa, ni su grado de interconexión, ni tampoco las funciones de transferencia de cada neurona. En general, la cantidad de neuronas de entrada y de salida vienen dadas por la naturaleza del problema. Sin embargo existen diversos problemas en que estas variables no se conocen con exactitud.

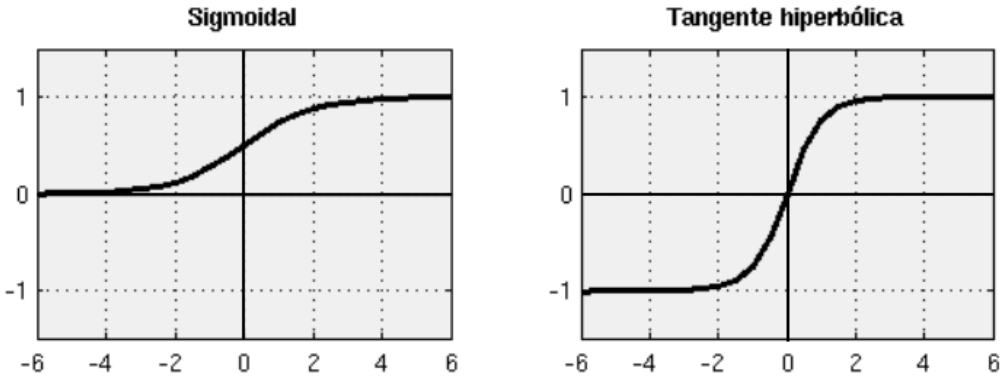

Figura 2.6: Funciones sigmoidal y tangente hiperbólica.

El funcionamiento de la red se efectúa evaluando cada neurona de forma independiente del mismo modo que el perceptrón clásico. La salida de cada neurona es utilizada como entrada para otra neurona, siempre y cuando estén conectadas. En este tipo de arquitecturas se suelen utilizar con frecuencia dos tipo de funciones de transferencia: la función sigmoidal, y la tangente hiperbólica [34]. La figura 2.6 muestra las gráficas de estas funciones.

En general la función de transferencia en el Perceptrón multicapa es común a todas las neuronas y es elegida por el diseñador dependiendo el problema a resolver. Ambas funciones están relacionadas mediante la expresión $f_1 = 2f_2 - 1$, siendo f_1 la tangente hiperbólica y f_2 la función sigmoidal. De este modo la elección de una u otra dependerá del recorrido de interés.

En ocasiones, dependiendo el problema, las neuronas de la capa de salida utilizan una función de transferencia diferente del resto de las neuronas, siendo las más usuales la función identidad y la función escalón.

Algoritmo de retropropagación

El aprendizaje utilizado para entrenar este tipo de redes es un aprendizaje supervisado por corrección de error, llamado algoritmo de retropropagación o *backpropagation*. El método es una generalización de la regla Delta vista en la sección 2.3.2 para el Adaline. Al igual que para la regla Delta se necesitan una serie de patrones de ejemplo y sus respectivos valores esperados. El aprendizaje de la red se formula como un problema de búsqueda del mínimo global dentro de la superficie del error.

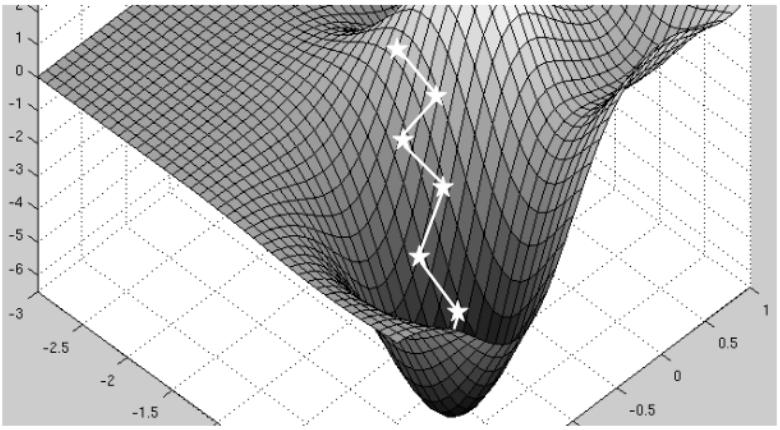

Figura 2.7: Técnica del descenso del gradiente.

La idea es minimizar el error para cada patrón, p , definido por:

$$E_p = \frac{1}{2} \sum_{k=1}^{m} \left(\delta_p^k - y_p^k \right)^2 \tag{2.9}$$

donde m es la cantidad de neuronas en la capa de salida de la red.

Esta ecuación resulta similar a la ecuación 2.7 del Adaline. Esto se debe a que, como ya se dijo, este método es una generalización del presentado anteriormente.

Para llegar al mínimo global en la superficie del error se utiliza una técnica llamada *técnica del descenso del gradiente* o también *técnica del gradiente estocástico*, la cual se basa en la sucesiva corrección de los pesos de la red por cada patrón de ejemplo. De esta forma, cada parámetro w de la red se actualiza para cada patrón p introducido, en base a la siguiente expresión:

$$w' = w - \alpha \frac{\partial E_p}{\partial w} \tag{2.10}$$

donde E_p es el error cometido para ese patrón definido por la ecuación 2.9, y α es el factor de aprendizaje, o factor de aceleración de la red.

Este algoritmo del gradiente es aplicado en el Perceptrón multicapa eficientemente a través de los distintos niveles de capas que posee la arquitectura. De aquí surge el nombre "algoritmo de propagación hacia atrás", ya que el error cometido en la salida de la red es transmitido desde la capa de salida hacia las capas ocultas, actualizando los respectivos pesos. La figura 2.7 muestra con un ejemplo el modo de funcionamiento de este método.

Un aspecto clave en el entrenamiento de una red neuronal es la correcta elección de los patrones de ejemplo. Si los patrones no son lo suficientemente representativos, la red no podrá generalizar correctamente. No obstante, el hecho de que una red

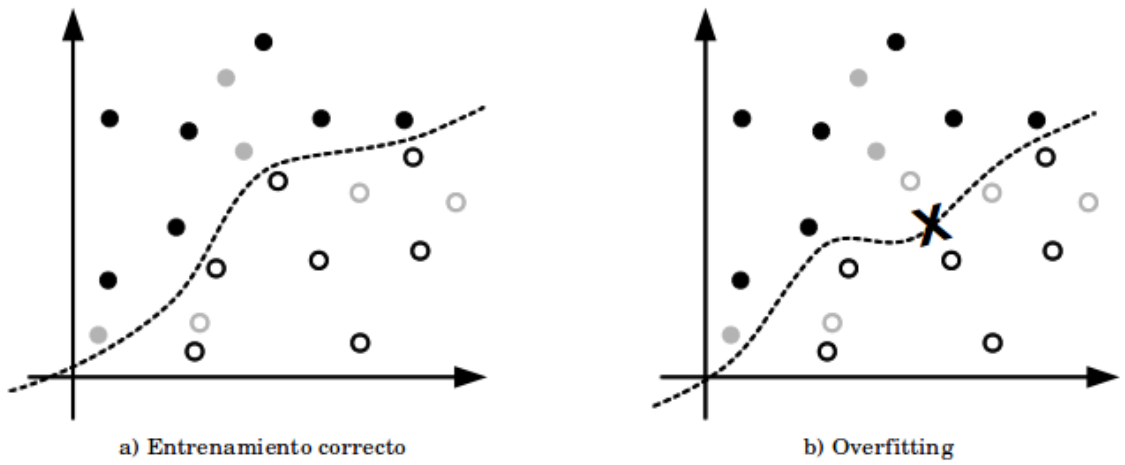

Figura 2.8: Ejemplo de overfitting en el entrenamiento de una red.

pueda aproximarse correctamente a una determinada función dependerá también de la arquitectura elegida así como de las funciones de transferencia utilizadas. Si todos estos aspectos no se encuentran en armonía la red podría entrenarse incorrectamente y generar un "sobreentrenamiento" u *overfitting* [50][55]. Esto causa que la red aprenda características específicas de los patrones, incluyendo malos experimentos o ruido en las muestras. La figura 2.8 muestra un ejemplo de este suceso. En el ejemplo se intenta encontrar una función que clasifique correctamente dos clases. Los puntos sólidos pertenecen a una clase y los puntos huecos pertenecen a otra. Para cada clase, los puntos negros indican los patrones usados como entranamiento, y los puntos grises indican el resto de los patrones que componen el grupo. En la figura 2.8-a puede verse una correcta adaptación de la red, ya que logra generalizar eficientemente incluyendo los patrones no introducidos en el entrenamiento. En la figura 2.8-b se utilizan otros patrones de entrenamiento lo que causa una incorrecta aproximación de la red a la función que clasifica las clases.

2.3.4. Otros modelos

Existen diversos modelos de RNAs desarrollados en los últimos años de los cuales no se hará mayor mención aquí. Entre estos podemos encontrar:

Las redes de base radial, o RBF [34], las cuales tienen un alto rendimiento en la fase de entrenamiento de la red. Además, estas redes presentan menor dependencia del orden en que se introducen los patrones de ejemplo que las redes tradicionales como el Perceptrón multicapa. Han sido muy usadas para reconocimiento del habla,

sistemas de identificación automática, etc.

Las redes competitivas como las redes SOM [42]. Estas están basadas en evidencia biológica sobre el reconocimiento adaptativo de los animales, gracias a las neuronas en su cerebro. Utilizan algoritmos no supervisados y competitivos. Son muy usadas para reconociemiento de patrones ópticos, reconocimiento de figuras, *blurring*, diagnóstico de la voz, monitorización de procesos, predicción de series temporales, control de brazos robóticos, telecomunicaciones, etc.

Las redes de resonancia adaptativa, o ART [34][20], son redes que se caracterizan por abordar con eficiencia dos temas centrales, a diferencia de otros modelos que no lo logran: la plasticidad y la estabilidad del aprendizaje. Esto significa que mientras que una red puede clasificar correctamente los patrones presentados como ejemplos y los que lo rodean en la misma clase, puede además adaptarse a la presencia de nuevas clases de patrones. En general, los demás modelos logran abordar correctamente uno de estos conceptos, pero no los dos juntos. Las ART utilizan una arquitectura de tres capas y con conexiones recurrentes. Se han utilizado estas redes para aplicaciones como reconocimiento de caracteres y rostros, reconocimiento de olores ("nariz electrónica"), reconocimiento de señales analógicas, etc.

2.4. Redes neuronales recurrentes

Este tipo de redes neuronales agrupa a todos los modelos que utilicen, a diferencia de los modelos vistos hasta ahora, bucles o ciclos en sus conexiones [34]. Estas conexiones recurrentes pueden ser desde una neurona hacia ella misma, desde una neurona a otra de una capa anterior, desde una neurona hacia otra en su misma capa, o cualquier disposición que uno pueda imaginarse. La figura 2.9 muestra dos ejemplos de redes recurrente.

Al tener conexiones de este tipo, las redes recurrentes ya no dependen sólo de los patrones de entrada que se le presentan, sino que ahora es necesario agregar la variable tiempo. La red responderá de dos formas diferentes para el mismo patrón en dos momentos distintos. Dada esta característica, se suele clasificar a estas redes como dinámicas, ya que pueden tomar decisiones específicas a lo largo del tiempo.

Dentro de los diferentes modelos de redes neuronales recurrentes pueden citarse dos modos de entrenamiento de la red:

- Entrenamiento de la red hasta alcanzar un punto estable. Es decir, lo mismo que se hace para los demás modelos.

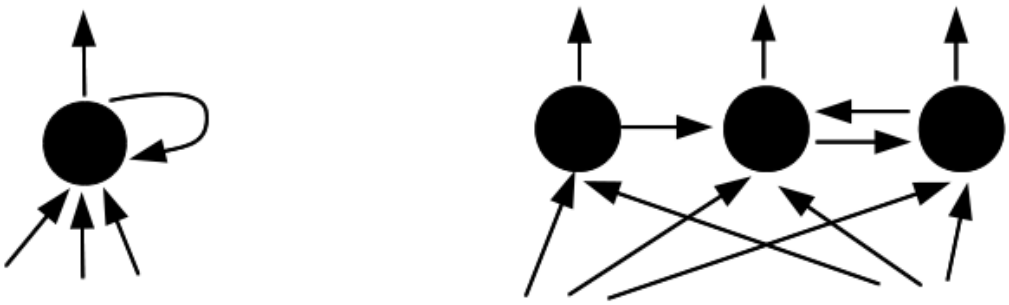

Figura 2.9: Ejemplos de redes con conexiones recurrentes.

- Entrenamiento en modo continuo. Esto implica un constante aprendizaje, generalmente en tiempo real, de la red.

El comportamiento dinámico de las redes recurrentes facilita el tratamiento de problemas en donde el tiempo es una variable crucial. Es decir que se aplican perfectamente a problemas donde un patrón particular necesita ser tratado diferente dependiendo del instante de tiempo en que se lo procese.

Algunos modelos neuronales recurrentes, como son las redes de Hopfield y la máquina de Bolzman comparten ciertas características similares, tales como: elementos de proceso no lineales, y conexiones sinápticas simétricas [4].

Las redes Hopfield son de las más conocidas en el área de redes recurrentes, aunque se aplican a patrones estáticos, donde no interviene la variable tiempo. Tienen sus raíces en los principios físicos de la mecánica estadística. Se trata de una red de memoria asociativa que permite recuperar patrones almacenados a partir de información completa o con ruido. Utiliza una arquitectura de neuronas binarias conectadas de forma completa.

La máquina de Bolzman es una generalización de la red de Hopfield que incluye neuronas ocultas. Para su funcionamiento, estas redes utilizan un algoritmo de *enfriamiento simulado* (ver capítulo 1).

Si bien las redes recurrentes presentan grandes complicaciones, sobre todo en su etapa de aprendizaje, han sido utilizadas exitosamente en diferentes tipos de problemas, tales como modelización neurológica, tareas lingüísticas, reconocimiento de palabras y fonemas, controladores robóticos, etc.

Capítulo 3

Algoritmos Evolutivos

La teoría evolucionista, a diferencia de la creacionista[1], propone un desarrollo de toda especie sobre la tierra, condicionado por su adaptación al entorno. De este modo, por medio de la *supervivencia del más apto*, cada ser vivo intenta reproducirse y transmite su información genética a sus descendientes, los cuales producen pequeños cambios en sus cromosomas y de esta forma cambian su adaptación al medio. Los individuos más aptos sobreviven en base a lo que se denomina la ley de la *selección natural*.

Los algoritmos evolutivos son técnicas metaheurísticas inspiradas en estos principios. Existe un "entorno" o "medio ambiente", el cual está representado por la superficie de alguna función multivariante que se desea aproximar o maximizar/minimizar. Existe una población de individuos, los cuales representan posibles soluciones (o puntos dentro de la función). Los individuos se reproducen y transmiten su información a la nueva generación, mientras operadores de cruce y mutación la modifican, generando individuos que pueden resultar mejor adaptados.

Se han aplicado diferentes técnicas evolutivas a lo largo de los años a diferentes problemas con resultados exitosos. Este capítulo describe de forma detallada las principales características de los algoritmos evolutivos, prestando mayor atención a técnicas y operadores específicos de los Algoritmos Genéticos, que constituyen una de las ramas más importantes dentro de la computación evolutiva[60] [10].

3.1. Introducción

Existen diversos algoritmos evolutivos, pero todos centrados en los mismos conceptos. Se trabaja sobre un conjunto de posibles soluciones (llamadas individuos)

[1]El creacionismo es una doctrina religiosa donde se establece que el hombre, al igual que todo ser vivo, fue creado por algún ser divino y bajo sus propias reglas.

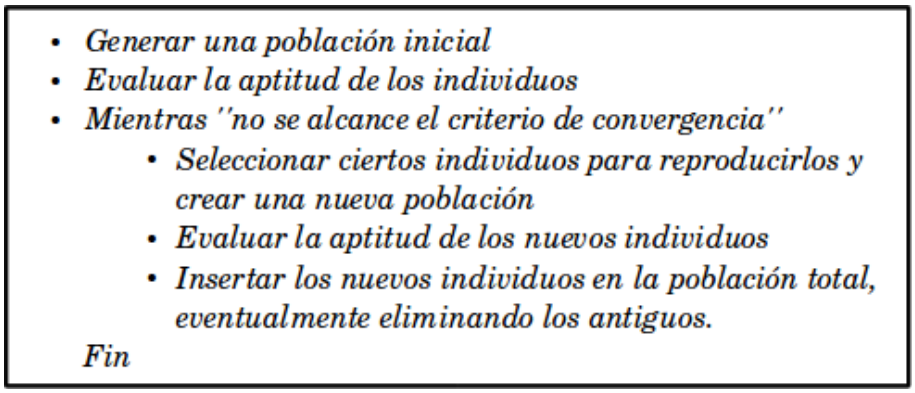

Figura 3.1: Pseudocódigo de un algoritmo evolutivo.

que se evalúan según una función de aptitud. Esta función indica una valoración para un individuo particular y es generalmente el "medio ambiente" donde se mueven los individuos. El objetivo del algoritmo es encontrar el valor óptimo de la función de aptitud. Para llevar esto a cabo se realizan cambios estocásticos en la soluciones siguiendo reglas fundadas en la evolución natural de las especies.

Si bien existen diversos algoritmos evolutivos, todos presentan un esquema similar al que se muestra en la figura 3.1. Como se aprecia, el algoritmo comienza generando una población inicial, generalmente aleatoria, y se evalúa su aptitud. Luego, el proceso entra en un bucle donde cada ciclo se conoce como una nueva "generación". A medida que transcurren las generaciones, la población va evolucionando. Para ello se seleccionan individuos de la población anterior y se reproducen. A estos nuevos individuos generalmente se le aplican operadores genéticos los cuales producen pequeños cambios similares a los que ocurren en la reproducción animal.

Ya que los algoritmos evolutivos no garantizan encontrar la solución óptima, la decisión de cuántas generaciones iterar está en manos del diseñador del algoritmo. Para alcanzar el criterio de convergencia se utilizan diferentes técnicas, entre las cuales se pueden encontrar: utilizar un número fijo de generaciones o utilizar una cantidad mínima de mejora (si los nuevos individuos no mejoran demasiado el algoritmo se detiene, ya que posiblemente se haya encontrado el óptimo de la función de aptitud).

Un problema frecuente en los algoritmos evolutivos, así como en otras metaheurísticas, es la convergencia prematura. Esto se debe en general al uso de mecanismos incorrectos para escapar de óptimos locales, lo que lleva a obtener malas soluciones.

La figura 3.2 muestra un ejemplo de cómo quedarían los diferentes individuos dispersos en la función de aptitud. En este caso, cada individuo tiene una sola variable, x. De este modo la función queda graficada en dos dimensiones. Esta función es el "medio ambiente" donde los individuos se adaptan.

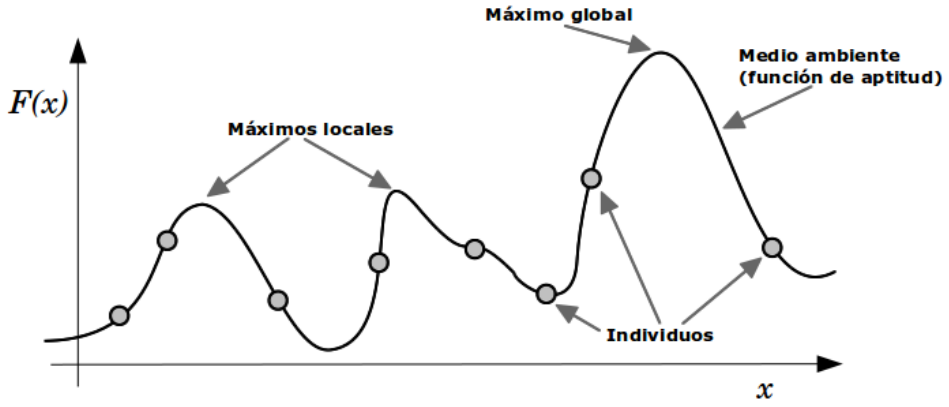

Figura 3.2: Individuos de una población dispersos por la superficie de la función de aptitud.

3.1.1. Fundamentos biológicos

Llegando al año 1900, y gracias a estudios de personas como C. Darwin, A Waismann y J. G. Mendel, se formularon teorías que cambiaron el modo de entender el origen de la vida y sobre todo del hombre [10], dando paso al pensamiento conocido como Neo-Darwinismo. Contraponiéndose a las teorías creacionistas, el Neo-Darwinismo explica que todo proceso de vida se basa en cuatro principios básicos: La reproducción, la selección, la competencia y la mutación. Según este enfoque, todas las especies partieron de un punto en común hace millones de años y gradualmente se han ido ramificando en nuevas especies, hasta llegar a ser tal como son hoy en día (figura 3.3).

Genotipo y fenotipo

Cada ser vivo posee cierta información genética codificada en una serie de cromosomas en las células de su cuerpo. Esta información indica todo el aspecto y el comportamiento de un ser particular. Dentro de un cromosoma la información genética está codificada por medio de sustancias químicas; la combinación de estas sustancias forman lo que se denomina el *genotipo* de un ser. La decodificación de esta información produce la morfología, el aspecto y el comportamiento de un ser vivo. Estos rasgos físicos característicos de un ser particular son llamados el *fenotipo*. La figura 3.4 muestra una analogía entre el genotipo y fenotipo de un cromosoma biológico y de uno artificial. El cromosoma artificial está compuesto por una cadena de unos y ceros (ver sección 3.1.3), que representan las variables de un determinado problema[2]. En el ejemplo, el fenotipo del

[2]Diferentes autores utilizan el término “cromosoma” para hacer referencia a todo el genotipo de un individuo. Otros hablan del “genotipo” del individuo, ya que este posee varios cromosomas.

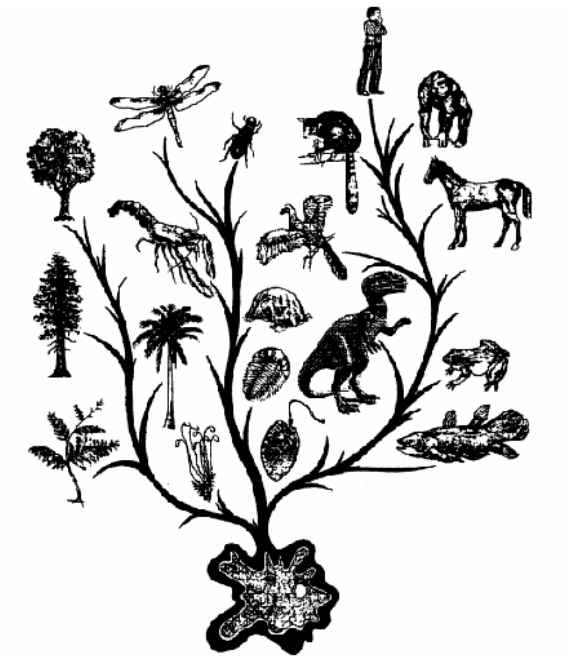

Figura 3.3: Evolución de las especies.

cromosoma artificial representa el valor de dos variables (x e y), con lo cual la función de aptitud queda graficada en tres dimensiones [10].

Reproducción y competencia

Por medio de la reproducción los seres crean descendientes transmitiendo su información genética. Para llevar a cabo la reproducción, hay un proceso natural de selección de individuos. En general, los individuos de una especie tienden a elegir como candidatos de reproducción a individuos aptos, entendiendo como tales a individuos bien adaptados a su medio. Para esto, en la naturaleza existe la atracción sexual, que permite que los individuos más desarrollados logren reproducirse y crear descendientes. Para generar este comportamiento, los algoritmos evolutivos utilizan técnicas de selección (ver sección 4.2.5) que permiten establecer una estrategia de elección de individuos para la reproducción. Luego, se utilizan diferentes estrategia para crear una nueva generación de individuos. Dependiendo del algoritmo utilizado, la nueva generación reemplaza a la anterior.

Ya sea la lucha de un depredador contra su presa, o la de dos individuos por la escasez alimento, o cualquier tipo de lucha librada por la naturaleza, los individuos de una especie o de diferentes deben competir de algún modo por la supervivencia. Esto es lo

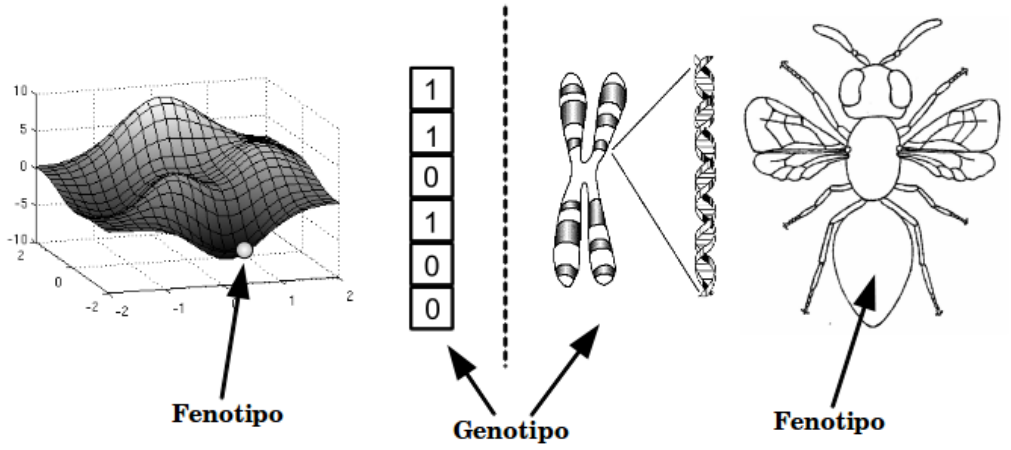

Figura 3.4: Analogía entre un cromosoma biológico y uno artificial.

que se conoce como *supervivencia del más apto* o *selección natural*. Los individuos mejor adaptados a su entorno sobrevivirán contra individuos mal adaptados. Esto es un proceso dinámico ya que el mundo entero está en constante movimiento, presentando cambios climáticos, cambios de vegetación, etc. Por su parte, las técnicas evolutivas utilizan una función de aptitud que asocia a cada individuo un puntaje a fin de medir su adaptación al entorno. De esta forma los individuos compiten, de diferentes formas para sobrevivir, en base a este puntaje [43]. En general, los algoritmos evolutivos se aplican sobre funciones de aptitud estáticas, es decir que a diferencia del mundo real las funciones de aptitud no están en constante cambio. No obstante existen aplicaciones de algoritmos evolutivos con funciones de aptitud dinámicas.

Al crearse un nuevo individuo existe lo que Darwin descubrió como uno de los pilares en la evolución: la mutación. La mutación genera pequeños cambios en los cromosomas de los nuevos individuos, lo cual produce una adaptación diferente a la de sus antecesores. Los algoritmos evolutivos simulan este comportamiento con diferentes técnicas de mutación (ver sección 3.3.3), donde en general se aplican pequeños cambios en las variables de la función objetivo, generando pequeños desplazamientos del individuo sobre la hipersuperficie del espacio de soluciones [10].

3.1.2. Clasificación

Desde que estas técnicas comenzaron a desarrollarse hace unos 50 años, existe una gran variedad de algoritmos planteados, incluyendo soluciones generales y soluciones a problemas específicos. Si bien hoy en día se desarrolla una gran diversidad de técnicas, se suele diferenciar en tres ramas principales de algoritmos evolutivos [60][10]:

Programación Evolutiva

Estas técnicas se enfocan en adaptar los individuos más que en trabajar sobre el genotipo de estos. De esta forma no se utilizan operadores de combinación sexual como el cruce. En general, estas técnicas pretenden modelar procesos evolutivos a nivel de especies y no a nivel de individuo, como las demás técnicas.

Probablemente fue una de las primeras técnicas aplicadas a problemas de predicción, además de ser la primera en utilizar codificación variable de los individuos. Este fue uno de los primeros intentos por simular la co-evolución.

Estrategias Evolutivas

Estas técnicas fueron desarrolladas en un comienzo para resolver problemas de ingeniería, que consistían en la optimización de la forma de algún componente que debía comportarse hidrodinámicamente. Estos problemas carecían de buenas soluciones con métodos tradicionales, lo cual llevó a considerar la aplicación de un algoritmo evolutivo.

Estos algoritmos se caracterizan por trabajar con vectores de números en punto flotante, a diferencia de otros métodos que utilizan en general codificación binaria o entera. Hoy en día existen diversas adaptaciones de estos algoritmos a soluciones de todo tipo.

Algoritmos Genéticos

Estos métodos son los más antiguos y desarrollados de los algoritmos evolutivos, originándose en la década de 1960 por J. Holland [32], teniendo en cuenta la adaptación biológica de los seres vivos en base a la reproducción. El eje central está en la utilización de un operador de cruce que toma dos individuos e intercambia su información genética para producir uno nuevo. Este método se basa en la idea de que distintas partes de una solución pueden separarse y combinarse para generar soluciones mejores. Adicionalmente se suele utilizar un operador de mutación para generar diversidad en la población.

3.1.3. Representación

Un punto muy importante en el diseño de un algoritmo evolutivo es la codificación de la información que tiene cada individuo, es decir, el genotipo. Esta representación debe ser adecuada y contener toda la información necesaria para poder decodificar este genotipo a un fenotipo adecuado que se corresponda con el problema particular. La correcta elección de la representación a utilizar puede ser decisiva para el buen desempeño del algoritmo. En general las representaciones genéticas se codifican como cadenas binarias, ya que estas pueden representar casi cualquier cosa, ya sean variables

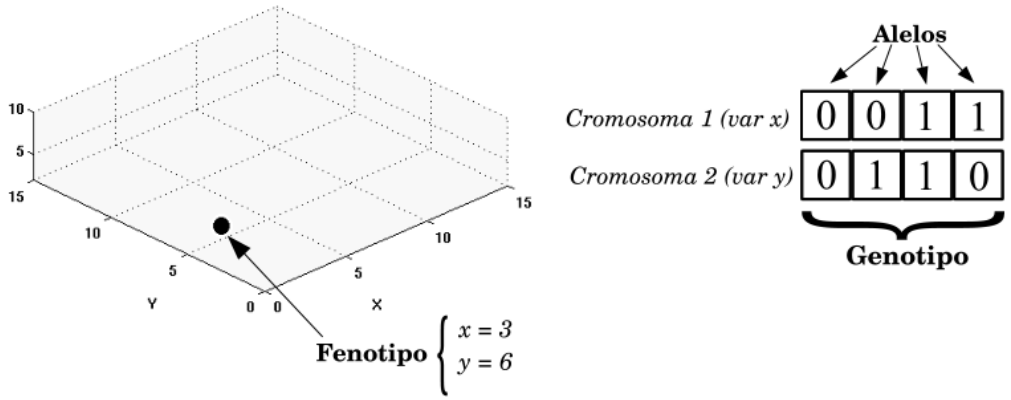

Figura 3.5: Un ejemplo de genotipo de dos cromosomas con representación binaria.

discretas, reales, conjuntos, o cualquier otro tipo de información que requiera el problema en cuestión, aunque existen otros tipos de representación que se detallan en breve.

El genotipo de un individuo puede tener uno o más cromosomas compuestos de uno o más genes cada uno, dependiendo del problema particular. En general, se intenta que cada gen represente una variable independiente del problema, y de esta forma la combinación genética resulta más sencilla de llevar a cabo.

En ocasiones, la mutación genera un salto en el genotipo de un individuo que resulta en un valor fuera el dominio válido para una cierta variable del problema. En este caso el algoritmo debe implementar una estrategia que solucione este problema. Puede optarse por eliminar la solución inviable o en algunos casos repararla [1].

Representación binaria

Una de las representaciones más utilizadas es la binaria. Esta consiste en una cadena binaria que representa los posibles valores de la función objetivo, ya sean números reales, enteros, etc. Cada cromosoma por lo general tiene la misma longitud y cada alelo, o posición, puede tomar los valores 0 o 1.

Una limitación de esta representación es el dominio acotado que puede utilizarse para las variables del problema. Por ejemplo, si se utilizan 5 alelos para un cromosoma, el fenotipo del individuo sólo puede tomar 32 valores diferentes ya que $2^5 = 32$. El ejemplo anterior permite deducir que la elección de la cantidad de bits o alelos a utilizar, condiciona sensiblemente la precisión que tendrá la solución obtenida a través del algoritmo genético.

La figura 3.5 muestra un ejemplo de utilización de la representación binaria. Aquí el genotipo del individuo está compuesto por dos cromosomas donde cada uno representa

una variable del problema (x,y). Cada cromosoma tiene 4 alelos (o bits). Por lo tanto, considerando una representación sin punto fraccionario, cada cromosoma puede tomar valores enteros entre 0 y 15 ($2^4 = 16$). El primer cromosoma tiene el valor 0011 con lo cual su fenotipo se traduce como 3. El segundo cromosoma tiene el valor 0110, siendo su fenotipo igual a 6. El tercer eje del gráfico representa los valores de la función de aptitud que son calculados luego de procesar el fenotipo del individuo.

Existen diversas razones por las cuales elegir una representación binaria frente a una en números reales (como podría considerarse más intuitivo). En general estas razones están fundadas en los estudios de Holland [32], quien mostró que una representación con mayor cantidad de genes, cada uno de los cuales sólo puede tomar un valor de un conjunto reducido de valores posibles (pocos alelos), genera más *esquemas*, lo cual produce mejoras en el desempeño del algoritmo a medida que transcurre el tiempo [1][32]. Las representaciones binarias necesitan mayor cantidad de alelos para el mismo dominio que las representaciones reales, ya que por ejemplo con 13 dígitos binarios pueden representarse 8192 valores distintos, cantidad que con una representación real solo necesita 4 dígitos.

Códigos de Gray

Uno de los problemas más grandes que tiene la representación binaria es que no mapea adecuadamente el espacio de representación con el espacio de búsqueda. Es decir, ante un pequeño cambio en un dígito binario, puede ocasionarse un gran salto en el número representado; y del mismo modo, dos números consecutivos pueden diferir por varios dígitos binarios. Tomando por ejemplo los números 5 y 6, los cuales son adyacentes (considerando números enteros), sus representaciones en binario son 101 100 respectivamente, las cuales difieren en dos dígitos. Este fenómeno ocasiona problemas ya que al aplicar un operador de mutación sobre un bit (o sea, cambiar un 0 por un 1 o viceversa) puede producirse un gran salto en el espacio de búsqueda, contraponiéndose a la idea de mutación.

Los códigos de gray aseguran que la adyacencia en el espacio de búsqueda se mantenga en el espacio de representación. De este modo un pequeño cambio en el genotipo genera fenotipos con saltos menores que una representación binaria clásica.

Codificación real

Al necesitar un dominio que abarque valores reales existen diversas opciones a utilizar. La más directa, pensando en la representación binaria o de gray es utilizar un punto fraccionario imaginario dentro del cromosoma. Esto presenta la limitación obvia de discretizar la precisión, siendo un problema si en verdad se necesita trabajar con todos los números reales. Por otro lado, al aumentar el número de variables y los dominios de

estas, la cantidad de dígitos necesarios crece considerablemente, convirtiendo al algoritmo en un proceso demasiado lento. Por ejemplo, un algoritmo donde se necesitan 50 variables (nada raro en un problema de neuroevolución), y donde cada variable tiene un dominio en el rango (-1024, 1024) con 5 dígitos binarios fraccionarios, la longitud del genotipo crece a 50 cromosomas de 25 dígitos cada uno, es decir 1250 componentes en el genotipo de un individuo.

Otra posibilidad es usar una representación estándar de codificación en punto flotante como puede ser una de IEEE donde existe una mantisa y un exponente. Sin embargo, este tipo de representación sigue presentando los problemas propios de las representaciones binarias antes explicados, además de ser mucho más costosa computacionalmente la traducción del genotipo al fenotipo.

Una tercera opción muy usada en la práctica es utilizar números reales directamente para la representación de cada variable. Siguiendo el ejemplo anterior donde se tenían 50 variables, ahora el genotipo de un individuo contendría 50 números reales con la representación propia de algún lenguaje de programación.

Mientras que desde un punto de vista teórico los alfabetos pequeños son más efectivos que los alfabetos grandes, se ha mostrado con innumerables aplicaciones concretas que la aplicación de variables reales para la representación de genotipos resulta muy efectiva [10].

3.2. Función de aptitud

Para poder llevar a cabo la selección y reproducción de los individuos de la población, es necesario que cada uno posea algún tipo de puntaje o valor de aptitud[51]. Para esto se utiliza una función de aptitud, también llamada función objetivo, o función de *fitness*. Este proceso es algo análogo a lo que ocurre a nivel biológico (aunque en menor grado). Los animales utilizan sus instintos para seleccionar pareja para la reproducción. Algunos organismos, como las plantas, dejan sus posibilidades de descendencia a la suerte, liberando polen al aire o esperando que otro organismo ayude a transportarlo. Las arañas, por ejemplo, tienen decenas de descendientes al mismo tiempo, de los cuales sólo unos pocos sobreviven. De este modo, sólo los individuos mejor adaptados logran llegar a la adultez y seguir produciendo descendientes, siguiendo con la idea de *selección natural* que formuló Darwin [13].

La función de aptitud define el entorno, el “medio ambiente”, en que los individuos de una población se mueven. Los valores de fitness que cada individuo posee constituyen una métrica de su desempeño y son usados para la selección de candidatos a reproducción, así como también para la selección de individuos que deben ser reemplazados al pasar de una generación a otra.

Las funciones de fitness dependen siempre del problema específico y es el diseñador

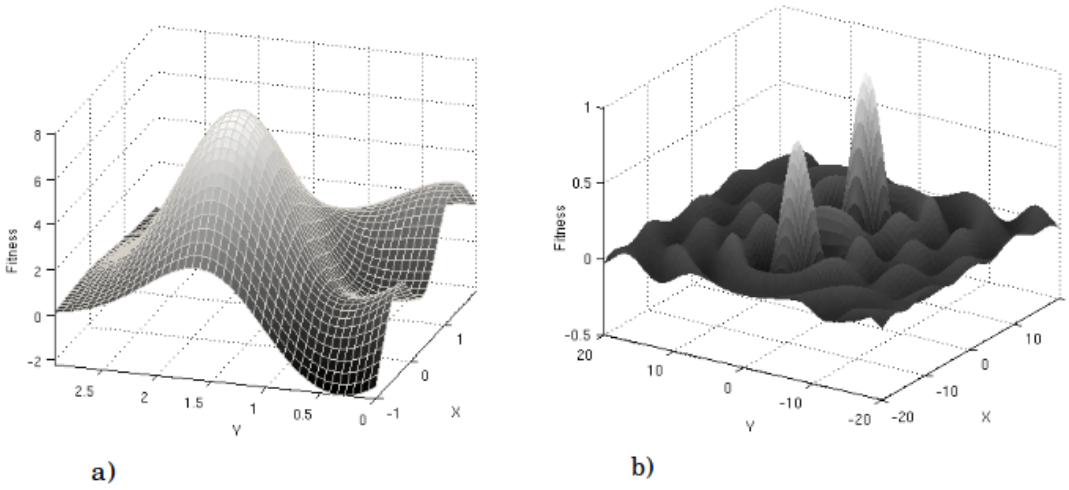

Figura 3.6: Dos ejemplos de funciones de aptitud.

del algoritmo el que debe desarrollarla. En muchas ocasiones estas funciones resultan triviales al problema a resolver, ya que se intenta aproximar una función simbólica específica. Sin embargo, en muchos casos se desea minimizar cierto aspecto pero no se conoce una función adecuada que represente el problema. Por ejemplo, se podría querer minimizar el rozamiento del aire en el diseño de un nuevo avión, sin que se disponga de una función que retorne la cantidad de aire que colisiona sobre el avión, por lo cual es necesario que el mismo entre a un simulador. Esto mismo sucede en general con los diseños de controladores robóticos, donde la función de aptitud está basada en la prueba del robot en un simulador específico.

Los algoritmos evolutivos tienen la particularidad de que se comportan muy bien con funciones de aptitud estocásticas. Es decir, funciones donde para una misma entrada pueden responder diferentes salidas, incluso nunca la misma. Esto no resulta nada raro en modelos del mundo real, donde los simuladores suelen tener funciones aleatorias para imitar el ruido del medio ambiente. Incluso se han desarrollado algoritmos evolutivos donde la función objetivo es una opinión subjetiva de alguna persona. De esta forma se han conseguido algoritmos para crear música u opinar sobre un cuadro [36].

La figura 3.6 muestra dos ejemplos relativamente sencillos de funciones de aptitud. Ambas son funciones con dos variables (x e y). La figura 3.6-a es una función bastante simple casi sin óptimos locales y con el óptimo global bien marcado. La figura 3.6-b es una función más complicada, la cual presenta diversos óptimos locales, necesitando un algoritmo evolutivo con estrategias más sofisticadas para poder escapar de estos.

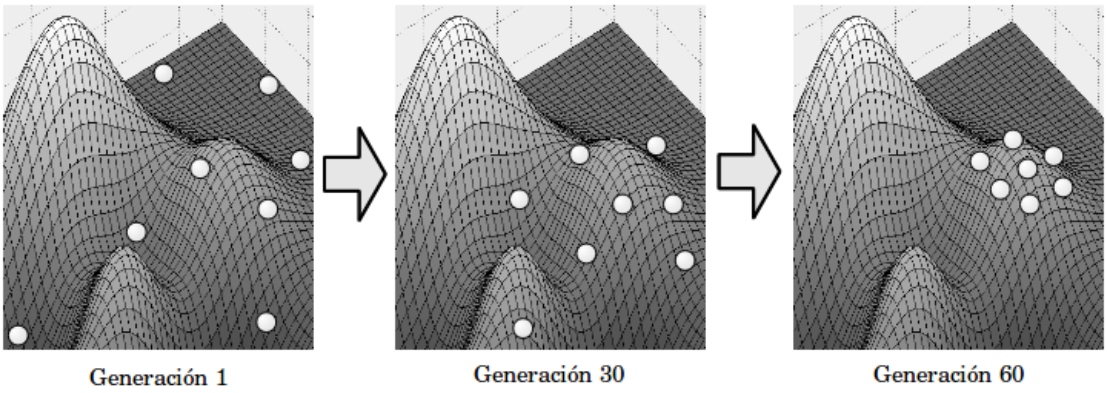

Figura 3.7: Posible estancamiento debido a una mala técnica de selección.

3.3. Reproducción y recombinación

El núcleo funcional de los algoritmos evolutivos radica en la reproducción para lograr diversidad genética. Los algoritmos evolutivos utilizan cuatro procedimientos esenciales para lograr la reproducción de forma análoga a la vida natural: la selección de padres, la mutación, la recombinación (o cruce) y el reemplazo (o defunción de los individuos)[10][51]. Dependiendo de cuál algoritmo o estrategia particular se trate, puede ocurrir que cierto operador no se use o se use más que otro. Por ejemplo, los algoritmos genéticos se centran en la recombinación genética como estrategia exploratoria y utilizan la mutación como un agregado extra. Muchas estrategias evolutivas no utilizan operadores de cruce, dejando todo el poder exploratorio a los operadores de mutación.

En el resto de la sección se describen algunas de las técnicas más utilizadas para estos procedimientos.

3.3.1. Técnicas de selección

Para llevar a cabo el proceso de selección de padres se utiliza alguna estrategia que evalúe el valor de aptitud de cada individuo. De esta forma, mejores individuos tienen mayores posibilidades de dejar descendientes, como ocurre en la vida real[26]. Sin embargo, en general se utilizan métodos probabilísticos, de forma tal que los individuos menos aptos aún tengan alguna posibilidad de dejar descendientes. Esta es una de las estrategias más importantes en los algoritmos evolutivos para escapar de máximos locales, ya que si sólo se enfocase en los mejores individuos el proceso podría quedar atrapado en un falso máximo.

La figura 3.7 muestra un ejemplo de este suceso. Debido a una mala disposición

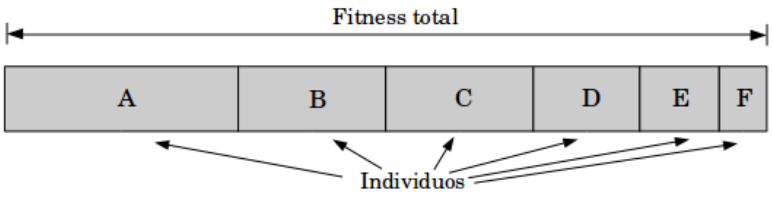

Figura 3.8: Selección proporcional.

inicial de los individuos[3], y suponiendo que solo se seleccionan los mejores individuos para la reproducción, el algoritmo queda atascado en un máximo local sin posibilidad de escapar. No es posible crear mayor diversidad genética ya que los operadores de cruce y mutación suelen producir sólo pequeños desplazamientos en la hiper-superficie de búsqueda. La mala disposición inicial de los individuos no es algo poco común, ya que un problema real podría tener decenas de variables y una función objetivo muy compleja; y la cantidad de individuos dispersos inicialmente no suele ser suficiente para abarcar grandes secciones. Estos hechos crean la necesidad de establecer estrategias para escapar de los atascamientos en máximos locales; una de estas estrategias es dejar que ciertos individuos malos sean padres de la nueva generación.

En general, las técnicas de selección existentes pueden ser clasificadas en tres grandes grupos[10]:

Selección proporcional

Este nombre describe un grupo de técnicas de selección que originalmente fueron propuestas por Holland[32], en las cuales se elige a los individuos de acuerdo a su valor de aptitud en proporción con los valores de toda la población. De esta forma, cada individuo tiene una probabilidad de ser elegido proporcional a su fitness.

La figura 3.8 muestra un ejemplo de cómo quedarían alineados los individuos en base a su función de aptitud. Luego se aplica algún método específico que selecciona los individuos como candidatos para la reproducción.

Existen cuatro grandes grupos dentro de las diferentes técnicas de selección proporcional:

- **Ruleta**. Posiblemente el método más usado en los algoritmos genéticos. El método consiste en ubicar a los individuos en forma de ruleta y hacerla girar para cada padre a elegir. De esta forma, los individuos más aptos tienen más posibilidad de

[3]Se considera “mala disposición” ya que ninguno está cercano al máximo global.

ser elegidos. El método es simple pero presenta el inconveniente de que el peor individuo puede ser elegido más de una vez, además de tener complejidad $O(n^2)$.

- **Sobrante estocástico**. Asigna determinísticamente las partes enteras de los valores esperados para cada individuo y usa un esquema proporcional para la parte fraccionaria. Resuelve los problemas de la ruleta pero puede causar convergencia prematura al tener mayor presión de selección.

- **Universal estocástico**. Intenta minimizar la mala distribución de los individuos en función de sus valores esperados. Si bien presenta una complejidad $O(n)$, puede tener convergencia prematura, y ocasiona que los individuos más aptos se reproduzcan muy rápidamente.

- **Muestreo determinístico**. Similar al sobrante estocástico, presentando los mismos inconvenientes. Adicionalmente requiere un algoritmo de ordenación.

Selección mediante torneo

Este método se basa en elegir al azar una cantidad establecida de individuos (normalmente 2) y comparar su aptitud. Luego, el ganador es elegido. Este proceso se repite hasta seleccionar la cantidad de individuos que se desea [10]. En los Algoritmos Genéticos se suele seleccionar una cantidad de padres igual al tamaño de la población.

Existen ciertas variaciones o alternativas aplicadas a este procedimiento:

- **Torneo con reemplazo.** En este caso, cada vez que un individuo compite contra otro, vuelve a ser introducido en la población. De esta forma, tiene grandes posibilidades de seguir siendo elegido como padre. Un posible inconveniente a tener en cuenta es que algunos individuos pueden participar en múltiples torneos, mientras que otros pueden no participar en ninguno [26][27].

- **Torneo sin reemplazo.** Aquí, luego de que un individuo compita en un torneo, no puede volver a la población. De este modo se asegura que cada individuo compita sólo en un torneo.

- **Torneo probabilístico.** Puede ser con o sin reemplazo. La única diferencia con el torneo tradicional es el agregado de una probabilidad p de ganar o no cada torneo. La idea es dar posibilidad a malos individuos de tener descendientes. Esta técnica intenta aumentar la diversidad genética, ya que el torneo tradicional hace mucha presión hacia los mejores individuos, sobre todo en su variante con reemplazo.

La figura 3.9 muestra un esquema sencillo de un torneo binario (se seleccionan grupos de dos individuos y gana el más apto).

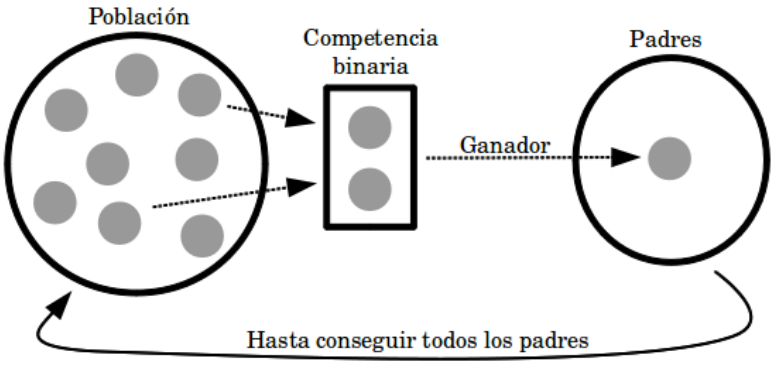

Figura 3.9: Selección por torneo binario.

Selección de estado uniforme

Es la técnica menos extendida de las tres. Se usa en algoritmos evolutivos donde solo algunos individuos son reemplazados en cada generación (los menos aptos). Suele utilizarse este método para sistemas basados en reglas, donde el aprendizaje es incremental [10].

Este método resulta muy útil cuando la población resuelve el problema colectivamente y no de manera individual.

3.3.2. Métodos de reemplazo

Una vez que los nuevos individuos son creados, se insertan en la población. Es necesario ir descartando soluciones anteriores, de lo contrario la población se haría innumerable en pocas generaciones. Existen muchas estrategias de reemplazo o "descarte" de las antiguas soluciones. La más común consiste en reemplazar la población completa por los nuevos descendientes, de esta forma cada individuo vive sólo una generación y dependiendo de su aptitud, su posibilidad de dejar descendientes es limitada. Esta es la estrategia más usual en los Algoritmos Genéticos [51].

Otra estrategia muy usada es la *actualización gradual*. En este caso los individuos se crean en menor cantidad y son insertados en la población inmediatamente. Por cada individuo que es insertado, otro debe salir. Para la elección del individuo a descartar existen diferentes estrategias. La más trivial es descartar al peor, pero esto genera una presión selectiva muy alta, con lo cual no es una buena opción. Otra forma es quitar el individuo más antiguo. Aunque lo más usual es hacer algún tipo de torneo.

Existen métodos más dinámicos aún, donde se permite que cada individuo viva una determinada cantidad de generaciones y no se lo quita de la población porque otro haya

nacido, sino porque llega a su tiempo de vida máximo (ver sección 3.4).

Una variación que se le puede hacer a muchas técnicas de selección y reemplazo es el *elitismo*. El *elitismo* consiste es mantener siempre vivos a cierto número de los mejores individuos. Este método se usa en técnicas de selección estocásticas, para no dejar que las mejores soluciones desaparezcan en caso de no ser elegidas por el método de selección. Si el número de individuos de la *elite* resulta muy grande, la presión selectiva se incrementa y el algoritmo podría converger prematuramente hacia un máximo local. Por lo tanto el número de individuos suele ser muy pequeño (1..5). Un método de selección determinista que puede ser visto como un caso extremo de elitismo es el truncamiento. Este consiste en seleccionar los mejores x individuos y descartar el resto [5].

3.3.3. Métodos de mutación

Uno de los ejes de varios algoritmos evolutivos es la mutación. Esta se basa en la mutación biológica, donde pequeños cambios en los genotipos pueden ocurrir al nacer un nuevo ser, dando como resultado un fenotipo con alguna característica diferente. La mutación en los algoritmos evolutivos fue pensada originalmente para representaciones binarias, donde consistía en cambiar un bit de cero a uno o viceversa. Hoy en día se aplica en representaciones reales e incluso en problemas de permutación donde la mutación debe tener alguna estrategia particular [63][10][51][7].

La mutación se aplica con una probabilidad muy pequeña en cada gen del genotipo de un individuo. Históricamente esta probabilidad estaba entre 0,001 y 0,01, aunque algunos autores recomiendan usar una probabilidad de $p = \frac{1}{L}$, donde L es la longitud del cromosoma [10]. En general, todas estas estrategias estuvieron pensadas para Algoritmos Genéticos, donde la mutación es un operador secundario (siendo el principal el cruce). Sin embargo en otras estrategias evolutivas la mutación es el operador de variación principal, con lo cual esta probabilidad puede aumentar ligeramente [7].

En representaciones binarias, o con códigos de Gray, la mutación simplemente consiste en, dada la probabilidad de mutar, cambiar el valor del bit de cero a uno o viceversa. En representaciones con más de dos símbolos, en general es necesario aplicar alguna estrategia para decidir qué símbolo será el que reemplace al actual. Esto podría decidirse de forma aleatoria o mediante algún orden de prioridad, dependiendo del problema.

En representaciones reales, la mutación es más complicada de aplicar, ya que la cantidad de símbolos válidos para un gen (alelo), es teóricamente infinita. Para esto, existen diversas técnicas [10]. Una posible es generar un nuevo número aleatorio con alguna distribución simétrica (como puede ser una distribución normal o una uniforme), y reemplazar el original por este. Esto puede resultar caótico en algunas ocasiones, ya que lejos de producir un cambio pequeño, el hecho de crear un nuevo número dentro

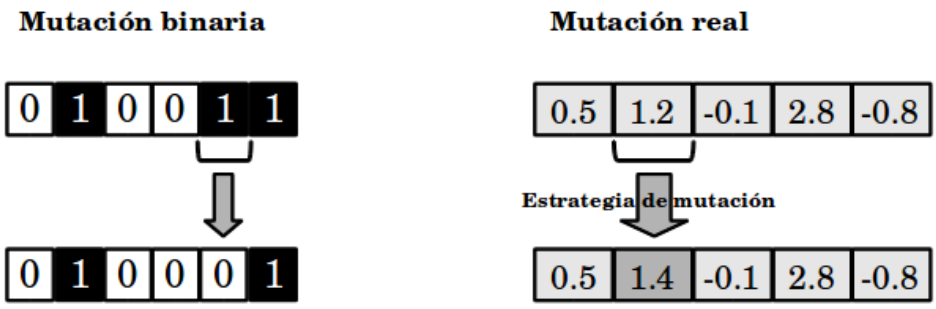

Figura 3.10: Ejemplos de mutación binaria y real.

de un rango permitido puede generar saltos grandes dentro del espacio de búsqueda. Para solucionar esto, otra técnica posible es generar un número aleatorio y sumarlo al valor original. De esta forma, es posible controlar la mutación y generar solo pequeños desplazamientos dentro del espacio de búsqueda. La figura 3.10 compara, con dos ejemplos, cómo cambia un cromosoma con representación binaria y con representación real.

Existen otros esquemas de representación donde la mutación debe presentar alguna estrategia particular. Por ejemplo, en un problema de permutaciones, no es posible generar un número al azar, ya que no siempre el cromosoma es válido. En este caso se utiliza alguna estrategia que cambie uno o más valores del cromosoma para generar un desplazamiento de la solución, como por ejemplo la técnica de inserción, o la técnica de intercambio recíproco[10].

Existen diversas estrategias particulares de aplicación de mutación dependiendo del problema particular. Una muy usada es comenzar la evolución con un parámetro de mutación alto, y decrementarlo, para terminar casi sin efecto de mutación. Esto tiene el fin de comenzar con una mayor exploración del espacio de búsqueda, y terminar con una especialización de las soluciones y una mejor convergencia.

3.3.4. Métodos de cruce

En la naturaleza, la cruza (recombinación genética, o *crossover*) es un proceso complejo, que se basa en la alineación de parejas de cromosomas donde estos se parten en ciertos fragmentos y se intercambian entre sí. Este proceso da como resultado un nuevo individuo que presenta rasgos fenotípicos de ambos padres [10].

En computación evolutiva, se simula este comportamiento intercambiando fragmentos de cromosomas artificiales de dos individuos. Si bien las técncias básicas suelen aplicarse a cromosomas binarios, existen variaciones que permiten aplicarse casi a cualquier

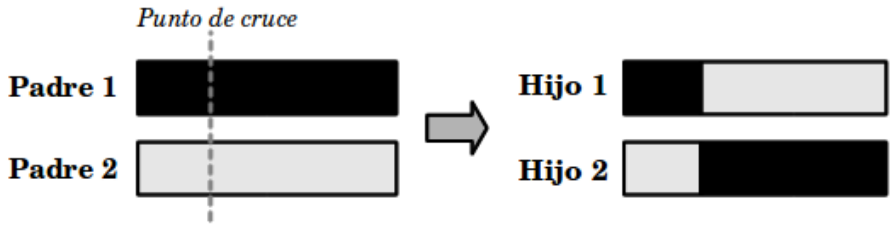

Figura 3.11: Cruce de un punto.

representación. La probabilidad de cruce en general ha sido considerablemtente más alta que la de mutación, sobre todo en Algoritmos Genéticos, llegando a valores de 0,8 (80 % de individuos con cruce). Sin embargo en otras estrategias evolutivas se han usado valores más pequeños.

A continuación se describen las técnicas más usuales [51][10], en las que siempre se consideran dos padres y dos hijos como resultado.

Cruce de un punto

Como su nombre lo indica, esta técnica se basa en considerar un punto fijo o aleatorio e intercambiar los fragmentos de cromosoma anterior y posterior a ese punto de ambos padres. De esta forma los nuevos individuos tienen información cruzada de ambos padres como lo muestra la figura 3.11. En la figura, un padre tiene un cromosoma negro, y el otro uno blanco, los hijos quedan con fragmentos de ambos padres.

Una limitación que presenta este método es que ciertos esquemas no pueden ser formados [51][10]. Por ejemplo, suponiendo que los padres tienen los siguientes esquemas:

$$Padre1 = 11 * * * * * 11$$

$$Padre2 = * * * * 1 * * * *$$

no hay forma de generar un hijo con el esquema

$$11 * *1 * *11$$

ya que se necesitarían dos puntos de cruce.

Cruce multipunto

Esta es una generalización del cruce de un punto. Aquí se toman n puntos dentro de los cromosomas de los padres para ser intercambiados y generar dos nuevos hijos.

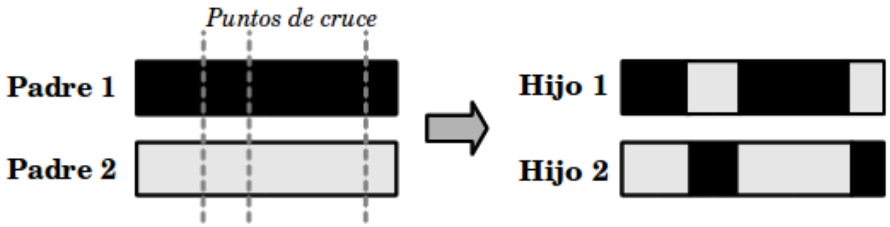

Figura 3.12: Cruce multipunto.

Un valor muy común en la práctica es $n = 2$, ya que se sabe que minimiza los efectos destructivos de la cruza, siendo posible hacer intercambio de sólo una variable específica. Para $n \geq 3$ no existen consensos con respecto a las ventajas o desventajas de usar dichos valores, incluso estudios empíricos no muestran grandes resultados[37].

La figura 3.12 muestra un ejemplo de cruce multipunto, en este caso con n=3. Según el problema a resolver, los lugares del cruce podrían definirse fijos, para intercambiar ciertas variables, o aleatorios.

Cruce uniforme

Esta técnica es aún más genérica que la cruza multipunto [51][10]. En este caso se utilizan todos los puntos posibles (todos los alelos) considerando una máscara de cruce generada aleatoriamente. Si el alelo tiene un 1, se copia el alelo del primer padre, si es un 0, se copia el alelo del segundo padre. Para el segundo hijo se hace lo contrario. Esta técnica es la más destructiva de las técnicas de cruza aquí planteadas.

Otros métodos

Existen diversos métodos planteados dependiendo del problema y la representación específica [63]. Se han propuesto, por ejemplo, técnicas que extraen características comunes de ambos padres. Otro ejemplo muy conocido es la cruza acentuada, que intenta autoadaptarse para conseguir patrones favorables. Para problemas de permutaciones, se han desarrollado diversos métodos de cruce, entre los que se puede nombrar la técnica *order crossover*, *partially mapped crossover*, etc. [10].

Para la representación real se han desarrollado diversas técnicas entre las que se pueden mencionar la *cruza intermedia*, la *cruza aritmética simple*, la técnica *Simulated Binary Crossover*, etc. [10].

3.4. Población

La población es el conjunto de todos los individuos que representan las soluciones con las que trabaja el algoritmo evolutivo. Existen dos grandes problemáticas con las cuales lidiar en cuanto a la población: cómo crear la población inicial, y cómo manejar el tamaño de la población.

Generación de la población inicial

En el primer caso, para crear la población inicial, existen diversas técnicas. En general, se intenta que la población sea lo más diversa posible, ya que, de esta forma se logra una gran distribución en el espacio de búsqueda. En el caso de utilizar una representación binaria, es usual generar de forma totalmente aleatoria cada genoma, generando un 1 o un 0 para cada alelo. De esta forma (excepto que el dominio del problema sea más acotado que todos los posibles cromosomas), se tiene una población bastante diversa al comenzar el programa. Esto es lo más usual en los Algoritmos Genéticos. Sin embargo, si la representación es real, no es tan fácil generar la población, ya que el lenguaje, lejos de tener dos símbolos, tiene una cantidad infinita. En este caso, es necesario plantear alguna estrategia para la generación de números, dependiendo del problema particular. Es usual utilizar alguna distribución simétrica. Si el dominio es acotado puede utilizarse una distribución uniforme dentro del rango válido. Si en cambio el dominio es muy amplio es preferible utilizar alguna distribución como la normal, donde los números se generan en torno a un valor.

Utilizar una buena estrategia para generar la población inicial puede ayudar al algoritmo a comenzar con un fitness elevado, aunque no es bueno perder mucha diversidad, ya que se podría estar orientando al algoritmo hacia una mala solución.

Tamaño de la población

En general, cuando se utiliza una población de tamaño fijo, la decisión de este tamaño no presenta grandes problemas. A mayor tamaño existen más posibilidades de encontrar buenas soluciones ya que hay más diversidad genética, pero con la contra de un tiempo de ejecución más lento. Con poblaciones muy pequeñas los tiempos son rápidos pero hay grandes posibilidades de quedar atrapado en un óptimo local.

Se han hecho diversos estudios [10] a cerca de los tamaños óptimos a utilizar, pero ninguno que sea efectivo. Algunos autores recomiendan utilizar un tamaño de 20 individuos para los Algoritmos Genéticos, aunque dependiendo el problema este número puede resultar muy pequeño.

3.4.1. Población de tamaño variable

Una variante de los algoritmos evolutivos muy poco vista es la utilización de poblaciones de tamaño variable. En este caso los individuos no son reemplazados por cada nueva generación, sino que cada individuo vive un cierto tiempo, hasta que muere y es quitado de la población, teniendo más posibilidades de generar descendientes.

Aquí se introducen algunos conceptos nuevos. Por un lado la *edad* de un individuo. Esto representa la cantidad de generaciones (iteraciones dentro del algoritmo) que este ha vivido. Por otro lado, cada individuo tiene un *tiempo de vida*. Esto representa la cantidad de generaciones que el individuo va a vivir. Esta cantidad se calcula mediante alguna técnica en proporción a su valor de aptitud. De este modo, individuos mejor adaptados viven más generaciones.

La asignación de tiempos de vida puede producir demasiada presión selectiva, aún mayor que la técnica de selección. Por esta razón es muy importante utilizar una buena estrategia de asignación. A continuación se listan algunas de las técnicas más usuales.

- **Asignación proporcional.** Similar a la técnica de selección por ruleta. Utiliza el promedio del fitness de toda la población y calcula los tiempos de vida de cada individuo en base a un mínimo y un máximo establecido. Presenta una gran debilidad ya que no todos los rangos de tiempos de vida son correctamente usados.

- **Asignación lineal.** Asigna los tiempos de vida en proporción al fitness máximo y mínimo que encuentra en la generación anterior a la que va a asignar. A medida que la población mejora, y los fitness se incrementan, se asignan tiempos de vida altos, lo que provoca un incremento importante del tamaño de la población.

- **Asignación bilineal.** Divide a la población en dos clases y asigna los tiempos de vida en relación a la distancia del fitness de cada individuo al fitness promedio.

- **Asignación por clases.** Esta estrategia agrupa a los individuos según su fitness en una cantidad fija de clases a través del algoritmo de clustering *k-medias* [35]. Cada clase recibe un rango de tiempos de vida válidos. Luego, se asignan los tiempos de vida a cada individuo de forma proporcional a su fitness dentro de cada clase [46]. La longitud del rango de tiempos de vida a asignar a cada clase puede ser fijo (una fracción igual para cada clase, donde los valores más altos pertenecen a clases con mayor fitness), o proporcional al número de individuos en cada clase (en este caso, clases con más individuos tienen rangos más amplios).

3.4.2. Especiación

Una técnica bastante reciente es la que divide a la población en especies [14][49][57]. Esta técnica imita la división en especies en el mundo biológico. Se trata

de grupos de individuos que buscan sobrevivir de manera independiente o con una interacción mínima.

En los algoritmos evolutivos se utiliza la especiación con el fin de generar mayor diversidad genética. De este modo, diferentes grupos de individuos exploran el espacio de búsqueda en distintos sectores. Para diferenciar los individuos entre sí y agruparlos en especies se utilizan diferentes métodos, como por ejemplo los de clustering. En general, se agrupa a los individuos según su genotipo; de forma que individuos de la misma especie están geográficamente más cercanos. Los algoritmos basados en especies suelen crear independencia evolutiva entre estas, sin embargo suelen permitir el intercambio de información de una especie a otra, en base a diferentes parámetros.

Capítulo 4

Robótica Evolutiva

Dentro del estudio de la robótica autónoma, desde hace varios años existe lo que se denomina Robótica Evolutiva (RE). En esta área se desarrollan controladores robóticos utilizando estrategias evolutivas aplicadas a redes neuronales [28]. La Robótica Evolutiva muestra un fuerte interés en estas estructuras por considerarlas un modelo artificial de la manera en que los humanos aprenden y por poseer la capacidad de representar el conocimiento adquirido a través de una estructura que, una vez entrenada, es capaz de operar en tiempo real.

Este capítulo contiene el aporte principal de esta tesina el cual consiste en la definición de una estrategia para obtener un controlador neuronal de arquitectura mínima capaz de comandar un robot autónomo. En estos casos, las metaheurísticas poblacionales resultan más adecuadas para lograr la adaptación de la red que un entrenamiento por gradiente. Esto se debe a que generalmente se trata de problemas complejos, donde es necesario adquirir alguna estrategia particular.

La estrategia propuesta ha sido aplicada para obtener un controlador robótico capaz de comandar un robot Khepera II en la resolución de cierto problema específico.

4.1. Trabajos relacionados

Para enfrentar el problema de la obtención de un controlador apropiado para resolver un determinado problema o tarea, existen diferentes soluciones desarrolladas en los últimos años. Las mismas abarcan muchas estrategias, que van desde algo simple como la adaptación de una única estructura mínima, hasta la división de la tarea en partes más pequeñas, que pueden ser entrenadas independientemente para luego combinarse en una única estructura. En [45] se desarrolló un mecanismo de integración automática de módulos que permitió combinar comportamientos básicos aprendidos previamente reduciendo de esta forma el costo de entrenamiento. También se han analizado distintas

estrategias elitistas de evolución como forma de mejorar el efecto de los operadores genéticos [47].

Desde otra perspectiva, se han desarrollado estrategias donde la solución al problema no se encuentra en un único individuo sino en la población en su conjunto [53], o en la combinación de subpoblaciones [44].

4.2. Estrategia propuesta

La estrategia propuesta se basa en un algoritmo evolutivo que utiliza una metaheurística poblacional de tamaño variable para llevar a cabo la adaptación del controlador. La evolución se aplica solo a los pesos de la red, con lo cual, la arquitectura se define a priori y queda fija durante todo el proceso evolutivo. Como herramienta para preservar la diversidad de los individuos se incorpora el concepto de especiación a través del cual se controla la selección de los individuos para la reproducción. La combinación de estas técnicas resulta un enfoque novedoso, ya que resuelve los dos principales problemas de las soluciones convencionales: la necesidad de definir a priori el tamaño de la población a utilizar y la pérdida de diversidad que provoca la presevación de las buenas soluciones.

A continuación se realiza una breve descripción de los problemas abordados. En el resto de la sección se detallan los aspectos de implementación de la estrategia desarrollada.

4.2.1. Descripción de la estrategia y problemas abordados

La estrategia desarrollada se utilizó para resolver dos tipos de problemas específicos muy estudiados en la robótica: la evasión de obstáculos y el alcance de objetivos. Para abordar esto, no se utilizaron procedimientos independientes, sino que se evolucionó el controlador con una función de fitness que premia a los individuos por llegar a un objetivo particular al mismo tiempo que dirige la búsqueda de la mejor solución orientando al individuo a no colisionar con los obstáculos que se le presentan.

La figura 4.1 muestra uno de los escenarios utilizados para la evaluación de los controladores obtenidos. En este caso, el robot parte de un punto específico y debe llegar a un objetivo ubicado en la esquina inferior izquierda del diagrama, representado con un ícono en forma de hogar[1], sorteando una serie de obstáculos que se le presentan en el camino.

[1]Normalmente se conoce este tipo de problemas con el nombre de “robot homing”.

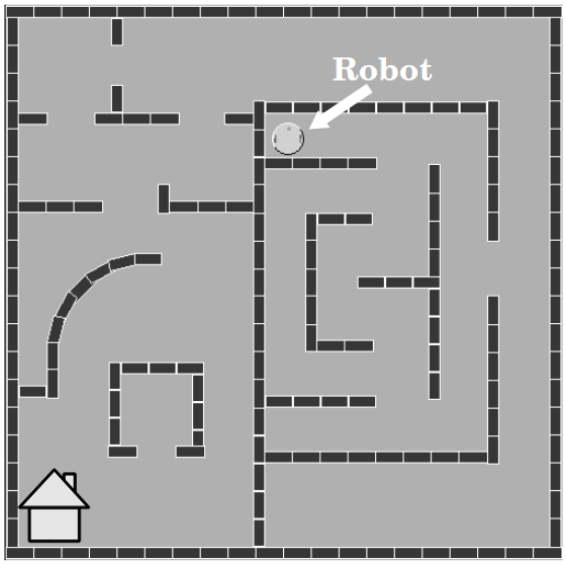

Figura 4.1: Laberinto usado para la adaptación del controlador robótico.

El problema del tiempo

Al desarrollar una estrategia evolutiva para la obtención de un controlador robótico es necesario diseñar una función de aptitud que califique a cada individuo según su desempeño. Para poder llevar a cabo esta calificación es necesario utilizar el controlador en un robot real y observar qué ocurre, para luego otorgar un puntaje a ese individuo. Este proceso puede llegar a ser infinitamente lento si se considera el movimiento de un robot. El encontrar un objetivo particular es una tarea que puede tomarle varios minutos a un robot. Sumando esto al hecho de que cada generación en el proceso evolutivo conlleva a la evaluación de varias decenas de individuos nuevos, se tendrá como resultado un proceso evolutivo que no converge o lo hace de una forma excesivamente lenta.

Para solucionar este problema, es muy normal en robótica el uso de simuladores específicos. Los simuladores imitan el comportamiento del robot real pero a una velocidad mucho más rápida, lo cual resulta muy ventajoso en este tipo de estrategias. Aún con este incremento en la velocidad de prueba de un controlador, el proceso de simulación sigue siendo mucho más lento que el resto del algoritmo evolutivo. Este problema se analiza con sumo detalle en los capítulos 5 y 6, donde se desarrolla la implementación del algoritmo en un entorno de programación paralela. En el apéndice B se describe brevemente el simulador utilizado para este trabajo, así como diversas modificaciones realizadas para ello.

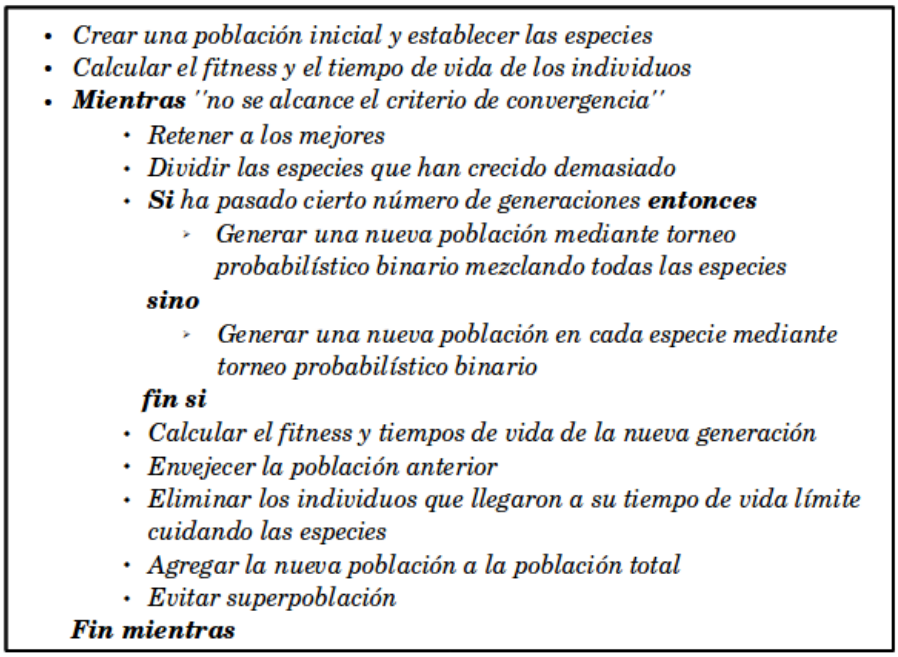

Figura 4.2: Pseudocódigo del algoritmo desarrolado.

El algoritmo

El algoritmo utilizado es una estrategia evolutiva híbrida, que si bien conserva muchas de la características de los Algoritmos Genéticos, no utiliza el operador de cruce como operador de variación principal, sino dos operadores de mutación, uno de los cuales ha sido definido específicamente para este problema.

A esto se suma la clasificación de la población en especies con el fin de explorar adecuadamente el espacio de soluciones.

La figura 4.2 describe en pseudocódigo el algoritmo planteado. Este parte de una población inicial generada aleatoriamente, la cual es dividida en ciertas especies (ver sección 4.2.3) con características comunes. En segundo lugar, se calcula el valor de aptitud de cada individuo y el tiempo de vida que permanecerán en la población. Luego, el algoritmo entra en un bucle que continua hasta que se alcanza el criterio de convergencia. En este caso se utilizó un número fijo de generaciones. La técnica utiliza cierto porcentaje de elitismo para conservar las mejores soluciones encontradas por el proceso de selección.

Dentro del proceso evolutivo, diferentes especies se desarrollan de forma paralela explorando el espacio de soluciones en sectores diferentes. Por cada iteración, cada especie se reproduce de forma independiente, generando una nueva población de hijos. Para llevar este proceso a cabo se utiliza un torneo probabilístico binario con una

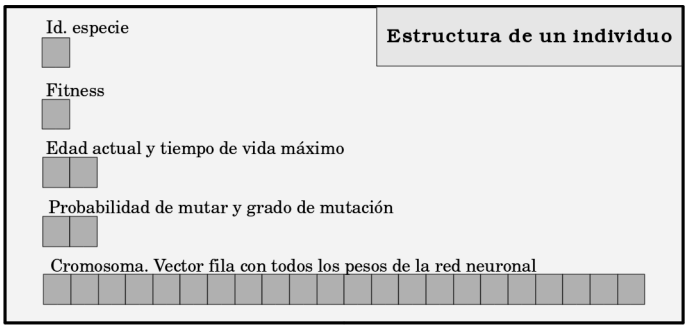

Figura 4.3: Estructura interna de un individuo de la población.

probabilidad de ganar de 0.8. Al pasar una cierta cantidad de generaciones, las diferentes especies se mezclan, dando origen a un torneo entre toda la población. Aquí las especies con mejores individuos (mejor fitness promedio) tienen más posibilidades de dejar descendientes que las demás, creciendo considerablemente el número de individuos en ellas. Si una especie crece demasiado (excede cierto número de individuos) es dividida en dos, dando lugar a una nueva especie. Luego de generar descendientes, se calculan sus valores de aptitud al igual que sus tiempos de vida.

Por cada generación, los individuos aumentan su edad y eventualmente mueren. Para esto se utiliza una estrategia de prevención de la desaparición de las diferentes especies. Un individuo que debe morir es quitado de la población sólo si su especie no está por debajo de un número mínimo de individuos permitido. De este modo, el proceso se asegura de retener las diferentes especies con el fin de seguir explorando diversas zonas del espacio de soluciones, aunque para un momento dado no sean del todo fructíferas. Esta estrategia intenta evitar caer en un óptimo local por conseguir cierta especie un buen desempeño en la función de aptitud.

Un problema frecuente al utilizar poblaciones de tamaño variable es la super-población. Si todos los individuos reciben altos valores de tiempos vida, o si se generan más individuos de los que mueren, puede ocurrir que la población total crezca paulatinamente hasta llegar a un número que estanque al proceso evolutivo, debido a su lentitud. Para evitar este suceso, se utiliza una estrategia que controla el tamaño de la población descripta en 4.2.4.

Estructura de los individuos

Cada individuo dentro de la población lleva asociada la siguiente información:

- El **cromosoma**, o genotipo del individuo, que contiene la representación de la red neuronal de arquitectura fija que determina el controlador robótico. Se trata de un

vector de números reales correspondientes a los pesos de los arcos. Su longitud es fija ya que la arquitectura nunca cambia.

- Un **identificador de especie** que permite saber de que rama de la población desciende. Es un valor numérico que se establece al crear la población inicial y es pasado a los descendientes de cada individuo, con excepción de cuando nace una nueva especie, donde este identificador es cambiado.

- El **valor de aptitud** o fitness del individuo que surge al evaluar su desempeño en la resolución del problema.

- La **edad actual** y el **tiempo de vida** máximo. Estos valores controlan la permanencia del individuo dentro de la población.

- La **probabilidad de mutar** y el **grado de mutación** son parámetros que determinan la capacidad de cambio del individuo indepndientemente del material genético de sus progenitores.

La figura 4.3 representa la información asociada a cada individuo. Es importante aclarar que el proceso evolutivo de variación genética sólo es aplicado al cromosoma. El resto de los datos es información extra para el algoritmo.

A continuación se explican con más detalle cada uno de los valores asociados a un individuo.

4.2.2. Arquitectura de la red

El controlador desarrollado está basado en una red neuronal de arquitectura mínima. Como ya se dijo anteriormente, el hecho de utilizar una arquitectura mínima posibilita la utilización de esta red como un módulo para un problema más complejo en un proceso de adaptación incremental. Así mismo, utilizar una arquitectura pequeña permite que el proceso sea muy veloz. Esta arquitectura se describe en la figura 4.4.

La red consta de ocho neuronas de entrada que equivalen a los ocho sensores de proximidad que posee el robot y que serán usados con el fin de evadir obstáculos (el apéndice A describe los componentes del robot Khepera II). La capa de entrada simplemente repite los valores de los sensores sin procesar esta información y envía las señales a la capa de salida, que está compuesta por dos neuronas. Estas neuronas tienen como función establecer, con sus salidas, la velocidad de cada uno de los dos motores del robot. La neurona nombrada *N8* es simbólica y representa el bias (o umbral) que se aplica a cada una de las dos neuronas de salida.

De este modo, el robot recibe ciertos valores en sus sensores de proximidad, los envía a la red neuronal, la cual responde con dos valores que son utilizados para establecer

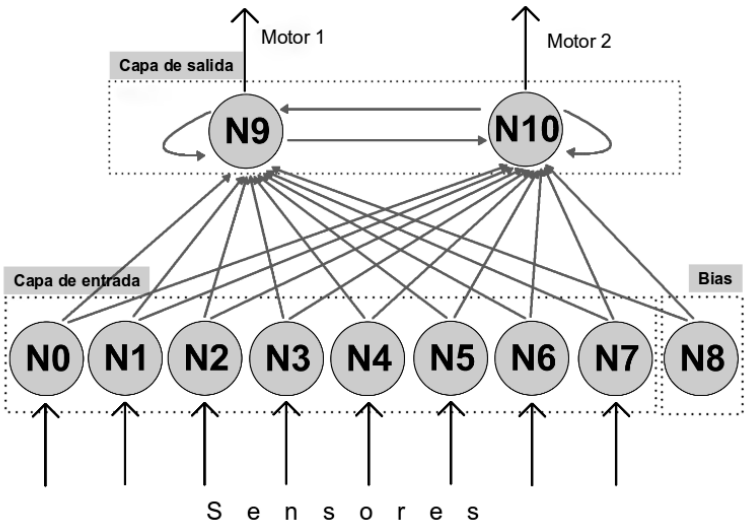

Figura 4.4: Arquitectura de la red neuronal del controlador.

las velocidades de los motores de las ruedas del robot. Este proceso se repite cada vez que el robot debe efectuar un movimiento individual; es decir, que la realización de un recorrido completo implica invocar repetidamente a la red neuronal pasándole como entrada la información actualizada de los sensores.

La arquitectura es un grafo conexo si se consideran como unidades funcionales sólo a las dos neuronas de salida. Es decir que existe un vértice de una neurona a otra y desde una a sí misma; además de los vértices de las neuronas de entrada a las de salida. Los arcos recursivos generan, como ya se explicó previamente, algún tipo de memoria para el controlador. Esta característica es necesaria para la resolución de problemas dinámicos. Es la que permite que el robot se mueva por el laberinto tomando acciones diferentes para la misma información de entrada. La memoria permite agregar información de la trayectoria que se está llevando a cabo. De no ser así, cada movimiento del robot seria independiente de los anteriores.

Genotipo

Ya que el proceso evolutivo sólo se aplica a los pesos de la red, el genotipo de un individuo sólo precisa retener los valores de estos pesos y ninguna información sobre la arquitectura en sí. De este modo, el único cromosoma de cada individuo es un vector fila con los 22 pesos que componen la red neuronal, codificados como números reales. De forma tal que el programa busca un óptimo en una función con 22 variables continuas.

4.2.3. Especiación

El algoritmo utiliza la especiación como estrategia para escapar de óptimos locales. Diferentes especies exploran el espacio de búsqueda en distintos sectores al mismo tiempo. Como ya se mencionó anteriormente, se intenta preservar a todas las especies con la idea de que sigan explorando aunque su fitness promedio no sea bueno.

Cada individuo tiene su identificador de especie, el cual es transmitido a sus hijos. Al comenzar el programa se separa a la población en 5 especies. Luego, cada especie se reproduce de forma independiente generando descendientes cercanos en el hiper-espacio. Cada cierto número de generaciones, las especies pueden mezclarse en el proceso de selección de candidatos a reproducir, provocando una mayor presión selectiva. Si una especie supera una cierta cantidad de individuos es dividida en dos nuevas especies: una con los individuos más aptos y otra con los menos aptos dentro de la especie.

Al envejecer y llegar a su edad límite, cada individuo debe ser retirado de la población. Este proceso es llevado a cabo cuidando que cada especie no decaiga por debajo de un mínimo establecido a priori. De este modo, cada especie sigue generando una pequeña cantidad de descendientes, incluso cuando su fitness promedio no es bueno.

Población inicial

Para generar la población inicial se utilizó una distribución normal con media 0 y desviación estándar 0.4. Cada alelo de cada individuo es generado aleatoriamente siguiendo esta distribución. De este modo, los pesos de la red neuronal podrán tener cualquier valor real, pero con mucha más posibilidad de que sea un valor cercano al cero. Si se utilizase una distribución uniforme habría que establecer cuidadosamente los límites y en cualquier caso, no se estaría teniendo buena variabilidad genética.

Luego de generar aleatoriamente 150 individuos iniciales, se los agrupa en especies utilizando el algoritmo de k-medias [2] y tomando como medida de similitud la distancia euclídea. Este algoritmo es del tipo ”winner-take-all”, es decir que como resultado se obtendrán k centroides que servirán para identificar a las distintas especies. El valor de k es un parámetro del algoritmo. La figura 4.5 muestra un ejemplo de algunos individuos agrupados en 3 especies. En este caso se consideran solo 3 variables de las 22 que tienen los individuos del problema, para fines gráficos. Al tener cromosomas similares, los individuos de la misma especie no sólo exploran en zonas cercanas del espacio de búsqueda, sino que además poseen comportamientos similares al evaluarlos en el simulador.

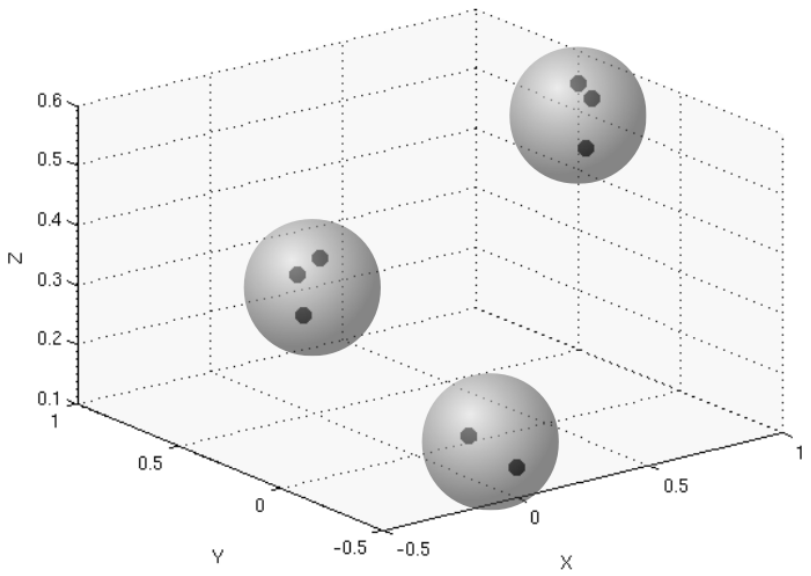

Figura 4.5: Ejemplo de agrupamiento por especies mediante k-medias.

4.2.4. Población de tamaño variable

En general, los algoritmos evolutivos utilizan una población de tamaño fijo, donde cada generación reemplaza a la anterior. En este trabajo se utiliza una población de tamaño variable, lo cual da más posibilidades a cada individuo de generar descendientes, ya que estos compiten en diversos torneos a lo largo de su vida.

Cada individuo posee una edad, que indica la cantidad de generaciones que vivió hasta el momento. Como se vio en el pseudocódigo del algoritmo, esta edad es incrementada en todos los individuos por cada generación. Además, cada individuo posee un tiempo de vida que indica la cantidad máxima de generaciones que vive en la población.

Asignación de tiempo de vida

El tiempo de vida de un individuo se calcula en base a su fitness siguiendo el algoritmo definido en [46]. Este es un método de asignación por clases, que obtiene una mejor distribución del rango de tiempo de vida a asignar en comparación con los métodos tradicionales, en el sentido de priorizar los individuos con mejor fitness.

La estrategia se basa en agrupar a los individuos en diferentes clases según su fitness. Para esto, se selecciona la cantidad de clases a utilizar y se utiliza un algoritmo de k-medias para que agrupe a los individuos, según la distancia euclídea entre sus valores de aptitud. Luego de este proceso, los diferentes individuos de la población quedan

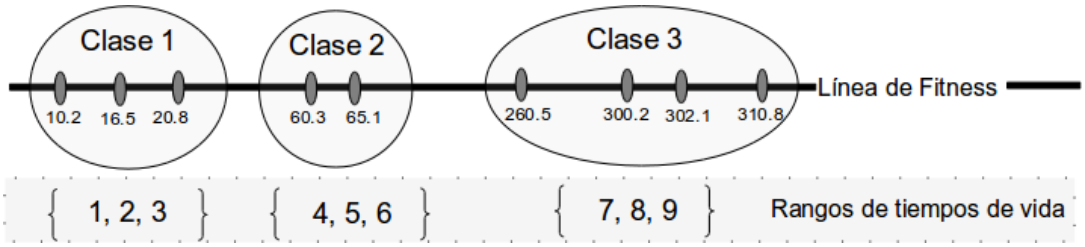

Figura 4.6: Método de asignación de tiempo de vida por clases.

etiquetados con un identificador de clase, que luego es usado para asignar los diferentes tiempos de vida. De este modo, las clases varían desde las que contienen a los individuos con peor fitness, hasta las clases que contienen a los mejores individuos. Luego de crear las clases, el rango de tiempos de vida a asignar es dividido en partes iguales por la cantidad de clases que haya. Luego, en cada clase se asigna el tiempo de vida a cada individuo proporcional a su fitness dentro de la clase.

La figura 4.6 muestra en un ejemplo cómo se divide a la población en clases y cómo es dividido el rango de tiempos de vida en cada clase. En este caso, el rango de tiempos de vida a asignar va de 1 a 9. Estos son los valores utilizados en este trabajo. La cantidad de clase se estableció en 3. Por lo tanto, a la clase 1 le corresponde el rango de 1 a 3, a las clase 2 de 4 a 6 y a la clase 3, de 7 a 9. De este modo, los mejores individuos, situados en la clase 3, recibirán tiempos de vida grandes (7 a 9), mientras que los individuos no adaptados recibirán tiempos de vida menores.

Por cada generación, el proceso de asignación de tiempos de vida por clases es realizado independientemente en cada especie de la población. De este modo, las especies se reproducen de modo independiente. Esto implica que los tiempos de vida altos o bajos son asginados a individuos buenos o malos dentro de una misma especie, respectivamente. De este modo, viendo a la población como un todo, dos individuos con fitness muy diferente pueden tener el mismo tiempo de vida. Esta estrategia intenta evitar los óptimos locales por demasiada presión selectiva. Eventualmente, cada cierto número de generaciones, cuando la selección de los individuos para la reproducción se lleva a cabo a través de un único torneo en el que participan todas las especies, los tiempos de vida a asignar son calculados considerando a la población como un todo y no dividida en especies.

Superpoblación

Para evitar que la población crezca desmedidamente se utilizó una estrategia de regulación. Una forma trivial que existe para evitar la superpoblación es la poda. Esto consiste en eliminar ciertos individuos cuando el tamaño de la población sobrepasa un

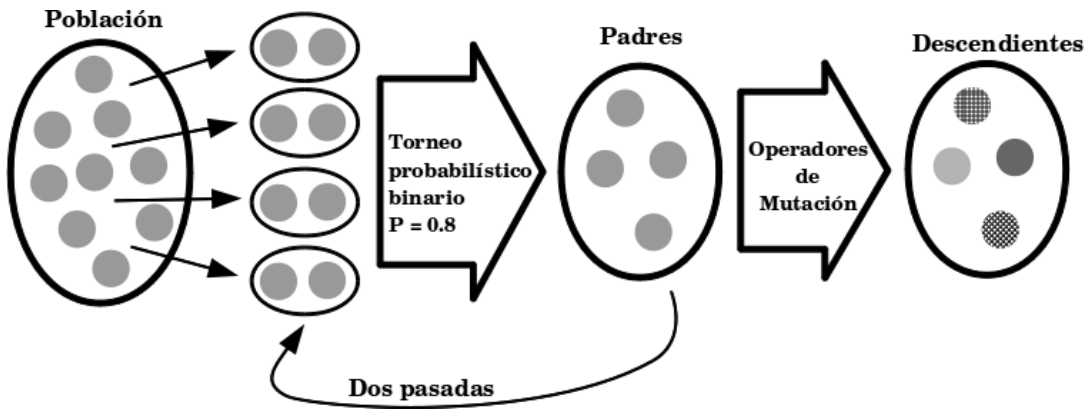

Figura 4.7: Método de selección utilizado. Torneo probabilístico binario en dos pasadas.

máximo prestablecido. Para esto se utiliza algún criterio de poda, que en general consiste en eliminar los individuos menos aptos. Esto genera una gran presión selectiva.

En este trabajo, se optó por utilizar un método de regulación más que de poda. Cuando la población total sobrepasa un máximo, establecido en 300 individuos, el rango de tiempos de vida a asignar cambia de 1-9 a 1-6. De este modo, durante algunas generaciones, los individuos tendrán un tiempo de vida promedio inferior, lo cual genera un decremento paulatino del tamaño de la población. Esta técnica evita tener que eliminar individuos de forma selectiva, o al azar, que podrían no haber tenido aún descendientes.

4.2.5. Selección y reproducción

Como en todos los algoritmos evolutivos, en cada generación es necesario seleccionar ciertos individuos como candidatos de padres de la nueva generación. La técnica utilizada en este trabajo es un torneo probabilístico binario en dos pasadas. El proceso consiste en mezclar todos los individuos y realizar un torneo de toda la población (o especie) de a pares, con una probabilidad de ganar de 0,8. Es decir que aproximadamente el 20 % de las veces se seleccionará un individuo no tan bueno. El resultado de la primera pasada del torneo es un conjunto de individuos cuya cardinalidad es equivalente a la mitad del tamaño de la población original. Con estos individuos se realiza otra pasada idéntica, dando como resultado un subconjunto del conjunto anterior formado por un cuarto de individuos de la población original. Luego de esta selección de individuos, el proceso de reproducción es directo. Se utiliza un algoritmo sin cruce, con lo cual, de cada padre sale exactamente un hijo. Luego de convertirse en descendiente, cada nuevo individuo es mutado según se explica en la sección 4.2.6.

Por lo tanto, por cada generación nace una nueva población de hijos cuyo tamaño

es la cuarta parte del tamaño de la población anterior. Este número se equilibra con el promedio de vida de cada individuo que es cercano a 4. De este modo la población está medianamente equilibrada en tamaño a lo largo de las generaciones. La figura 4.7 ilustra el comportamiento del método.

4.2.6. Operadores genéticos

Como en muchos algoritmos evolutivos, este método se basa en la mutación como eje de variación genética. El algoritmo no utiliza cruce, ya que el proceso de reproducción es de un padre a un hijo. Por otro lado, los métodos de cruce tradicionales estudiados en capítulos anteriores, no otorgan buenos resultados cuando se trata de diversificar redes neuronales. Dividir el cromosoma de una red neuronal y mezclarlo con otro no genera otra red con comportamientos similares a ambos. Existen muy pocos métodos de cruce diseñados para estos casos.

En este trabajo se utilizaron dos operadores genéticos. El primero es una mutación estándar, que genera pequeñas variaciones aleatorias en el genotipo del individuo. El segundo es operador de mutación más parecido a un operador de cruce, pero partiendo de un sólo individuo, al que se nombró *mutación de comportamiento simétrico*. Este es un operador estructural, no aleatorio, que se basa en cambiar el comportamiento del individuo y fue exclusivamente diseñado para este trabajo.

Mutación aleatoria

El principal operador de mutación es un operador estándar para representación de números reales. Como se vió en un capítulo anterior, una forma sencilla de generar variación en un cromosoma con representación real es adicionando un cierto valor a cada alelo[2]. De este modo, para cada alelo del genotipo, se verifica si hay que mutar y en caso afirmativo se genera primero algún número con una distribución de probabilidad y se lo adiciona al valor original del alelo. Podría utilizarse un incremento fijo, pero esto genera menor diversidad y un modelo de variación genética más estático, ya que para llegar de un cromosoma a otro particular habría que pasar obligatoriamente por una serie de cromosomas intermedios. En este trabajo se genera un valor aleatorio para ser incrementado y dependiendo la distribución elegida, se generan pequeñas variaciones en el cromosoma del individuo mutado.

En este punto, es de vital importancia la elección de la distribución a utilizar para generar el incremento del alelo. Una primera opción es utilizar una distribución uniforme. En tal caso, es difícil definir los límites de la distribución, ya que el dominio de la

[2]La estrategia podría elegir entre adicionar o sustraer aleatoriamente, o generar un valor aleatorio que incluya los números negativos.

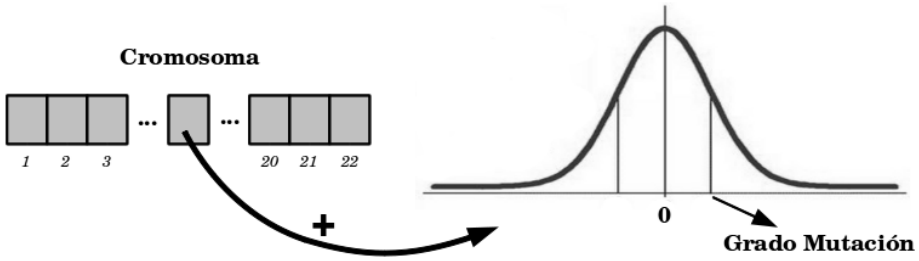

Figura 4.8: Funcionamiento del operador de mutación estándar.

codificación no los tiene. Si se elige un rango muy pequeño se podría estar en un estado muy similar al de incrementar un valor fijo. Esto dependerá de cuánto afecte un cambio a la red neuronal, de lo cual existen muy pocos estudios en la actualidad. Una segunda opción, utilizada en este trabajo, es optar por una distribución normal (o *gaussiana*) centrada en el cero. De este modo se generan números en general pequeños pero muy esporádicamente puede producirse un gran salto en el genotipo de un individuo.

Cada individuo posee dos datos muy importantes para llevar a cabo la mutación en sus descendientes: una probabilidad de mutar y un grado de mutación. La probabilidad es usada para decidir si se muta o no cada alelo del cromosoma del nuevo hijo. El grado de mutación es la desviación estándar de la distribución normal utilizada para generar los incrementos de la mutación. La figura 4.8 ilustra el modo de funcionamiento de este operador.

Al nacer, a cada individuo se le asigna una probabilidad de mutar de 0.25 y un grado de mutación de 0.20. Estos valores son utilizados para mutar los descendientes que el individuo tenga. Es decir que se mutan aproximadamente 5 o 6 de los 22 alelos de cada nuevo hijo, con incrementos aleatorios dentro de la desviación de 0.2. Estos valores generan cambios pequeños en los cromosomas si se necesita explorar grandes distancias dentro del espacio de búsqueda. Sin embargo son saltos muy grandes cuando se intenta adaptar una red neuronal, donde en general se utilizan incrementos muy pequeños.

Para solucionar esto, la probabilidad de mutar junto con el grado de mutación de un individuo disminuyen a lo largo de su vida. Cada individuo comienza con los parámetros detallados en el párrafo anterior y muere con una probabilidad de mutar de 0.05 y un grado de mutación de 0.08. Estos valores son linealmente reducidos a lo largo de la vida del individuo considerando los parámetros iniciales y finales, junto con su tiempo de vida. Por cada generación, todos los individuos envejecen y sus parámetros de mutación son actualizados según las ecuaciones (4.1) y (4.2).

$$ProbabilidadDeMutar = Mut_{ini} - (Mut_{ini} - Mut_{fin}) * \frac{edad}{tiempoDeVida} \quad (4.1)$$

$$GradoMutacion = Grado_{ini} - (Grado_{ini} - Grado_{fin}) * \frac{edad}{tiempoDeVida} \quad (4.2)$$

donde Mut_{ini} y Mut_{fin} son los valores de probabilidad de mutar inicial y final respectivamente; $Grado_{ini}$ y $Grado_{fin}$ son los valores de la desviación inicial y final respectivamente.

Esta estrategia permite que se explore de forma amplia el espacio de búsqueda al mismo tiempo que se especializan ciertos individuos cercanos al óptimo.

Mutación de comportamiento simétrico

El segundo operador genético utilizado es un operador más parecido a una cruza que a una mutación. Sin embargo este se aplica a un solo padre de modo que, formalmente hablando, se corresponde mejor con una mutación. Se trata de un cambio no aleatorio que modifica las posiciones de los genes que componen el cromosoma. Este operador fue especialmente diseñado para este trabajo con el fin de realizar un esfuerzo más por escapar de óptimos locales. El operador provoca un cambio en la distribución de los pesos de la red neuronal con el fin de generar un comportamiento diferente pero definido.

La idea de este operador radica en la observación del funcionamiento del controlador obtenido en el robot. En ciertas ocasiones, el robot aprende una estrategia de movimiento buena que no lleva necesariamente al lugar buscado. En este sentido se ideó un operador que hiciera que el robot funcionase “en espejo” con respecto a su movimiento. Es decir que cuando el robot dobla a la izquierda según los datos en sus sensores, luego del operador dobla a la derecha, como lo muestra la figura 4.9.

Esto provoca algo similar a la habilidad diestra o zurda de los seres humanos. Si se aplica este operador a un padre seleccionado, su hijo hará exactamente lo mismo que su padre, pero hacia el otro lado. Para lograr entender cómo deben cambiarse los alelos del cromosoma debe pensarse en términos de la red neuronal que este representa. La idea es que la neurona que comanda al motor 1, ahora comande al motor 2, de modo tal que en vez de doblar hacia un lado, lo haga hacia el otro. Sin embargo, no basta con intercambiar ambas neuronas. Si se hiciese esto, el robot hijo doblaría a la derecha en el caso que su padre doble a la izquierda, sin considerar los obstáculos de ambos lados. El objetivo, en cambio, es que el hijo procese la información de un lateral, del mismo modo que el padre procesa la del otro. La figura 4.10 muestra un ejemplo de cómo debe entenderse el cambio

Figura 4.9: Movimiento del robot antes y después de la mutación de comportamiento simétrico.

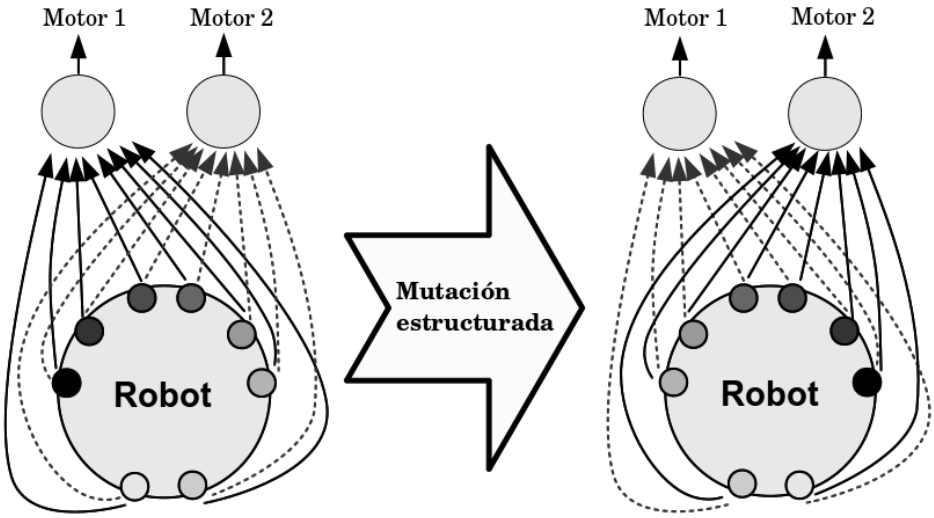

Figura 4.10: Esquema de los pesos de la red neuronal antes y después de la mutación de comportamiento simétrico.

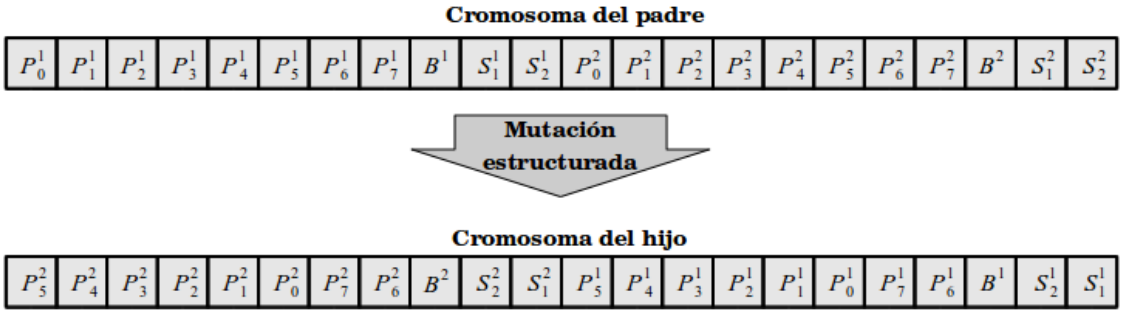

Figura 4.11: Esquema de un cromosoma antes y después de la mutación de comportamiento simétrico.

de pesos en la red. Cada peso que va desde un sensor[3] x hasta cierta neurona de salida, en primer lugar debe ubicarse en el lugar del peso que va desde el sensor x', ubicado en espejo con respecto a x. Luego de realizar estos cambios, ambas neuronas de salida son intercambiadas entre sí. Vale mencionar que los pesos entre las neuronas de salida también son intercambiados.

La figura 4.11 es un esquema preciso de cómo debe cambiarse el cromosoma para generar esta mutación. En el esquema, los superíndices indican hacia qué neurona de salida se dirige cada peso; los pesos nombrados P son los que van desde una neurona de entrada hasta una de salida, numerados del 0 al 7; los pesos nombrados S son los pesos que van de una neurona de salida a otra de salida, donde el subíndice indica desde qué neurona sale la señal; y por último, los pesos nombrados B indican el valor del bias para una cierta neurona de salida.

4.2.7. Función de fitness

Para evaluar el fitness de cada individuo se utiliza un proceso de simulación que permite analizar el funcionamiento del controlador obtenido en el robot Khepera II. El proceso de simulación consta de una serie de pasos en los que se evalúa el comportamiento del robot, obteniendo un puntaje directamente proporcional a su velocidad de desplazamiento e invesamente proporcional a su distancia al objetivo. Para esto, el robot no sólo utiliza los valores de sus sensores y motores, sino que es dotado de una especie de sistema de posicionamiento que le indica el punto preciso en el que está ubicado.

La función de fitness presenta dos grandes etapas. La principal, es la encargada

[3]En realidad, los pesos van desde una neurona a otra. En el dibujo se quitaron las neuronas de entrada por simplicidad.

de orientar a los individuos a moverse más rápido intentando encontrar el objetivo. La segunda etapa mejora el desempeño de los individuos que logran llegar a este objetivo.

La evaluación itera una cierta cantidad de veces. En cada iteración se observan los valores de los sensores, se procesa la información por el controlador y se efectúa el movimiento apropiado. El fitness es aumentado en cada iteración en base a la ecuación (4.3).

$$fitnessActual = fitnessActual + (fitObjetivo * fitVelocidad * fitGiro) \tag{4.3}$$

siendo:

$$fitObjetivo = (1/8)^{(distanciaAlObjetivo/800)} \tag{4.4}$$

$$fitVelocidad = \frac{(mot_1 + mot_2)}{2} \tag{4.5}$$

$$fitGiro = \frac{2 - abs(mot_1 - mot_2)}{2} \tag{4.6}$$

donde:

mot_1 y mot_2 son las salidas de los motores 1 y 2 respectivamente.

$distanciaAlObjetivo$ es la distancia euclídea en milímetros que hay entre el robot y el objetivo.

A continuación se analizan las ecuaciones 4.4, 4.5 y 4.6.

La primera es la que establece un valor en base al objetivo buscado. En este caso se intenta dar un valor de aptitud inversamente proporcional a la distancia entre el objetivo y el individuo. Es decir que cuando más cerca esté el individuo del objetivo, su fitness será mayor. En este caso se utiliza una función no lineal obtenida de forma experimental. La expresión $(1/8)^{(distanciaAlObjetivo/800)}$ es una función exponencial.

La distancias utilizadas para evaluar el fitness son distancias en milímetros. El escenario usado es un cuadrado de $1000x1000mm$, de forma tal que el robot nunca puede estar a más de $1500mm$ de distancia del objetivo. Estos valores son analizados en la figura 4.12, donde se ve graficada la función exponencial. La idea es que al comenzar el movimiento del robot no resulte significativo el acercamiento al objetivo pero en caso de estar cerca, se podría decir a partir de $500mm$, los valores de aptitud comienzan a ser rápidamente incrementados. Como se ve en la figura, esta función toma valores entre 0 y 1.

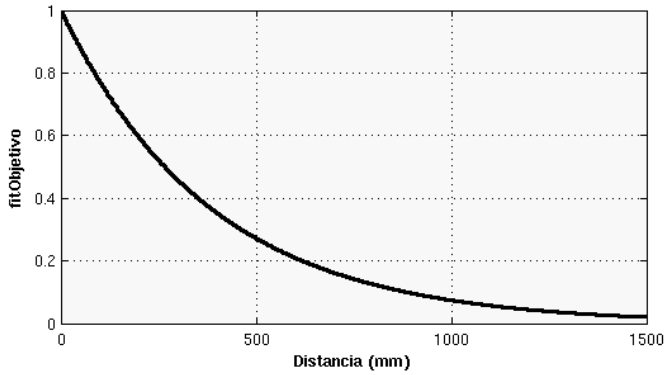

Figura 4.12: Función utilizada para calcular el fitness relativo al objetivo.

En el caso de la ecuación 4.5, esta otorga valores de aptitud más altos cuanto más rápido se desplace el robot. Los motores pueden tomar valores enteros entre -1 y 1, de forma tal que esta ecuación puede tomar valores entre -1 (ambos motores en -1) y 1 (ambos motores en 1)[4]. Esta expresión puede hacer que el fitness tome valores muy negativos si el individuo evaluado utiliza sus motores en reversa mucho tiempo. La idea de la expresión es orientar a los individuos a buscar el objetivo desplazándose hacia adelante y avanzando lo más rápido posible.

La ecuación 4.6 intenta evitar el efecto de giro de los individuos, reforzando de este modo la ecuación 4.5. Esta expresión puede tomar valores entre 0 (un motor en -1 y el otro en 1) y 1 (ambos motores con la misma velocidad).

Cabe destacar que mientras la ecuación 4.5 toma valores entre -1 y 1, la ecuación 4.6 los toma entre 0 y 1. De este modo, la expresión $fitVelocidad$ tiene mayor importancia en el cálculo del fitness total que las expresiones $fitGiro$ y $fitObjetivo$.

El bucle de evaluación puede terminar por tres razones:

- Por alcanzar un máximo de pasos de evaluación establecido en 400.
- Por colisionar el robot contra alguna pared.
- Por llegar al objetivo.

Dada cualquiera de estas condiciones, el proceso de evaluación termina y se retorna el fitness acumulado hasta el momento.

Si el robot encuentra el objetivo se intenta premiar fuertemente al individuo, para diferenciarlo de otros que no lo hayan hecho. En tal caso, el fitness es aumentado según la ecuación 4.7.

[4]En realidad los motores toman valores enteros entre -10 y 10. En este caso son escalados antes de utilizarse para fines prácticos.

$$fitnessActual = abs(fitnessActual) * \frac{maxPasosAEvaluar}{pasosDados} * \alpha \tag{4.7}$$

donde:

maxPasosAEvaluar es la constante que determina cuántos pasos se evalúan como máximo para un mismo individuo, en este caso 400.

pasosDados es la cantidad de pasos que el robot dio al momento de encontrar el objetivo.

α es una constante calculada experimentalmente que incrementa el fitness hasta donde sea conveniente. En este caso $\alpha = 5$.

Aquí es donde comienza la segunda etapa de la función de fitness, ya que el valor aumenta considerablemente y no es comparable con el fitness de individuos que no hayan encontrado el objetivo. La expresión $abs(fitnessActual)$ convierte el fitness a positivo en caso que este sea negativo. Esto se realiza ya que no importa con qué fitness consigue el individuo llegar al objetivo, este debe aumentar. La expresión $\frac{maxPasosAEvaluar}{pasosDados}$ establece un fitness inversamente proporcional a la cantidad de pasos dados hasta encontrar el objetivo, o dicho de otro modo, otorga mayor fitness a individuos que hayan llegado al objetivo en menor tiempo.

Al observar cuidadosamente la función de fitness se pueden inducir ciertos óptimos locales que es necesario evadir en el proceso evolutivo. Por ejemplo, el robot podría conseguir un valor de aptitud muy alto por recorrer grandes distancias de forma muy veloz, minimizando los giros. Para encontrar el camino al objetivo, es probable que los individuos deban sortear este óptimo local, ya que es necesario realizar diversos giros para encontrar algún camino válido.

Concluyendo el análisis de la función de aptitud (y del resto de la estrategia) cabe destacar que esta evaluación es la más costosa computacionalmente hablando. Incluso en un simulador ejecutado en una computadora con buena capacidad de cálculo, la evaluación implica muchas más operaciones en comparación al resto del proceso evolutivo. Este aspecto es desarrollado en los capítulos 5 y 6 donde se propone una implementación más veloz del algoritmo.

4.2.8. Resultados

La estrategia propuesta fue evaluada numerosas veces con el fin de obtener un controlador apropiado que resuelva los problemas planteados. La figura 4.13 muestra dos gráficos con el promedio de 50 ejecuciones independientes cada uno. En cada gráfico se observa el fitness del mejor individuo y el fitness promedio de toda la población a lo

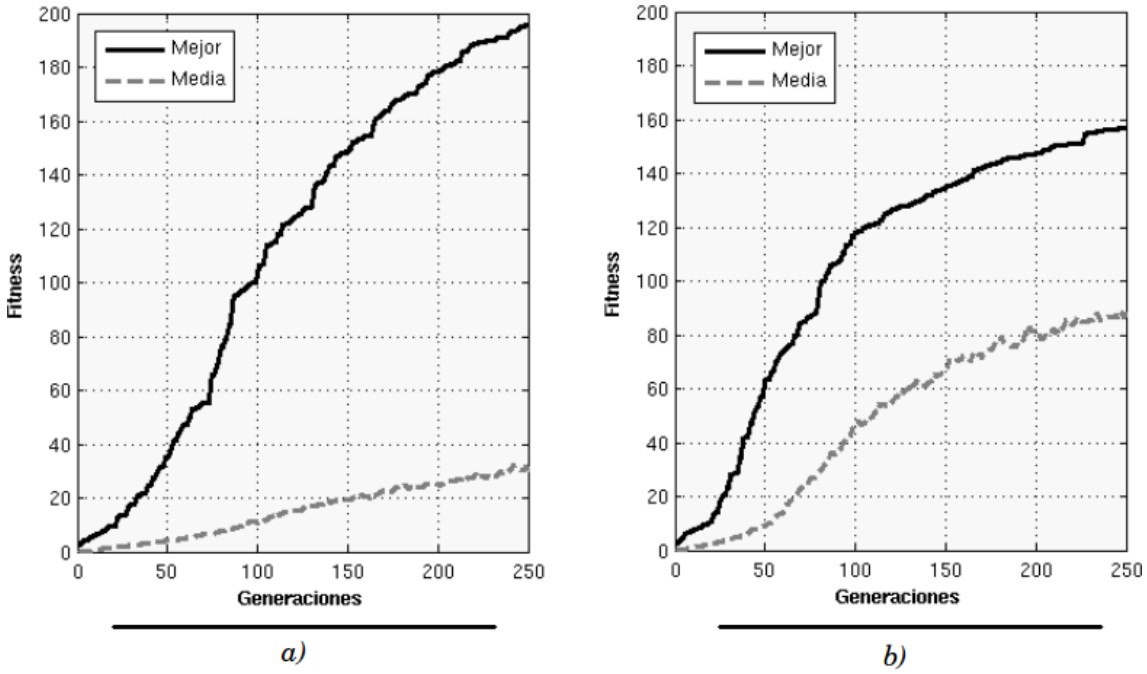

Figura 4.13: Gráfico del fitness del mejor individuo y del promedio de la población, de 50 ejecuciones independientes. a) Estrategia propuesta. b) Sin especiación ni operador de mutación estructurada.

largo de 250 generaciones. El gráfico *a)* muestra los resultados obtenidos con el método propuesto. Para el gráfico *b)* se utilizó una estrategia similar pero sin la especiación ni el operador de mutación de comportamiento simétrico, que son las dos estrategias principales introducidas en el algoritmo para escapar de los optimos locales que presenta la función objetivo.

Para analizar los gráficos es necesario recordar la naturaleza de la función de aptitud. La misma consta de dos etapas, una en la que los individuos aprenden a moverse y otra en la que, luego de llegar al objetivo y se especializan para hacerlo de forma más veloz. Al conseguir un controlador que llegue al objetivo, el valor del fitness se incrementa en 5 veces. De este modo se genera un salto abrupto en la curva del fitnes de cada individuo. Esto no se ve reflejado en las curvas de la figura 4.13 ya que es un promedio de muchas pruebas. Algo fundamental a considerar en la obtención de este controlador, es que no es de utilidad un proceso que otorgue grandes valores de fitness cuando el individuo no llega al objetivo buscado. En el caso del gráfico *a)* en todas las pruebas realizadas existía al menos un individuo que resolvía el laberinto, de modo que siempre se logró conseguir un controlador apropiado. En el caso del gráfico *b)*, no siempre se consiguió un individuo que encuentre un camino hasta el objetivo, aún teniendo valores de aptitud altos.

Estos resultados expuestos pueden ser deducidos si se realiza un análisis más riguroso de los gráficos. Algo que puede apreciarse a simple vista es que si bien la curva de los mejores individuos es superior con la estrategia propuesta, la curva del promedio de la población es notablemente inferior, comparándola con el gráfico *b)*. Este suceso es totalmente razonable. La estrategia convencional (gráfico *b)*), utiliza una presión selectiva mayor al no dividir la población en especies. De este modo, algunas pruebas resultan en individuos que logran encontrar el camino en el laberinto; el resto de la población sigue a estos individuos líderes y el fitness promedio se acerca considerablemente al fitness máximo. Otras pruebas, sin embargo, quedan atrapadas en un óptimo local de modo que el fitness no es totalmente bajo pero sí inferior a las pruebas que obtienen un controlador apropiado. La estrategia propuesta, en cambio, utiliza especies que exploran el espacio de búsqueda en diferentes zonas; si una especie queda atrapada en un óptimo local, no necesariamente el resto de la población debe seguirla. De este modo, mientras ciertas especies logran encontrar la zona del óptimo global (es decir, obtienen un controlador apropiado), otras quedan atrapadas en óptimos locales, o en zonas de bajo fitness. De este modo, el fitness promedio de toda la población es considerablemente bajo con respecto al fitness máximo.

Además de lograr eficientemente escapar de óptimos locales, la estrategia propuesta resulta muy rápida. Como se verá en capítulos siguientes, es posible obtener un controlador apropiado en apenas algunos minutos, lo cual resulta muy eficiente al momento de generar controladores que sirvan de módulos para la creación de un controlador con mayor poder resolutivo.

4.3. Conclusiones

En este capitulo se desarrolló un método eficiente y veloz capaz de obtener un controlador robótico para comandar un robot Khepera II. En este caso se utilizaron los problemas de evasión de obstáculos y alcance de objetivos, obteniendo resultados satisfactorios.

El hecho de utilizar una arquitectura neuronal de tamaño mínimo es ventajoso si se considera la velocidad de ejecución dentro del robot. Sin embargo el poder de resolución puede verse limitado dependiendo el problema a resolver. Así mismo, esta arquitectura puede ser utilizada como módulo para la creación de controladores más generales como se realizó en [45].

Los experimentos realizados mostraron la efectividad de las técnicas utilizadas y del operador de mutación implementado. La especiación muestra ser una novedosa técnica a la hora de escapar de óptimos locales, existiendo en la actualidad diversas variaciones posibles a aplicar.

Capítulo 5

Sistemas paralelos

Uno de los principales desafíos en la informática es reducir los tiempos de cómputo de los algoritmos. Desde sus orígenes, esta ciencia intenta realizar procesos automáticos de forma eficaz y veloz. En este sentido, existen nuevos algoritmos y computadoras cada vez más veloces. Sin embargo, aún con algoritmos inteligentes, técnicas de optimización como las metaheurísticas, y arquitecturas cada vez más rápidas, existen diversidad de problemas donde el tiempo de cómputo no resulta aceptable. Esta es una de las razones por las que existe la computación paralela en la cual no se utiliza un sólo procesador en la resolución de un problema, sino que se intenta aprovechar de forma eficiente varios de ellos [67][6].

Es posible definir un sistema paralelo como un algoritmo paralelo junto con una arquitectura subyacente, necesaria para que las aplicaciones funcionen, agregando también ciertos protocolos o mecanismos de comunicación. En este capítulo se desarrollan los aspectos teóricos más importantes en el estudio de sistemas paralelos.

5.1. Introducción

En las computadoras convencionales existen una única unidad de procesamiento, o cerebro, que se encarga de procesar la información de un algoritmo[1]. Un programa, o algoritmo, está compuesto por una serie de pasos discretos que son ejecutados uno detrás del otro. De este modo, el núcleo, o procesador de la computadora ejecuta una única instrucción por cada ciclo de procesamiento. Una forma de incrementar este poder es utilizar más de un procesador para realizar tareas que puedan llevarse a cabo concurrentemente.

[1]Se considera computadora convencional a las PCs del año 2005 hacia atrás, ya que en adelante, y hoy en día, es muy usual que haya computadoras con más de un núcleo y sistemas operativos que administren las tareas en ellos.

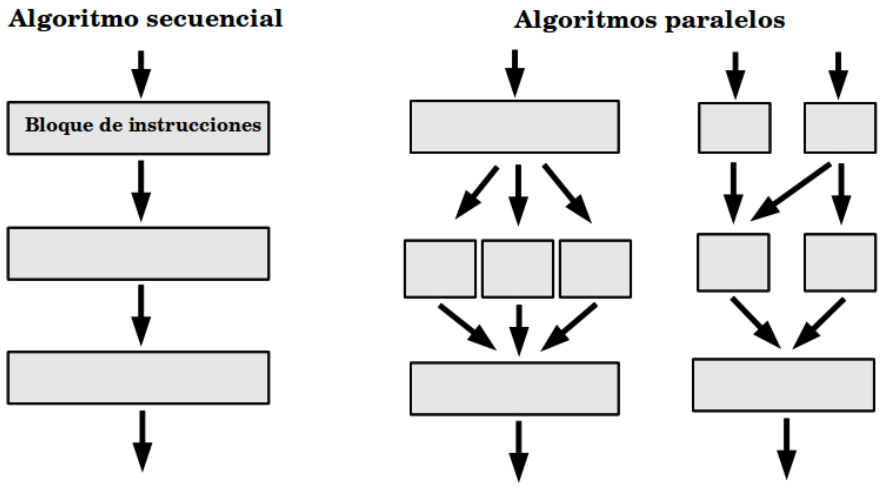

Figura 5.1: Esquemas de un algoritmo secuencial y dos ejemplos de algoritmos paralelos.

En este sentido, el diseño de un algoritmo paralelo consiste en identificar cierto grupo de instrucciones que pueden ejecutarse simultáneamente y establecer estrategias de sincronización entre estas. La figura 5.1 muestra un esquema comparativo entre el modo de funcionamiento de un algoritmo secuencial y algunos ejemplos paralelos. En el algoritmo secuencial las instrucciones se ejecutan una tras otra, en el paralelo existen bloques de código que pueden ejecutarse en diferentes unidades de cómputo.

En los últimos años se han desarrollado computadoras cada vez más rápidas, con más memoria y más potentes. Sin embargo existen límites físicos que se alcanzarán donde ya no se podrá seguir incrementando el poder computacional de forma directa sino que es necesario utilizar técnicas de software para lograrlo. Uno de estos límites es la velocidad de la luz. Si se incrementa el número de componentes en el hardware de una computadora las distancias de comunicación aumentan y del mismo modo el tiempo de cómputo; esto además agrega un aumento de la temperatura lo que lleva a técnicas más sofisticadas de refrigeración, por la imposibilidad de trabajar a ciertas temperaturas. Otro límite físico es la cantidad de transistores por milímetro cuadrado que pueden colocarse. Hoy en día ya se habla de transistores a nivel atómico. Desde este punto de vista, no se podrá aumentar el poder de cómputo cuando se alcance este límite físico.

En los últimos años, la tendencia ha sido desarrollar el paralelismo a nivel de hardware (unidades con varios núcleos, instrucciones en forma de pipelines, etc.) [6], así como también el desarrollo de nuevas técnicas en algoritmos paralelos.

Casos de aplicación de programación paralela

Existen diversas razones por las cuales es conveniente utilizar una estrategia paralela, entre las que se puede nombrar:

- **Problemas de tamaños excesivos.** Muchos problemas tienen un domino demasiado grande para explorar, lo que los hace inabordables con computadores convencionales si se desea tener resultados en tiempos razonables.
- **Reducir el tiempo de ejecución.** En muchas ocasiones es necesario reducir el tiempo de ejecución de un programa, ya sea por simple competencia o por un límite temporal en el problema particular. Diversos problemas reales requieren límites de tiempo fijos para la toma de decisiones rápidas, o para el desarrollo de productos. Un ejemplo de esto son los procesos de simulación, donde estos deben ejecutarse en un tiempo acotado como para que el diseñador pueda trabajar sobre los datos obtenidos [67].
- **Resolver problemas con mayor precisión.** Si un problema no presenta una limitación temporal grande, otra característica que presentan los sistemas paralelos es que posibilitan el aumento de datos de entrada para el mismo tiempo de cómputo. Es decir, que es posible procesar más datos, y posiblemente conseguir mayor precisión en los resultados. Por ejemplo, en un algoritmo evolutivo, sería posible mantener el tiempo del algoritmo secuencial aumentando considerablemente el número de individuos en la población. De este modo se obtiene mayor variabilidad genética.

Terminología y conceptos básicos

A continuación se listan una serie de términos comunes en el área de la programación paralela:

Tarea. Una tarea es una unidad de trabajo que normalmente se asigna a un procesador. Un programa normalmente es dividido en tareas, donde se establece un grafo de dependencias entre ellas.

Dependencia. Cuando dos tareas poseen una dependencia significa que deben ser ejecutadas en un orden específico. Por ejemplo una tarea podría requerir de ciertos datos que debe procesar otra tarea primero. Las dependencias limitan fuertemente el grado de paralelismo.

Método de descomposición. Es la forma en que un problema, o algoritmo secuencial, es dividido para generar tareas que pueden ejecutarse concurrentemente.

Proceso e hilo. Son unidades de trabajo. Cada tarea es asignada (o *mapeada*) a un proceso o hilo, el cual contiene un flujo de control propio. Los hilos normalmente son utilizados en modelos de programación con memoria compartida, mientras que los procesos se utilizan en modelos distribuidos, o de pasaje de mensajes. De ahora en más se llamará *proceso* indistintamente para identificar ambos conceptos.

Granularidad. Este término establece la relación entre la cantidad de tareas y el volumen de información que debe procesar cada una de ellas. Una descomposición con muchas tareas de poco cómputo y muchas comunicaciones se suele llamar de *grano fino*. Por el contrario, una descomposición con pocas tareas con gran volumen de datos, las cuales requieren poca comunicación, se denomina de *grano grueso*.

Inconvenientes en la programación paralela

A la hora de desarrollar una aplicación paralela, o dividir un programa secuencial en tareas con ciertas dependencias, existen dificultades inherentes a la programación paralela que no pueden evitarse.

Existen infinidad de problemas en los que la paralelización resulta trivial de implementar. Por ejemplo, el problema consta de diversas operaciones independientes que pueden ser evaluadas simultáneamente sin ningún tipo de restricción. En estos casos la división en tareas es simple y posiblemente el algoritmo paralelo elegido también sea de fácil implementación. Sin embargo existen muchas aplicaciones en las que no es fácil encontrar una división en tareas paralelas, o directamente es imposible, ya que siempre una tarea necesita de la anterior. Si bien existen diversos métodos de descomposición genéricos o específicos, el trabajo de descomponer un problema en tareas que puedan ser ejecutadas concurrentemente depende del programador.

A su vez, escribir un programa paralelo implica cierta complicaciones que no están presentes en el algoritmo secuencial. Por ejemplo, es posible que para el desarrollo de la aplicación paralela se necesite un lenguaje de programación específico, con el que quizá no se esté familiarizado. Además, las aplicaciones paralelas suelen tener instrucciones extras para la sincronización y comunicación entre los diferentes procesos. Esto implica tener conocimiento específico de técnicas de programación paralela. Todos estos procesos no solo son complejos sino que agregan una gran propensión a errores. Una mala descomposición de las tareas podría hacer que el rendimiento global sea pobre, incluso peor que un algoritmo secuencial.

El testeo de programas paralelos, es decir la prueba de su correcto funcionamiento, es también un problema. Al tener instrucciones concurrentes se genera no determinismo, lo cual conlleva a que dos ejecuciones de un mismo programa deriven en resultados diferentes aunque la información inicial (datos de entrada) sea la misma.

Por último, una gran limitación que presentan las aplicaciones paralelas hoy en día

es su dependencia del hardware. Mientras que la mayoría de aplicaciones o lenguajes de programación han logrado desarrollarse exitosamente en modo multiplataforma, muchas aplicaciones paralelas son desarrolladas específicamente para cierta arquitectura y no pueden ser trasladadas a otra. O en caso de ser trasladadas, su rendimiento puede caer considerablemente. Esto lleva a que el desarrollo de sistemas paralelos sea hoy en día un área de diversos estudios, sobre todo a nivel de arquitecturas y lenguajes de programación. Actualmente, es muy usual que una PC hogareña posea más de un núcleo, en una arquitectura con memoria compartida. Esto hace que el estudio y desarrollo de sistemas paralelo en estas arquitecturas sea muy amplio y popular.

5.2. Arquitecturas paralelas

Al momento de desarrollar una solución paralela a un problema particular es necesario decidir sobre qué arquitectura se va a trabajar. Como ya se dijo anteriormente, uno de los problemas de la programación paralela es su fuerte dependencia a la arquitectura o plataforma subyacente.

A lo largo de los años se han desarrollado numerosos modelos de arquitecturas paralelas cambiando diferentes componentes en ellas, como el número de procesadores, las redes de interconexión, los tipos de procesadores, etc. De modo que resulta muy difícil establecer una única clasificación. Existen diversas clasificaciones según el criterio a analizar. Una de las más comunes es la clasificación de Flynn, basada en la cantidad de flujo de instrucciones y datos durante la ejecución de un programa [66]. Otra clasificación muy usual es la basada en el modo de acceso a la memoria y en la interconexión de las unidades de procesamiento. En este caso se puede diferenciar entre plataformas de memoria compartida, plataformas de memoria distribuida y algunos sistemas híbridos. En el resto de la sección se detalla esta clasificación.

5.2.1. Plataformas de memoria compartida

Los sistemas multiprocesador de memoria compartida (o también llamados simplemente *multiprocesador*) son máquinas con varias Unidades de Procesamiento (UP) y una o más unidades de memoria comunes a las UPs. Es decir, hay un único espacio de direcciones visible por todas las UPs. Estas arquitecturas suelen ser muy atractivas para los programadores ya que resulta muy sencillo hacer uso y manejo de variables. Los módulos de memoria compartida resultan difíciles de construir lo que se refleja en un alto costo. Sin embargo presentan arquitecturas muy similares a los sistemas monoprocesador, y es por esta razón que muchos fabricantes ofrecen alternativas multiprocesador (normalmente 2 o 4 UPs) a sus tradicionales monoprocesador.

El esquema de acceso a este tipo de arquitectura puede caer dentro de tres

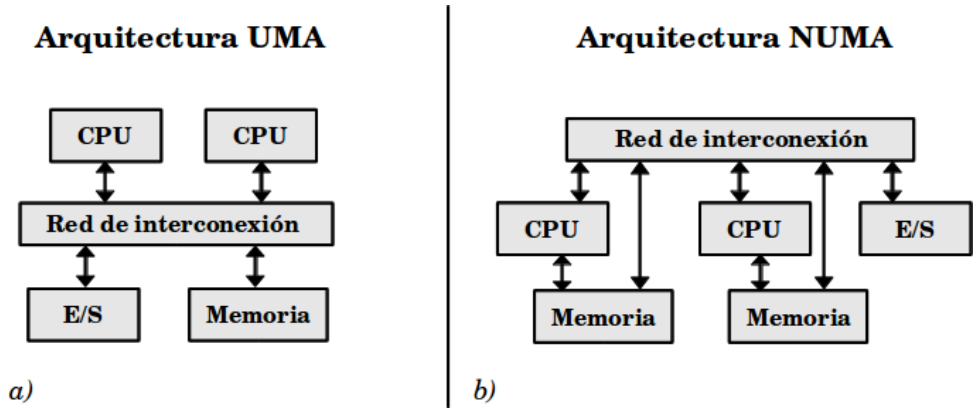

Figura 5.2: Esquemas de arquitecturas de memoria compartida.

alternativas descritas a continuación [19][29]:

- **UMA** (Uniform Memory Access). En estos casos los procesadores acceden con igual facilidad (mismo tiempo) a cada una de las celdas de memoria (figura 5.2-a) . Estos sistemas resultan muy similares a los monoprocesadores, de modo que resulta sencillo comunicar procesos cooperativos a través de la memoria común.

- **NUMA** (Non Uniform Memory Access). A diferencia del modelo anterior, en este caso cada procesador posee una memoria local, de modo que accede más rápido a ella que a las demás (figura 5.2-b). Es necesario que cada instrucción indique si se está haciendo referencia a la memoria local o global.

- **COMA** (Cache Only Memory Architecture). En estas arquitecturas las UPs siempre acceden a los datos a través de sus caches, y es necesario establecer mecanismos de actualización y coherencia. Este modelo puede ser considerado como un caso especial de NUMA, donde cada memoria local es una cache.

Si bien generalmente es preferible una arquitectura UMA, la mayoría de los sistemas multiprocesadores utilizan un modelo NUMA, debido a la dificultad de implementar hardware de rápido acceso a toda la memoria. Por esta razón, muchos sistemas utilizan modelos de jerarquías de memorias [67].

Uno de los modelos más desarrollados y difundidos en la actualidad, que utilizan memoria jerárquica son los *multicores* o *multinúcleos*. Aquí cada núcleo, además de poder acceder a la memoria global, posee un nivel interno de cache (incluso pueden existir otros niveles con cache compartida). Aquí se hace necesario tener hardware específico para manejar uno de los grandes problemas de la arquitectura de computadoras, que es la coherencia de cache.

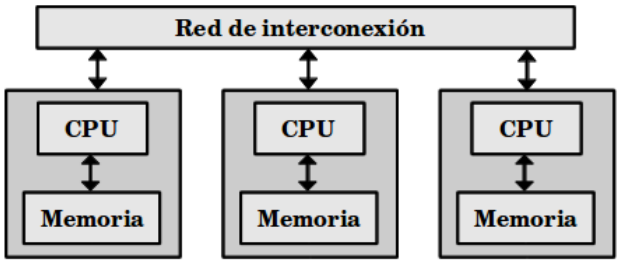

Figura 5.3: Esquema de una arquitectura de memoria distribuida.

5.2.2. Plataformas de memoria distribuida

Un sistema de memoria distribuida consta de diversas unidades de procesamiento cada una con sus propios espacios de direcciones interconectados por algún tipo de red. Cada unidad de procesamiento es una computadora es sí misma. El paradigma utilizado para programar este tipo de arquitectura es el pasaje de mensajes, donde estos se utilizan para intercambiar datos, transferir trabajo o sincronizar procesos [29] (este paradigma se detalla en la sección 5.4). Los sistemas de memoria distribuida resultan más fácil de escalar en comparación con los de memoria compartida [67]. La figura 5.3 muestra un esquema de alto nivel de una arquitectura con memoria distribuida. Ejemplos de este tipo de arqutiecturas son los clusters, multiclusters y GRIDs. A continuación se describen los clusters dado que es el tipo de arquitectura utilizada.

Clusters

Un cluster es un conjunto de computadoras conectadas de alguna forma con el fin de ser utilizadas como una única unidad de procesamiento. Esquemáticamente puede verse como un grafo donde cada nodo es un sistema de procesamiento en sí mismo, y las aristas representan algún tipo de red de interconexión. Estas redes de interconexión generalmente son redes LAN o WAN tradicional. Cada unidad de procesamiento puede tener su propio hardware y software idependiente de las demás unidades. Aquí se establece una clasificación fundamental a la hora de programar sobre un cluster:

- **Cluster homogéneo.** En un cluster homogéneo todas las computadoras que pertenecen al conjunto presentan las mismas características de hardware.
- **Cluster heterogéneo.** Aquí, cada unidad de procesamiento puede diferir en el hardware. Esto presenta un gran desafío al momento de programar, ya que habrá unidades más rápidas y con diferente tamaños de memoria que otras.

Una particularidad de los clusters es que presentan altas prestaciones a muy bajo costo. Las unidades de procesamiento pueden ser simples PCs convencionales conectadas por una red LAN. Esto hace que su precio sea realmente bajo comparándolos con otros tipos de plataformas de procesamiento paralelo. Al mismo tiempo, los clusters presentan una alta escalabilidad, ya que es muy sencillo instalar un nuevo nodo en el grafo del sistema, sobre todo en clusters heterogéneos.

Si bien los sistemas de clusters presentan ciertas dificultades para el programador que no están presentes en plataformas de memoria compartida, la potencia computacional suele ser mucho mayor, sobre todo por el rendimiento costo-beneficio del que se habló en el párrafo anterior.

El modelo de programación utilizado en sistemas de memoria distribuida, y particularmente en clusters, es el paradigma de pasaje de mensajes (sección 5.4). Si bien este paradigma es muy flexible, presenta el inconveniente que toda comunicación a llevar a cabo entre procesos deber ser hecha por medio de un mensaje a través de la red de interconexión. Una de las mayores fuentes de overhead en aplicaciones paralelas proviene de estos mensajes. El costo de la comunicación de cada mensaje depende de diversos factores, entre los que se pueden nombrar el modelo de programación, la topología de red utilizada, el método de asignación de ruta, etc. [29].

De forma general, el costo de comunicar un mensaje entre dos nodos está dado por el costo de la preparación del mensaje sumado a la transferencia propiamente dicha. En esta ecuación se ven involucrados el tiempo de latencia (tiempo que toma un mensaje en saltar de un nodo a otro) y el tiempo de transferencia por palabra. Existen diversas técnicas para eliminar el overhead comunicacional basándose en la reducción de estos tiempos. Por ejemplo, una técnica posible consiste en agrupar varios mensajes que se envíen al mismo nodo y enviarlos todos juntos, reduciendo así el tiempo de inicio a un sólo mensaje. Existen también algoritmos de ruteo óptimo y técnicas que dividen un paquete en partes para ser mandadas eficiéntemente.

5.2.3. Esquemas híbridos

Debido a que los sistemas de memoria compartida son más sencillos para el programador, y que los sistemas con memoria distribuida presentan mayor poder computacional a un menor costo, existen también algunos sistemas híbridos. Un ejemplo son los sistemas físicamente distribuidos pero lógicamente compartidos. En este caso el sistema operativo (o algún tipo de capa de abstracción) muestra el sistema como una gran memoria compartida cuando en realidad se compone de diversos nodos. Cuando una unidad necesita una posición de memoria que se encuentra en alguna otra terminal, se utiliza algún protocolo de pasaje de mensajes, de forma transparente al programador [3]. De modo que estos sistemas utilizan arquitecturas de memoria distribuida pero se comportan como si fuesen de memoria compartida. Otra arquitectura

que se está desarrollando mucho en la actualidad es el cluster multicore. Estos son simplemente extensiones de un cluster tradicional, donde cada nodo es una unidad multicore. Aquí, no se puede obviar la jerarquía de memoria al momento de programar una aplicación.

5.3. Métricas de rendimiento en sistemas paralelos

Al crear un programa paralelo resulta de gran importancia hacer un detallado análisis del rendimiento del mismo. Como ya se dijo, esto puede ser un problema debido al no determinismo de los mismos. Para lograrlo es necesario realizar un correcto estudio de la plataforma de hardware utilizada, así como del comportamiento del algoritmo para diferentes situaciones o volúmenes de información. Para llevar a cabo este análisis existen diversas métricas que evalúan diferentes aspectos del rendimiento de un sistema [29].

Una de las principales métricas utilizadas es el tiempo de ejecución paralelo. Esta medida sirve para comparar diferentes soluciones paralelas a un mismo problema. En general el tiempo de ejecución está dado en función del tamaño de entrada del problema, que se define como la cantidad de operaciones básicas realizadas por el mejor algoritmo secuencial [3]. De este modo, el tiempo de ejecución paralelo no es un indicador objetivo de qué tan bueno es un algoritmo, ya que no incluye en su análisis a la arquitectura utilizada, o ningún otro tipo de parámetro que indique la calidad del algoritmo.

En el resto de la sección se discuten 3 de las principales métricas utilizadas con frecuencia en sistemas paralelos que complementan el tiempo de cómputo paralelo con análisis más objetivos.

5.3.1. Overhead

El overhead, o también llamado *overhead paralelo*, es una medida del tiempo que un programa paralelo desperdicia [29]. Además del tiempo consumido en computar el problema, un programa paralelo puede gastar tiempo en diversas razones, las cuales generan overhead:

- **Comunicación entre procesos.** En casi todo programa paralelo es necesario realizar comunicaciones entre procesos. Esta es una de las principales causas de overhead. Eventualmente estas comunicaciones pueden ser tan costosas que el tiempo paralelo no reduce al secuencial.

- **Ocio.** En ocasiones un proceso espera que otro termine para seguir su ejecución. Esto puede deberse simplemente al modo de interacción de los procesos o a un desbalance de carga debido a que ciertos procesadores son más rápidos que

otros, o tenían menos trabajo por hacer. En estos casos el tiempo de cómputo desaprovechado genera overhead al sistema total.

- **Exceso de computación.** En algunos casos no es posible paralelizar la mejor solución secuencial a un problema particular. Siendo así, se utiliza una solución no tan buena, lo cual lleva a tener exceso de cómputo en cada proceso. También puede haber un exceso de computación al agregar operaciones extras como réplicas del mismo algoritmo (muy utilizado en metaheurísticas paralelas), o cálculos adicionales necesarios en la solución paralela (como por ejemplo hacer un balance de carga entre procesos).

El overhead se puede definir según la siguiente ecuación:

$$T_o = pT_p - T_s \tag{5.1}$$

donde T_p es el tiempo paralelo, p es la cantidad de procesadores y T_s es el mejor tiempo secuencial.

5.3.2. Speedup

Una de las métricas más utilizadas en sistemas paralelos es el *Speedup* [29]. Esta métrica indica una medida de cuánto mejora el tiempo paralelo en comparación con el secuencial. Es decir que indica el beneficio relativo de realizar la paralelización de un algoritmo secuencial. Para llevar a cabo este cálculo se compara el mejor tiempo obtenido del mejor algoritmo secuencial con una serie de ejecuciones paralelas, dependiendo la arquitectura y métodos usados.

Puede definirse al speedup según la siguiente ecuación:

$$S = \frac{T_s}{T_p} \tag{5.2}$$

De este modo, a medida que se consigue un menor tiempo de ejecución paralelo, el speedup aumenta.

Al incrementar el número de procesadores se espera que el tiempo de ejecución baje y por consiguiente el speedup aumente. Esto no es necesariamente así, debido al overhead paralelo, como ya se explicó. Suponiendo un caso ideal, el tiempo paralelo es T_s/p, donde p indica la cantidad de procesadores utilizados. Dependiendo del comportamiento del tiempo paralelo al agregar procesadores, puede clasificarse al speedup en tres casos:

- **Speedup lineal.** Este es el caso ideal, por cada procesador que se agrega, el speedup aumenta en uno, formando una curva identidad.

- **Speedup sublineal.** Este es el caso más usual. Debido al overhead, el tiempo paralelo no siempre es T_s/p, sino mayor. De forma que agregar más procesadores implica una ganancia cada vez menor de tiempo, y el speedup se comporta de forma logarítmica.

- **Speedup superlineal.** Este es un caso raro, pero que puede darse. El speedup aumenta por encima de la curva identidad por cada procesador que se agrega. Esto puede deberse a cuestiones de memoria/cache de la arquitectura utilizada como también al problema específico que se está resolviendo, donde al dividir el problema, la cantidad de cómputo necesaria disminuye en un grado mayor. Un ejemplo de esto es la búsqueda en un árbol [29], donde al dividir el problema en diferentes procesadores, el trabajo total es menor al que tendría que haber hecho un sólo procesador.

Es normal que partes de un algoritmo secuencial no puedan ser divididas para ser procesadas en paralelo. En este caso es necesario que esos fragmentos de código sean ejecutados secuencialmente, lo que implica que un sólo procesador trabaje mientras los demás están ociosos. Esto, además de generar overhead, permite deducir cierta propiedad conocida como la *ley de Amdahl*. Si f representa la fracción de tiempo que el programa no puede ser paralelizable, esta ley indica que el máximo speedup alcanzable está dado por $1/f$. Es decir, que no importa cuántos procesadores se agreguen, no será posible superar ese umbral en el speedup. Más allá de esta propiedad, en muchas ocasiones aumentar considerablemente el número de procesadores puede implicar un exceso de overhead debido a la comunicación, lo cual trae un decremento importante del speedup. La figura 5.4 muestra los diferentes estados que puede tener el speedup a medida que se incrementa el número de procesadores.

5.3.3. Eficiencia

Como ya se dijo antes, en general un programa no hace un uso totalmente eficiente de sus procesadores, sino que pierde tiempo en comunicar procesos o en espera de los mismos. Esto genera un overhead paralelo que se refleja en el speedup. La eficiencia denota la cantidad de tiempo que cada procesador es usado eficientemente. Es decir, que la eficiencia es una medida del buen uso que se le da a los procesadores, y se define mediante la siguiente ecuación [29]:

$$E = \frac{S}{p} \tag{5.3}$$

De modo que la eficiencia es una relación entre el speedup S y la cantidad p de procesadores usados. Esta ecuación puede tomar valores entre 0 y 1. En el caso ideal

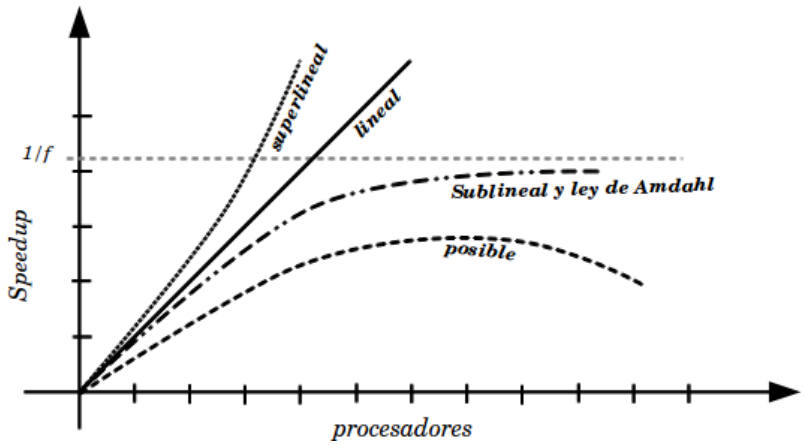

Figura 5.4: Posibles estados del speedup.

de tener un speedup lineal, la eficiencia toma su máximo valor, lo que significa que cada procesador está siendo eficientemente utilizado. En la mayoría de los casos la eficiencia será un valor inferior a 1, ya que el speedup suele ser sublineal. En caso extremo de tener un speedup superlineal la eficiencia toma valores mayores a 1.

5.4. Diseño de algoritmos paralelos

Al resolver un problema mediante un algoritmo secuencial se suele identificar una serie de pasos a seguir. Sin embargo, al diseñar un algoritmo paralelo es necesario considerar otras cuestiones no presentes en los algoritmos secuenciales, como la identificación de porciones de código paralelizables, asignación de trabajo a procesadores, administración de acceso a datos compartidos, sincronización de procesos, etc. [29]. En esta sección se analizan las principales técnicas para el diseño de algoritmos paralelos.

5.4.1. Técnicas de descomposición

Una de las principales tareas a realizar al diseñar un algoritmo paralelo es la descomposición que se haga del problema, para poder procesarse de forma concurrente. No existe una única técnica para realizar esto de forma eficiente, de modo que la correcta elección de la técnica usada dependerá del problema específico y queda en manos del diseñador del algoritmo.

Una de las principales clasificaciones se basa en el tipo de paralelismo utilizado. En tal caso se divide en *paralelismo funcional* y *paralelismo de datos* [15].

Paralelismo funcional

La idea de esta descomposición consiste en identificar funciones específicas que puedan ser realizadas por procesos diferentes. De este modo se crean diversas tareas con su grafo de dependencias. Luego, las tareas que no posean dependencias entre ellas pueden ejecutarse en forma paralela.

Paralelismo de datos

Este tipo de descomposición consiste en dividir el volumen de datos en diferentes partes y procesar cada una en distintas tareas. Este modelo muestra una forma natural de dividir y escalar un problema. Dentro de esta clasificación, es posible distinguir distintos métodos, algunos de propósito general y otros diseñados para cierta clase de problemas [29]:

- **Descomposición recursiva.** Esta técnica provee de concurrencia a procesos que pueden resolverse con una estrategia *divide y vencerás*. Es decir, problemas en donde es posible resolverlos en forma recursiva. Se divide al problema en partes de naturaleza muy similar al original pero de tamaño inferior, se procesan estas partes, y por último se combinan para generar una solución al problema inicial. Existen numerosos problemas que pueden ser resueltos mediante esta descomposición. Un método muy conocido al que se le puede aplicar esta técnica es el método de ordenamiento de vectores *quicksort*.

- **Descomposición de datos.** Es un método muy potente utilizado generalmente para generar concurrencia en problemas que trabajan con grandes estructuras de datos. El método se centra en particionar los datos del problema para poder procesarlos en tareas paralelas. El particionamiento puede ser de los datos de entrada, de los datos de salida, o de ambos.

- **Descomposición exploratoria.** Este método es utilizado en problemas combinatorios, donde se busca una solución en un conjunto de soluciones posibles independientes entre sí. En la descomposición exploratoria se divide el espacio de búsqueda en partes más pequeñas y se realiza una búsqueda concurrente de la solución. La búsqueda continúa hasta encontrar la solución deseada o hasta que se explore todo el espacio de soluciones. Problemas típicos que pueden resolverse con este método son el problema del juego *Ta-Te-Ti*, o el problema del *N-puzzle*.

- **Descomposición especulativa.** En muchas ocasiones no es posible encontrar a priori una división en tareas que puedan ejecutarse concurrentemente. La descomposición especulativa ejecuta tareas paralelas que procesan posibles ramas computacionales en un programa. Al llegar un momento en el programa en que

se debe tomar una decisión sobre qué camino tomar, los mismos ya se encuentran procesados, de forma que las demás computaciones son descartadas. Un ejemplo de esta aplicación es una instrucción *case of*, donde se evalúa una expresión y se ejecuta una rama de todas las posibles dependiendo del valor de la expresión. Aquí, diferentes tareas pueden computar las distintas ramas posibles, y otra tarea evaluar la expresión. Si bien esta estrategia de descomposición ahorra mucho tiempo, la eficiencia del sistema no es bueno, ya que gran parte del procesamiento se está desperdiciando.

5.4.2. Métodos de mapeo

Luego de descomponer un problema en tareas, un paso muy importante es asignar adecuadamente cada una de estas a los diferentes procesadores de los que se disponga. Esta asignación se conoce con el nombre de *mapeo*, e intenta minimizar el tiempo paralelo final.

La correcta elección del método de mapeo utilizado queda en manos del programador, pero también queda determinado por el modelo de programación utilizado (ver sección 5.4.4), las características de las tareas y la interacción entre las mismas [29]. Las diferentes técnicas pueden caer sobre tres principales categorías:

- **Mapeo estático.** Esta técnica asigna las diferentes tareas a los procesadores antes de comenzar la ejecución del programa. Para esto es necesario conocer el número y tamaño de las tareas. Este tipo de mapeo se utiliza generalmente en problema de descomposición de datos.

- **Mapeo dinámico.** Aquí, las tareas son asignadas a los distintos procesadores en tiempo de ejecución. Si las tareas son creadas dinámicamente (es decir que no se conoce de antemano el número de tareas) este método es el único que puede emplearse. En muchas ocasiones es más conveniente utilizar un modelo de mapeo estático, sobre todo si el volumen de datos a procesar es muy grande. Sin embargo para algunos problemas es conveniente utilizar un modelo dinámico. Esta técnica es muy utilizada en sistemas de memoria compartida, donde los hilos son creados dinámicamente para propósitos específicos.

- **Mapeo jerárquico.** Esta técnica es muy utilizada en problemas donde las tareas son expresadas como un grafo de dependencias. Es muy usual que mapear estáticamente las tareas en base al grafo de dependencias genere un gran desbalance de carga. Por ejemplo, al tener un grafo de dependencias en forma de árbol y asignar cada proceso a un procesador diferente, sólo habrá una gran concurrencia el procesar las hojas del árbol, pero al comenzar, la raíz se ejecuta en forma independiente, desaprovechando

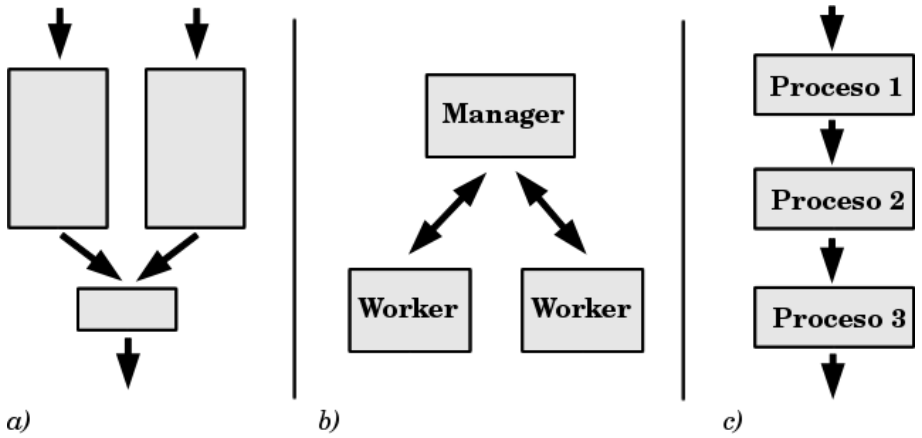

Figura 5.5: Paradigmas de interacción entre procesos paralelos. a) Datos paralelos. b) manager-worker. c) Productor-consumidor

recursos. El mapeo jerárquico resuelve parcialmente este problema, dividiendo ciertas tareas en tareas más pequeñas.

5.4.3. Paradigmas de interacción entre procesos

Dependiendo del problema que se desea resolver y la arquitectura con la que se cuenta, se pueden utilizar diferentes paradigmas o modelos de algoritmos paralelos que se adecuan mejor a dicho problema. Estos modelos permiten dar estructura a los algoritmos, y son la combinación de las técnicas de descomposición y mapeo elegidas junto con alguna técnica para reducir el overhead. Existen diversos modelos, algunos genéricos y otros más específicos. A continuación se detallan algunos de los más conocidos [29]:

Modelos de datos paralelos

Es uno de los modelos más simples. Se utiliza alguna descomposición de datos y se mapea estáticamente cada tarea a un procesador donde cada una realiza las mismas operaciones. Esta descomposición uniforme es de utilidad cuando es posible realizar una descomposición que balancee la carga de trabajo. Puede implementarse este modelo tanto en memoria compartida como distribuida. La figura 5.5-a ilustra este modelo.

Modelo manager-worker

En este modelo existen dos tipos de procesos: *manager* (generalmente es un sólo proceso) y *worker*. El proceso manager es el encargado de entregar trabajo y

administrarlo, mientras que los diferentes workers procesan esta información entregando los resultados nuevamente al manager. La asignación de tareas puede ser estática o dinámica, aunque esta última suele ser la más común. La figura 5.5-b muestra un ejemplo de este modelo con un manager y dos workers.

Un aspecto muy importante a considerar en estos modelos es la granularidad elegida. Si existen demasiados workers por cada manager, este podría saturarse y convertirse en un cuello de botella. Lo mismo sucede si la cantidad de trabajo que se entrega es demasiado escaso. De modo tal que la granularidad elegida debe generar un costo de trabajo mayor al de las comunicaciones.

Modelo productor-consumidor

También conocido con el nombre de *pipeline*, este modelo se basa una cadena de procesos donde cada uno es productor y consumidor. En general, al recibir información, un proceso la computa y la envía hacia adelante, donde otro la recibe. En este modelo cada proceso efectúa una función diferente, actuando como un filtro sobre los datos. La figura 5.5-c ilustra este modelo en un ejemplo con tres procesos.

Modelo híbrido

En muchos casos no es posible aplicar un solo modelo a un problema, sino que son necesarios varios de ellos, dando por resultado un modelo híbrido, o compuesto. En este caso se relaciona a los modelos de alguna forma jerárquica o secuencial.

5.4.4. Modelos de programación paralela

Cuando se habla de programación paralela explícita (es decir el paralelismo implementado por un programador) existen diversas librerías y lenguajes de programación desarrollados para diferentes modelos de arquitecturas paralelas. Esto lleva a diferentes modelos de programación. La principal diferencia en estos modelos es la forma en que el usuario ve el espacio de direcciones. De tal modo se distingue entre programación para plataformas con memoria compartida y programación para plataformas con memoria distribuida [29].

Modelos de memoria compartida

Al programar en un ambiente con memoria compartida el espacio de direcciones es único para todos los procesos que se creen. El principal modelo de programación en estas arquitecturas son los módulos basados en hilos. En este modelo, cada proceso puede tener múltiples flujos de procesamiento llamados hilos (o procesos livianos). Cada hilo

se mapea a diferentes procesadores, dependiendo la cantidad de estos y del tratamiento que haga con los hilos el sistema operativo. De este modo puede realizarse de forma automática por el sistema operativo, o de forma manual por el programador.

Al utilizar variables compartidas, uno de los principales problemas es el acceso a los datos. Varios procesos pueden querer acceder a una variable al mismo tiempo lo que puede generar inconsistencias, o comportamientos inesperados. De modo que es necesario establecer mecanismos de sincronización. En general se utilizan estructuras especiales como los semáforos (variables *mutex*) o los monitores. Una de las librerías más conocidas para el uso de threads (hilos) es *Pthreads* [3][29]. Esta librería, que data de 1990, es un estándar pensado para poder utilizarse en diferentes sistemas ofreciendo una buena portabilidad.

Otra posibilidad que existe al programar en arquitecturas de memoria compartida son los modelos basados en directivas. En este caso, el programador no debe estar al tanto de la creación y destrucción de los hilos de ejecución, sino que una librería específica se encarga de manejar estos factores. El trabajo del programador recae sólo en indicar mediante directivas lo que se quiere paralelizar. Una de las librerías estándar más utilizadas en este caso es *OpenMP* [3][29].

Modelos de memoria distribuida

Al utilizar una plataforma con memoria distribuida, los procesos no pueden acceder a datos que estén manejando otros procesos. Es por esta razón que la comunicación entre procesos se da utilizando un protocolo basado en mensajes [29].

La comunicación por mensajes trae tanto ventajas cono problemas. Cada vez que dos procesos necesitan intercambiar información deben enviar un mensaje. Si bien los mensajes no siempre proveen una forma fácil y natural de resolver un problema, implementan un mecanismo eficiente ya que el programador es consiente del intercambio de datos. De esto modo, el acceso a los datos es más seguro y estructurado. Además, los costos de las comunicaciones son fácilmente calculables. A diferencia de algunos paradigmas de memoria compartida, este paradigma obliga al programador a codificar explícitamente la comunicación y sincronización. Esto demanda un gran trabajo intelectual pero con la ventaja de que los programas suelen ser eficientes y fácilmente escalables.

Si bien este paradigma permite la ejecución de programas diferentes sobre las unidades de procesamiento, la mayoría de los programas desarrollados utilizan un enfoque SPMD (*Single Program Multiple Data*) [66]. Es decir que todos los procesos utilizan el mismo código pero trabajan con diferentes datos, locales a cada unidad de procesamiento. Para diferenciar el comportamiento de los diferentes procesos se suele utilizar alguna instrucción de selección tipo *if* o *case of*.

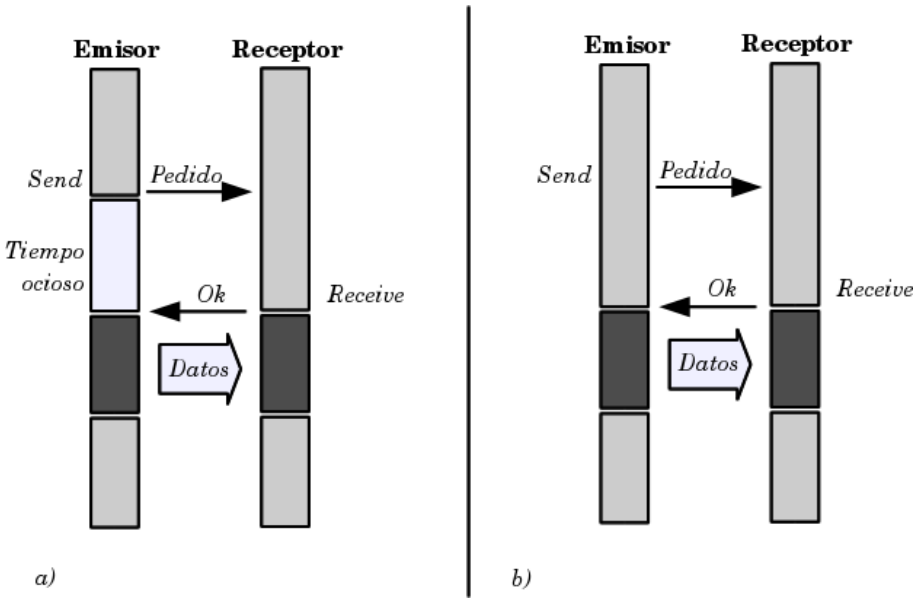

Figura 5.6: Mensajes *Send-Receive* entre dos procesos. a) Bloqueante. b) No bloqueante.

Existen diversos tipos de comunicaciones de acuerdo al lenguaje que se esté usando así como a la necesidad del problema. Las dos principales sentencias de comunicación utilizadas son el *Send* (que envía información) y el *Receive* (que recibe información). En cada una de estas sentencias se suele enviar con parámetros el emisor y/o receptor del mensaje, así como también los datos enviados. Estas operaciones tienen dos variantes esenciales: bloqueantes y no bloqueantes. En la primera, cuando un proceso llega a un mensaje *send* o *receive* queda bloqueado hasta que la operación se complete. En cambio, con mensajes no bloqueantes los procesos pueden continuar su ejecución luego de pasar por una de estas instrucciones, aunque el otro proceso no haya llegado. La figura 5.6 muestra la diferencia entre estos dos comportamientos. Existen otros tipos de sentencias de comunicación además de los descritos aquí que permiten diferentes tipos de comunicación como sentencias de sincronización, sentencias colectivas (en la comunicación participan varios procesos), etc.

Una de las librerías más utilizadas en el paradigma de pasaje de mensajes es MPI [29]. Esta define un estándar del paradigma que puede ser utilizado en C o Fortran. MPI define la sintaxis y la semántica de diversas instrucción que facilitan la comunicación por mensajes en un ambiente distribuido.

Si bien el paradigma de pasaje de mensajes es el más utilizado en ambientes distribuidos, existen otros modelos como el RPC (muy utilizado en programación Web), y el Rendezvous (principal mecanismo de sincronización/comunicación entre tareas del lenguaje ADA).

Capítulo 6

Paralelización de la estrategia propuesta

Uno de los principales problemas en los algoritmos evolutivos y sobre todo en Robótica Evolutiva, es el tiempo de convergencia. En algoritmos como el presentado en el capítulo 4 el proceso de simulación puede ser extremadamente lento y llevar a un tiempo de cómputo de horas o incluso días, dependiendo del problema específico y los tamaños de población utilizados.

En este capítulo se desarrolla una implementación paralela de la estrategia evolutiva planteada y se analiza su rendimiento con diferentes parámetros.

6.1. Algoritmos evolutivos paralelos

Se han desarrollado diversos estudios en el área de los algoritmos evolutivos paralelos (sobre todo para Algoritmos Genéticos). Este tipo de metaheurísticas resulta muy fácil de paralelizar, en parte gracias a su naturaleza concurrente. La vida real es de por sí paralela, por ejemplo los individuos de diferentes especies se desarrollan simultáneamente sin comunicarse o haciéndolo esporádicamente. Por otro lado, muchos operadores genéticos son fácilmente paralelizables.

Diversos algoritmos específicos se han desarrollado para implementaciones de algoritmos evolutivos [9]. Algunos utilizan las plataformas paralelas con el fin de realizar nuevos mecanismos de exploración del espacio de búsqueda, otros sin embargo sólo intentan reducir el tiempo de ejecución. En general, estos algoritmos pueden entrar dentro de esta pequeña clasificación:

- **Modelo de ejecuciones independientes.** Este modelo consiste en ejecutar un mismo algoritmo en diferentes procesadores. En general se utilizan situaciones

iniciales diferentes para ver cómo se comporta dicho algoritmo. Esto resulta útil para sacar estadísticas que pueden ser de gran ayuda [1].

- **Modelo maestro-esclavo.** Es un modelo bastante simple y fácil de visualizar, en donde se distribuye la evaluación de la función de fitness en varios procesadores, mientras que el bucle principal del algoritmo evolutivo se realiza en un único procesador. A diferencia de otros modelos, la utilización de la arquitectura paralela es utilizada únicamente para reducir el tiempo y no para explorar nuevas soluciones [9].

- **Modelo migratorio.** También conocido como *modelo distribuido* o *modelo de islas*, este modelo divide a la población en especies o islas que se reproducen de forma independiente en diferentes procesadores. Eventualmente, individuos de una subpoblación pueden migrar hacia otra [7] [17].

- **Modelo de vecindario.** También llamado *modelo celular*, consiste en distribuir a los individuos en una grilla donde cada procesador sólo posee algunos pocos de ellos. Los operadores genéticos sólo pueden aplicarse entre individuos cercanos geográficamente dentro de la grilla [17].

Existen también diversas estrategias híbridas o que resultan difíciles de clasificar.

6.2. Solución propuesta

La paralelización de la estrategia propuesta en el capítulo 4 tiene como principal objetivo reducir el tiempo de cómputo más que diversificar las búsquedas. La razón de esta paralelización está motivada por el gran tiempo de cómputo que requieren las evaluaciones de los individuos, haciendo que todo el proceso se torne lento.

Para entender la paralelización de la estrategia es necesario primero analizar el modo de funcionamiento del algoritmo. Luego se detalla el modelo paralelo utilizado.

6.2.1. Análisis del proceso evolutivo

La estrategia evolutiva presenta diversas etapas que pueden ser paralelizadas eficientemente. Por ejemplo, las diferentes especies se reproducen independientemente durante varias generaciones; los operadores de mutación pueden ser aplicados en forma paralela a todos los individuos de la población; la evaluación de aptitud de cada individuo es independiente de los demás. Casi todas estas situaciones provienen de la natutraleza paralela en la que se basan los algoritmos evolutivos. Sin embargo aquí se presta mayor atención a la última situación: la evaluación de los individuos.

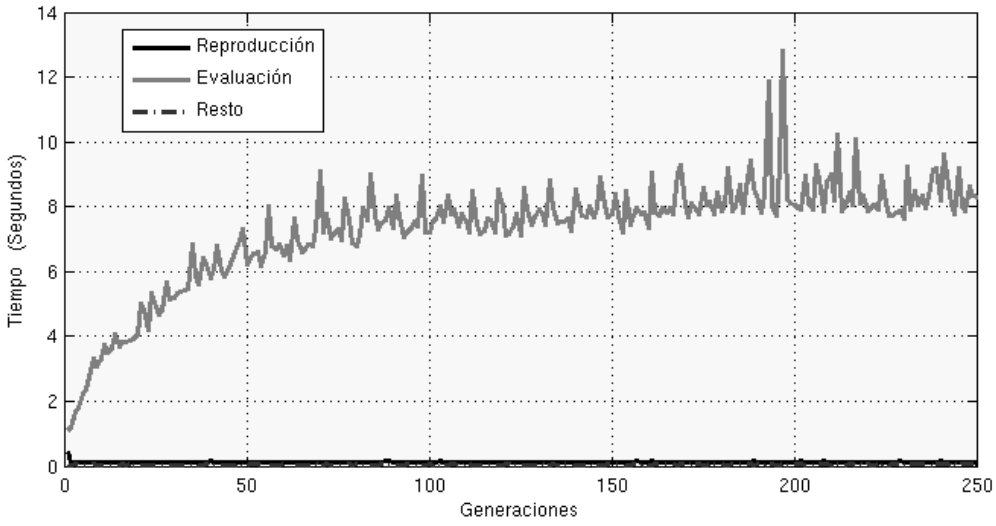

Figura 6.1: Gráfico del tiempo de las diferentes etapas del proceso evolutivo. Promedio de 50 ejecuciones independientes.

Como se vio en el capítulo 4, la estrategia presenta diversas etapas, en las que se pueden distinguir la reproducción (incluyendo el torneo y los operadores de mutación) y la evaluación, como las más costosas computacionalmente hablando.

Para evaluar qué tan costosas son estas etapas se midió el tiempo de las mismas en 50 ejecuciones independientes durante 250 generaciones del algoritmo. Para esto se tomó el tiempo de 3 etapas diferentes: el proceso de reproducción, la evaluación de los individuos y el resto de etapas entre las que se incluye: la división de las especies, el envejecimiento de la población, la técnica para evitar superpoblación, etc. Estas últimas etapas se miden todas juntas ya que son los módulos más rápidos del algoritmo. La figura6.1 muestra un gráfico de los tiempos medidos. A simple vista se aprecia la gran diferencia que existe entre el tiempo consumido por la evaluación de los individuos y todas las demás etapas. Mientras que el resto de las etapas consume mucho menos de 1 segundo, la evaluación de los individuos oscila entre 7 y 10 en casi todas las generaciones. Al comenzar el proceso evolutivo (hasta la generación 50) esta diferencia es sutilmente menor. Esto se debe principalmente a que los individuos no están bien adaptados, de modo que al probarlos en el simulador la mayoría quedan atascados contra alguna pared y el proceso de evaluación termina relativamente rápido.

Si bien podrían paralelizarse algunas partes del resto del algoritmo, la ganancia de tiempo que se conseguiría sería ínfima en comparación con el tiempo consumido por la etapa de evaluación. Esta gran diferencia de tiempo motiva a realizar una implementación en paralelo de la misma.

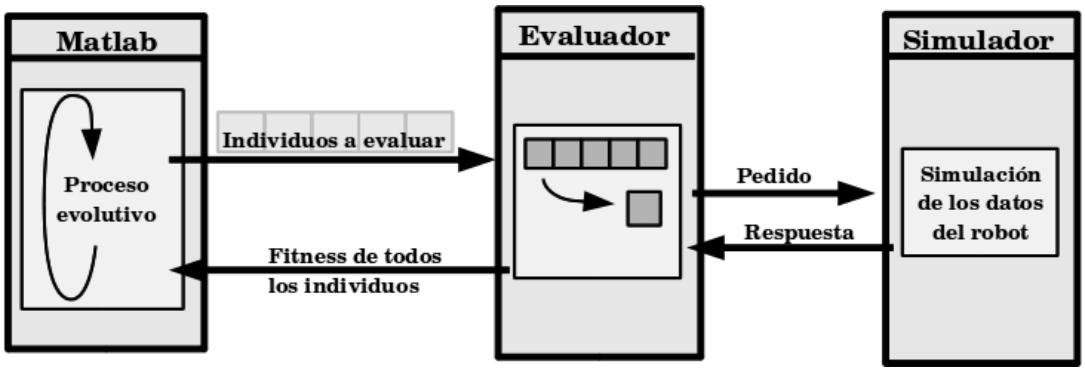

Figura 6.2: Esquema de la aplicación secuencial.

6.2.2. Comunicación con el simulador

La aplicación que realiza el proceso evolutivo está compuesta por tres partes: un proceso escrito en Matlab que posee el bucle principal del programa, el simulador que permite evaluar un individuo y un intérprete (o evaluador) escrito en C que hace de interface entre ambos procesos. Cuando la aplicación Matlab llega al punto de evaluación de los individuos envía los mismos al evaluador, que se encarga de realizar la evaluación de todos los individuos, uno por uno. Esta evaluación la realiza enviando pedidos al simulador y recibiendo los datos apropiados (por ejemplo se pide el valor de los sensores o se envía el valor de los motores). Esta comunicación entre los tres procesos queda reflejada en la figura 6.2. Para llevar a cabo la comunicación interproceso se utilizaron tuberías *fifos* propias del sistema *Linux*.

El hecho de usar el proceso evaluador no sólo permite una transparencia para la aplicación Matlab con respecto a la comunicación con el simulador, sino que además este proceso puede usarse para la comunicación con el robot real. Esto implica que la aplicación Matlab no distingue si se está comunicando con el simulador o el robot.

Este modelo de comunicación resulta muy útil para diseñar una solución paralela. Como el objetivo es paralelizar la etapa de evaluación, sólo debe enfocarse en el proceso evaluador.

6.2.3. Sobre la evaluación de los individuos

Algo a considerar al momento de diseñar una solución paralela es cómo se comportan los diferentes individuos a ser evaluados. En este caso, más que cómo se comportan interesa saber cuánto tarda el proceso de evaluar cada individuo. Si bien la función de fitness define un máximo de pasos a evaluar, cabe recordar que un individuo puede dejar de ser evaluado por diferentes razones. Es decir que, de forma general, cada

individuo se evalúa una cantidad de pasos diferentes.

No obstante estas diferencias de tiempos la evaluación es un proceso relativamente lento comparado con los tiempos de otras operaciones. Estos eventos resultan vitales al momento de llevar a cabo el diseño paralelo, ya que no va a ser eficiente realizar una distribución de los individuos en forma equitativa entre los diferentes procesos.

6.2.4. Modelo paralelo

En primer lugar es necesario definir sobre qué arquitectura trabajar. Al querer evaluar los individuos de la población en forma paralela es necesario ejecutar diversas instancias del simulador, ya que un simulador no puede atender simultáneamente varios pedidos. De este modo, cada nuevo proceso que evalúe individuos no correrá solo, sino junto a una instancia del simulador. Estas instancias requieren, además de procesador, recursos de memoria para diversas estructuras que maneja. A su vez, como la evaluación de un individuo es una operación muy costosa y el algoritmo paralelo no debe tener demasiada comunicación (ya que los individuos no dependen entre sí) se puede decir que, a grandes rasgos, el tiempo de cómputo en un procesador será mucho mayor que el tiempo de las comunicaciones, siempre dependiendo del modelo elegido.

Por estas razones se concluye que es conveniente utilizar una arquitectura distribuida, donde cada par evaluador-simulador puede ejecutarse en un terminal independiente con recursos propios. La figura 6.3 muestra el esquema utilizado, el cual se sigue desarrollando a continuación.

Como ya se dijo antes, realizar una distribución equitativa de los individuos resultaría perjudicial. Por esta razón la distribución de los mismos se hace a través de pedidos. Se utiliza un modelo *Manager-Workers* con el fin de que un proceso administre los datos recibidos desde Matlab y los pedidos de procesamiento, mientras los demás realicen las simulaciones. Para llevar esto a cabo se utilizó la librería MPI para realizar la comunicación en el *evaluador* paralelo escrito en lenguaje C, así como la implementación del algoritmo en la arquitectura distribuida.

El proceso *manager* recibe un vector con todos los individuos a evaluar por cada generación; luego, queda a la espera de pedidos. Cada *worker* pide trabajo cuando está libre. El *manager* atiende el primer pedido que llegue y responde con cierta cantidad de individuos para que ese *worker* procese. Cada vez que un *worker* termina de procesar sus datos (sus individuos) responde al *manager* con un vector de fitness de los individuos que evaluó. Al terminar la evaluación de todos los individuos de una generación, el *manager* retorna a la aplicación Matlab un vector con el fitness de todos los individuos. De este modo, para esta aplicación resulta transparente si se está comunicando con un entorno secuencial o uno paralelo.

Uno de los aspectos importantes a considerar en este punto es la cantidad de

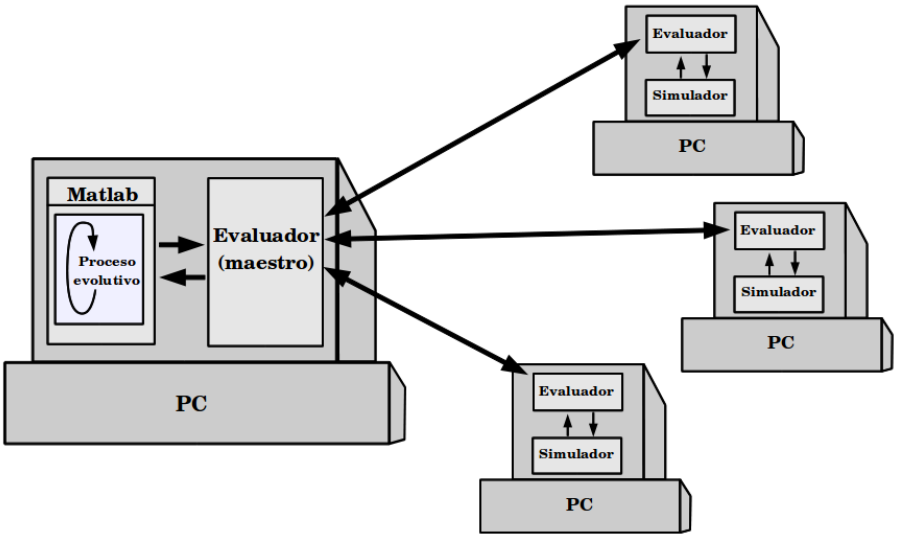

Figura 6.3: Esquema del modelo paralelo utilizado.

individuos que se envían en cada pedido, lo que se pasará a llamar *tamaño de bloque de envío*. Si bien se puede realizar un análisis teórico, este valor se obtuvo por medio de pruebas empíricas, lo cual se analiza en la próxima sección.

6.3. Experimentos realizados

Se realizaron diversas pruebas con el fin de obtener un análisis de rendimiento del algoritmo desarrollado. Para esto se utilizó un cluster de 12 máquinas interconectadas por un switch con las siguientes especificaciones: procesador AMD sempron LE-1200 2.1Ghz, memoria RAM DDR2 de 1Gb, sistema operativo Fedora 8 y *openmpi* para la implementación de la librería MPI.

Todas las pruebas fueron hechas con la configuración del proceso evolutivo descrita en el capítulo 4. Es decir que no se evaluó la estrategia con diferentes tamaños de población, tamaños de especies, o cantidad de generaciones, ya que el objetivo era reducir el tiempo de cómputo de esa estrategia.

Para lograr conclusiones contundentes se realizaron 10 pruebas independientes para cada uno de los casos descritos más abajo. En el caso de las pruebas paralelas se promediaron los datos obtenidos en las 10 pruebas. En el caso de las pruebas secuenciales se retuvo sólo el menor tiempo obtenido.

Estados iniciales

El algoritmo evolutivo presenta un gran problema a la hora de comparar la implementación secuencial con el modelo paralelo: el proceso es esencialmente estocástico. Cada ejecución del proceso evolutivo genera una población inicial aleatoria y paso a paso el proceso utiliza la aleatoriedad como su eje de funcionamiento. Por lo tanto dos ejecuciones independientes nunca son iguales y por esto no consumen el mismo tiempo. Para comparar el algoritmo secuencial con el paralelo es necesario utilizar programas que sean idénticos.

Si bien el proceso es estocástico existe una forma de que no lo sea. Las funciones *random* de los lenguajes de programación utilizan estados iniciales con los cuales se crea una secuencia determinista de números. Cada vez que un programa comienza el estado inicial cambia utilizando el reloj local y el último uso entre otros valores posibles. En este caso se utilizó la documentación de Matlab ([52]) que indica cómo establecer un estado inicial para que la secuencia de llamadas a instrucciones de tipo *random* sean iguales en diferentes programas. Este estado no es más que un simple número entero. Por lo tanto, al establecer un estado inicial se determina una ejecución particular de modo que todas las ejecuciones con ese estado son iguales. Sin embargo, si se elige un estado "malo", podría no ser representativo del normal funcionamiento del programa. Para evitar esto se realizaron pruebas con 5 estados iniciales diferentes para cada una de las configuraciones evaluadas, de forma que provean variedad en los resultados obtenidos.

Tamaño de bloque a usar

Para evaluar qué *tamaño de bloque de envío* es mejor, se realizaron pruebas con 5 tamaños diferentes: 1, 2, 4, 8 y 16. Es decir que cada vez que un proceso *worker* pida trabajo, el *manager* enviará dichas cantidades de individuos para que sean procesados. Al aumentar el número de procesadores las comunicaciones entre estos aumentan y por lo tanto el *manager* debe atender más pedidos.

Si las comunicaciones exceden el trabajo que realizan los *workers*, el *manager* puede convertirse en un cuello de botella, generando un gran overhead al sistema. Por esta razón, se evaluó de forma independiente los tamaños de bloque óptimos para cada cantidad de procesadores utilizados.

Comportamiento con distintas cantidades de procesadores

Es esperable que el algoritmo se comporte en forma diferente con distintas cantidades de procesadores. Para esto se utilizaron cuatro configuraciones diferentes: 4, 6, 8 y 10 unidades de procesamiento. Cabe destacar que en cada caso, una unidad contiene a la aplicación Matlab junto con el proceso *manager* mientras que las otras contienen

los *workers* con sus respectivos simuladores. En la siguiente sección se analizan los resultados de dichos experimentos, incluyendo un estudio del speedup y eficiencia para estas cantidades de procesadores.

6.4. Resultados

Los resultados obtenidos se dividen en dos etapas. En primer lugar se analiza el tamaño de bloque de envío óptimo. Luego se analiza el comportamiento del algoritmo con diferentes cantidades de procesadores.

6.4.1. Definición de tamaño de bloque óptimo

Como ya se dijo antes, el cálculo de tamaño de bloque óptimo se realiza para cada cantidad de procesadores utilizados. De este modo se intenta establecer una relación entre cantidad de procesadores y tamaño de bloque a usar, para luego realizar las pruebas de rendimiento.

La tabla 6.1 muestra los tiempos obtenidos con diferentes tamaños de bloque para cada uno de los 5 estados (nombrados de la *A* a la *E*) y para cada cantidad de procesadores utilizados. La figura 6.4 resume los datos obtenidos para el estado *A* (el resto de los estados se comporta de forma muy similar). Como se aprecia, los resultados con los diferentes tamaños de bloque fueron similares en todas las configuraciones. El tamaño de bloque 1 logró el menor tiempo en todos los casos, incrementándolo gradualmente hasta llegar al tamaño 16, que resultó ser el peor.

Los datos concluyen que el tamaño de bloque óptimo en todos los casos es 1. Esto se debe a que al pedir pocos individuos (en este caso uno) cada procesador no debe esperar mucho tiempo a que los demás terminen, en caso de que ya no haya trabajo; sin embargo podrían saturar al *manager* de trabajo haciendo que este no pueda atender todos los pedidos a tiempo. Al pedir varios individuos por vez, el *manager* atiende menos pedidos, pero existe mayor overhead en el momento que el trabajo se termina, ya que algún proceso seguramente tenga ciertos individuos encolados para evaluar. En este algoritmo el tiempo de evaluación es tan alto en comparación con el tiempo de las comunicaciones que pedir un individuo a la vez no trae mayores complicaciones al *manager*, incluso con 10 procesadores.

Cabe analizar qué pasaría si el número de *workers* aumenta. Si se aumentase de 10 a 50, 100, o quizá 200 procesos *workers* es posible que el *manager* comience a saturarse atendiendo pedidos y el tamaño de bloque óptimo deba aumentarse. Con una arquitectura de este tamaño habría que realizar nuevas pruebas para analizar el comportamiento.

4 Procesadores					
	Estados				
Tam. bloque	A	B	C	D	E
1	298.28	297.22	322.54	284.58	289.60
2	300.88	301.18	325.80	288.94	294.52
4	307.28	307.80	332.34	295.60	299.60
8	322.28	321.14	348.12	306.64	313.66
16	342.06	345.18	375.10	330.60	338.62
6 Procesadores					
	Estados				
Tam. bloque	A	B	C	D	E
1	212.80	211.88	226.42	206.28	208.36
2	216.78	215.92	230.90	212.22	212.06
4	227.38	227.08	241.44	220.78	222.64
8	252.94	247.96	266.96	240.40	241.56
16	288.22	288.16	309.78	278.50	281.22
8 Procesadores					
	Estados				
Tam. bloque	A	B	C	D	E
1	179.70	177.74	190.64	175.38	176.52
2	185.22	184.34	196.26	183.30	184.86
4	195.56	197.46	207.10	192.54	195.40
8	211.50	215.98	221.74	206.10	211.24
16	290.12	292.28	312.32	278.84	285.16
10 Procesadores					
	Estados				
Tam. bloque	A	B	C	D	E
1	152.54	149.84	157.36	146.58	146.94
2	159.06	157.88	164.46	154.68	155.96
4	174.46	171.72	176.92	165.06	168.10
8	198.94	197.78	203.02	185.84	194.20
16	287.66	286.90	303.58	271.68	281.22

Tabla 6.1: Tiempos obtenidos para el cálculo de tamaño de bloque óptimo.

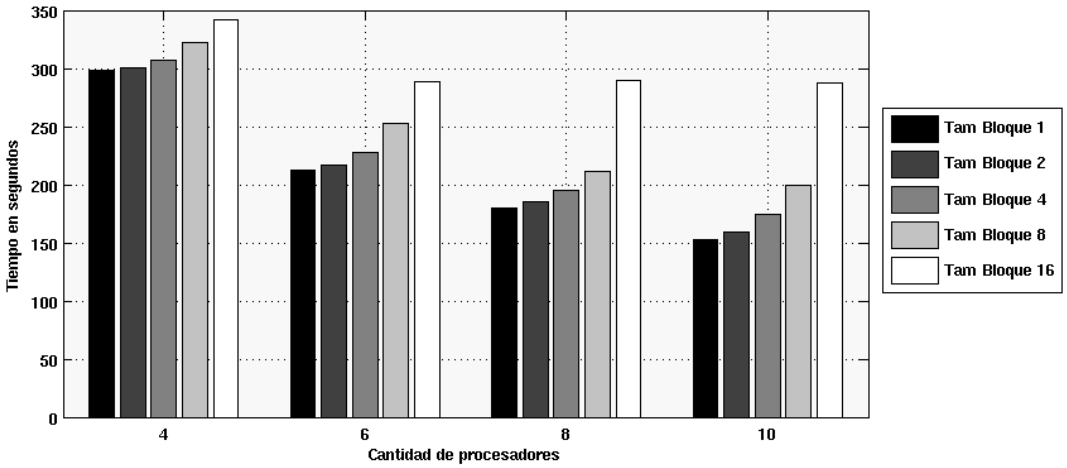

Figura 6.4: Gráfico de los tiempos para el cálculo de tamaño de bloque óptimo.

	Estados				
Cant. CPU	A	B	C	D	E
secuencial	824.22	820.62	919.62	779.46	790.68
4	298.28	297.22	322.54	284.58	289.60
6	212.80	211.88	226.42	206.28	208.36
8	179.70	177.74	190.64	175.38	176.52
10	152.54	149.84	157.36	146.58	146.94

Tabla 6.2: Tiempos obtenidos para diferentes cantidades de procesadores.

6.4.2. Análisis de rendimiento

En esta sección se analiza el rendimiento del algoritmo para la estrategia evolutiva propuesta, con el tamaño de bloque ya definido en 1.

Tiempos

La tabla 6.2 muestra los tiempos obtenidos tanto para el algoritmo secuencial como para las diferentes configuraciones de la arquitectura paralela, para cada uno de los cinco estados iniciales del proceso evolutivo. Aquí puede verse claramente como el algoritmo reduce el tiempo total consumido considerablemente a medida que se aumenta el número de CPU usadas. La figura 6.5 ilustra en un gráfico dichos resultados. En este gráfico se aprecia que los diferentes estados iniciales sólo cambian de forma sutil los tiempos finales.

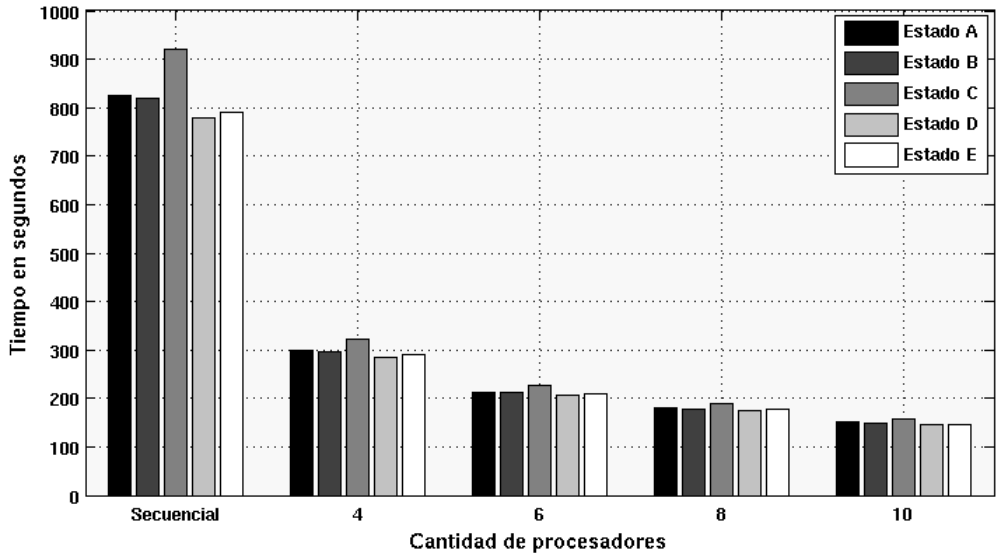

Figura 6.5: Gráfico de los tiempos para diferentes cantidades de procesadores.

	Estados				
Speedup	A	B	C	D	E
4	2.7632	2.7610	2.8512	2.7390	2.7302
6	3.8732	3.8730	4.0616	3.7787	3.7948
8	4.5866	4.6170	4.8239	4.4444	4.4793
10	5.4033	5.4766	5.8441	5.3176	5.3810

Tabla 6.3: Speedup logrado para diferentes cantidades de procesadores.

Speedup

En la tabla 6.3 se aprecia el speedup obtenido para las diferentes configuraciones de procesadores. La figura 6.6 refleja estos valores en un gráfico. Como ya se ha dicho, todos los estados iniciales se comportan de forma similar, con lo cual no se hará mayor mención a ello. El speedup logrado es sublineal (como en la mayoría de los problemas reales) pero con una curva satisfactoria, ya que parece seguir creciendo en forma lineal a medida que el número de procesadores aumenta.

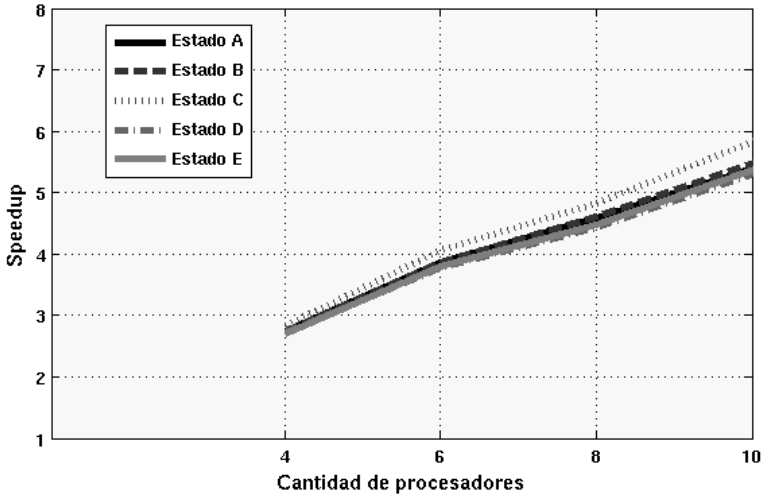

Figura 6.6: Gráfico del speedup logrado con diferentes cantidades de procesadores.

	Estados				
Eficiencia	A	B	C	D	E
4	0.6908	0.6902	0.7128	0.6847	0.6826
6	0.6455	0.6455	0.6769	0.6298	0.6325
8	0.5733	0.5771	0.6030	0.5556	0.5599
10	0.5403	0.5477	0.5844	0.5318	0.5381

Tabla 6.4: Eficiencia lograda para diferentes cantidades de procesadores.

Eficiencia

La eficiencia lograda se muestra en la tabla 6.4 y en la figura 6.7 mediante un gráfico. Los resultados obtenidos son satisfactorios, aunque como es de esperar decaen a medida que la cantidad de procesadores utilizados aumenta. Los valores varían entre 0.5 y 0.7, dependiendo la cantidad de procesadores. Si bien no son los mejores valores posibles cabe destacar que los tiempos de las diferentes etapas del algoritmo secuencial comienzan a tener relevancia al reducir el tiempo de ejecución total a estas magnitudes. Los diferentes procesadores usados para realizar las evaluaciones de forma paralela deben esperar a las demás etapas del proceso evolutivo (parte secuencial del algoritmo) y esta espera se vuelve cada vez más significativa. Siendo así, un valor de eficiencia de 0.55 utilizando 10 procesadores supone un buen uso de los mismos.

Si se utilizase un número mucho más grande de procesadores la eficiencia seguiría cayendo hasta llegar a un límite calculable mediante la ley de Amdahl. En tal caso sería

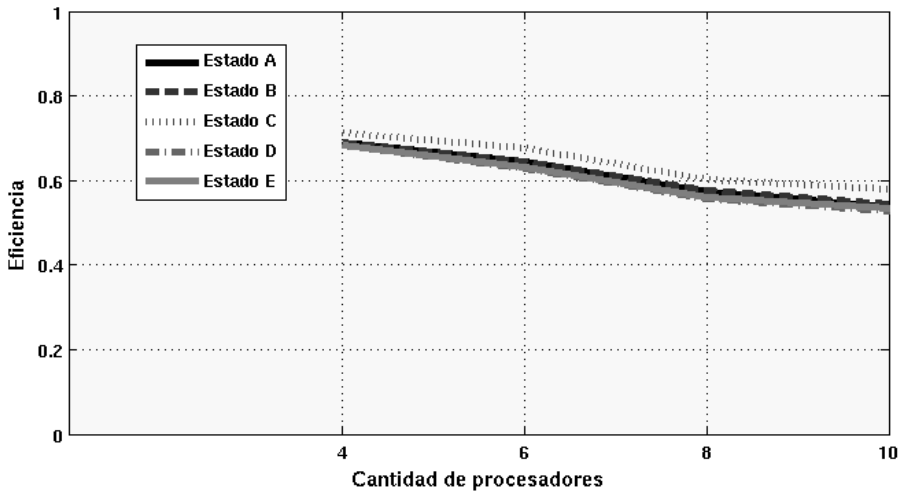

Figura 6.7: Gráfico de la eficiencia conseguida con diferentes cantidades de procesadores.

bueno centrarse en la parte secuencial del algoritmo, es decir paralelizar también las demás etapas del proceso evolutivo. Como ya se dijo antes, existen etapas que pueden ser procesadas en paralelo, de forma tal que el porcentaje de código secuencial podría ser reducido y así lograr una eficiencia mayor incluso con un número muy grande de procesadores.

6.5. Conclusiones

En este capítulo se desarrolló una implementación paralela a la estrategia evolutiva propuesta en el capítulo 4. Se analizó profundamente el algoritmo así como sus modos de comunicación, con el fin de reducir el overhead que pueda generarse.

La decisión del tamaño de bloque a utilizar mostró ser crucial. Los resultados obtenidos dejan ver que una mala elección del tamaño de bloque podría generar tiempos de hasta el doble de lo consumido con un tamaño de bloque óptimo. Por estas mismas razones se explicó también la importancia de realizar nuevas pruebas en caso de utilizar grandes cantidades de procesadores.

Los tiempos obtenidos resultaron satisfactorios considerando que el objetivo de la estrategia era conseguir una reducción de estos. La eficiencia lograda mostró buenos resultados sobre todo utilizando pocos procesadores. Se analizó también la posibilidad de paralelizar el actual segmento secuencial de código con el fin de utilizar un mayor número de procesadores y al mismo tiempo seguir obteniendo una buena eficiencia.

Capítulo 7

Conclusiones y trabajos futuros

En este trabajo se implementó una estrategia evolutiva capaz de obtener un controlador neuronal para un robot Khepera II que permite esquivar obstáculos y alcanzar objetivos. Así mismo, se desarrolló un algoritmo paralelo para reducir el tiempo de la estrategia.

7.1. Conclusiones

Las redes neuronales artificiales muestran ser un modelo eficiente en la resolución de este tipo de problemas aunque queda mucho por resolver con respecto a tipos de arquitecturas a usar en diferentes casos, sobre todo en problemas dinámicos como el de este trabajo, donde se requieren mecanismos que implementen sistemas de memoria. La red utilizada en este trabajo permite ser instalada fácilmente en un robot real ya que su pequeño tamaño hace que el cómputo necesario para evaluar un patrón sea mínimo. Sin embargo el poder computacional (el poder de aproximación a una curva específica) no es el mismo que el de una red con mayor cantidad de neuronas o capas.

Las metaheurísticas, y en este caso los algoritmos evolutivos resultan ser técnicas de aproximación eficaces y veloces a la hora de resolver problemas combinatorios con grandes cardinalidades y que llevarían un tiempo incontable con soluciones exactas. Mientras la complejidad de los problemas abordados aumenta día a día este tipo de soluciones se encamina a ser cada vez más utilizada.

La estrategia propuesta tiene como principal objetivo escapar de diversos óptimos locales generados por la función de aptitud. Los resultados obtenidos muestran que los componentes de la estrategia logran escapar de dichos óptimos de un modo mucho más eficiente que una estrategia convencional. La especiación implementa estrategias de búsqueda en diferentes sectores del espacio de soluciones, lo que permite generar mayor diversidad en la búsqueda. El hecho de preservarlas por medio de una cantidad

mínima de individuos es un factor muy importante para no generar presión selectiva. Utilizar poblaciones de tamaño variable agrega complejidad para controlar la población pero también mayor posibilidad a cada individuo de dejar descendientes. Así mismo, el operador de mutación de comportamiento simétrico es un agregado importante que permite variar a un individuo no sólo en base a su genotipo sino con un propósito particular relativo a su comportamiento.

La paralelización de la estrategia resultó muy provechosa, ya que se pudieron obtener controladores en apenas unos minutos. Si bien la eficiencia lograda no fue la mejor, se logró una buena eficiencia al detectar el punto más lento de toda la estrategia y centrar la paralelización en ello. El algoritmo paralelo puede utilizarse eficientemente para procesos evolutivos de poblaciones mayores o problemas más grandes.

7.2. Trabajos futuros

El trabajo presentado aquí deja un gran número de posibles mejoras a estudiar de las cuales aquí se nombran las principales.

En primer lugar, el hecho de utilizar una red neuronal de tamaño mínimo posibilita el uso de esta como módulo de una arquitectura más compleja que permite resolver otro tipo de problemas. De este modo se podrían conseguir controladores para ciertas tareas y luego agruparlos en un gran controlador para que el robot realice todas estas tareas de forma armónica.

Un agregado importante que puede pensarse es la posibilidad de que los individuos migren de especie si su cromosoma se acerca demasiado al de un individuo de otra. Del mismo modo, podría pensarse otro criterio de división de especies que no sea simplemente un gran número de individuos, sino una verificación respecto a los diferentes comportamientos, o distancias euclídeas en los cromosomas.

Con respecto a la estrategia paralela queda pendiente probar con un gran número de máquinas para observar cómo se comporta el algoritmo y qué eficiencia se logra. Por un lado, se debe recalcular los tamaños de bloque para dichas cantidades de procesadores. Por otro, ya que posiblemente la eficiencia comience a decaer, cabe la posibilidad de comenzar a paralelizar diferentes partes del proceso evolutivo para así conseguir una gran reducción del tiempo de ejecución. En este caso se estaría entrando en modelo de algoritmo diferente, donde no se podría hablar sólo de *manager-workers*, sino que habría un esquema de comunicación diferente. Una posibilidad es procesar cada especie en diferentes procesadores, ya que estas no necesitan comunicarse salvo cada un número fijo de generaciones.

Apéndices

Apéndice A

Khepera II

El Khepera II es un robot ampliamente utilizado en el ambiente de investigación y es considerado un estándar en el área. De hecho, fue diseñado con fines de enseñanza e investigación en el *Swiss Research Priority Program* en el año 1992. El Khepera II es el sucesor del Khepera, siendo totalmente compatible uno con otro, pero agregando el primero nuevas capacidades. El robot permite abordar un gran conjunto de problemáticas, entre las que se pueden nombrar la evasión de obstáculos, ejecución de trayectorias, preprocesamiento de la información de los sensores, etc.

A.1. Características principales

El robot posee una forma cilíndrica como puede verse en la figura A.1. Tiene un tamaño pequeño con unas dimensiones de 70mm de diámetro y 30mm de altura, y un peso de apenas 80g. La figura también muestra una vista superior donde se aprecian los 8 sensores y los dos motores que posee el robot. La figura A.2 muestra un diagrama más completo de los diferentes componentes externos que posee el robot [38].

Sensores

El robot posee 8 sensores infrarrojos que pueden ser utilizados de dos formas diferentes: como sensores de proximidad o como sensores de luminosidad. Para esto el robot posee comandos específicos que retornan el valor de cada sensores. En ambos casos los valores retornados se encuentran entre 0 y 1023 [38].

Al utilizar los sensores para detectar obstáculos, los mismos retornan un valor más alto cuanto más cerca se esté de un objeto. El rango de detección se encuentra entre 0cm y 10cm aunque en la práctica estos valores pueden cambiar considerablemente,

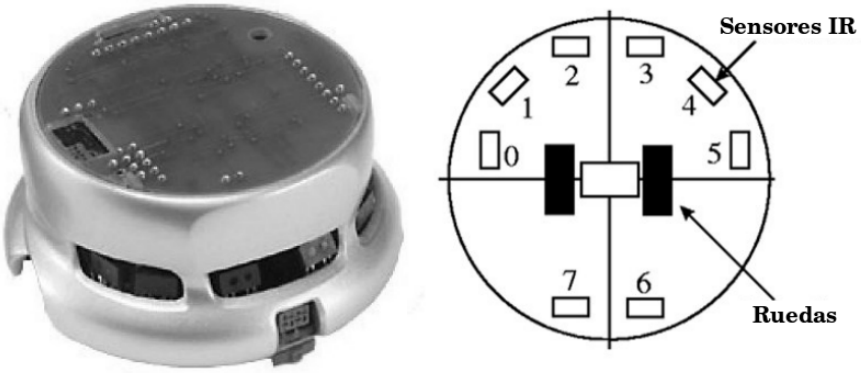

Figura A.1: Robot Khepera II.

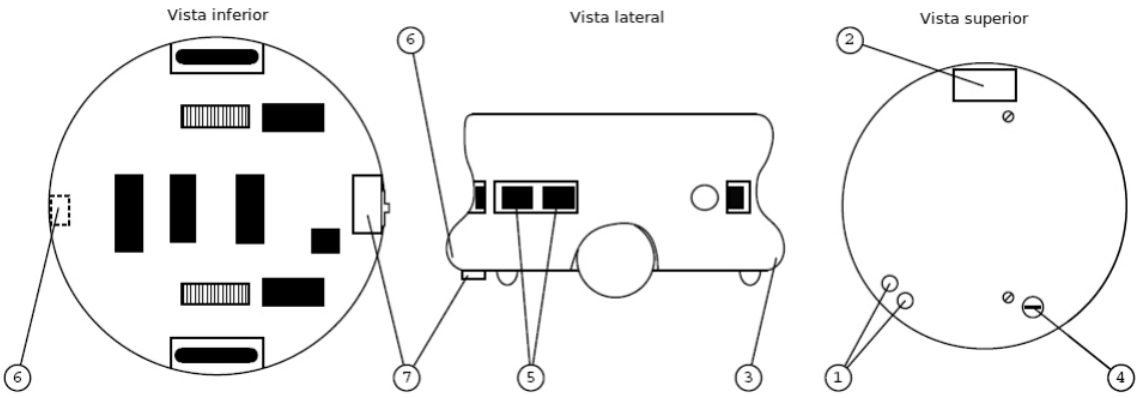

Figura A.2: Esquema externo de un robot Khepera II.

dependiendo en gran medida de la superficie del objeto y su capacidad para reflejar rayos infrarojos, así como también de la luz ambiental.

Al utilizar los sensores como detectores de luz, los mismos retornan valores más bajos cuanto mayor sea la intensidad lumínica detectada. El rango de detección depende en gran medida de la intensidad de la fuente generadora. Es decir que una fuente de mayor poder puede ser detectada desde una distancia mayor.

En adición a estos sensores infrarrojos, existen una serie de accesorios que pueden instalarse en el robot, como una cámara o una pinza mecánica, entre otros. Cada uno de estos accesorios son llamados torretas y pueden apilarse unos encima de los otros en el cuerpo del robot.

Hardware

El robot posee un hardware interno que controla ciertas funcionalidades así como un procesador que permite ejecutar aplicaciones previamente introducidas en el mismo. El procesador es un Motorola 68331 25Mhz, con 512 Kb de RAM. Posee además 512Kb de memoria FLASH (memoria no volátil para instalar aplicaciones) y una interface serie RS232 para conectar el robot a una computadora.

Motores

El Khepera II posee dos motores de corriente continua con una caja de reducción de 25:1. Cada motor controla de forma independiente las dos ruedas del robot. En el eje del motor está situado un codificador incremental que otorga una resolución de 600 pulsos por revolución de una rueda, lo que equivale a 12 pulsos por milímetro que se desplaza el robot. El procesador tiene conexión directa con cada uno de los motores y puede leer la información del codificador incremental.

Una rutina del procesador controla los pulsos de los motores y establece las velocidades de los mismos. Los motores pueden recibir valores entre -128 y 127, donde los negativos hacen que el motor gire hacia atrás, los valores positivos hacen que el motor avance hacia adelante, y el valor 0 detiene el motor. Cada unidad representa una velocidad de 8mm por segundo de recorrido de las ruedas del robot [38].

A.2. Modos de programación

Existen diferentes formas de programar un robot Khepera II. Uno de los modos más comunes es a través de una interface gráfica propuesta por LabVIEW. Diversas empresas

han creado interfaces que pueden utilizarse para comunicarse con el robot entre las que se pueden nombrar *MathWorks*, *Calerga* o *Cyberbotics*.

El robot funciona a través de envío de comandos, donde estos pueden estar cargados en un programa dentro del robot, o pueden ser enviados a través del cable serie. Aquí se distinguen dos modos de programar al robot. En el primer caso se escribe un programa en lenguaje C limitado a ciertas librerías que posee el compilador cruzado, el cual permite crear programas para el robot. En el segundo caso, el robot es usado en modo esclavo. La aplicación, o controlador, está situado en un lugar externo al robot (generalmente una PC estándar) y se envían comandos en base a decisiones de este programa. Si bien el modo esclavo no es el mejor visto en el área de robótica posee la ventaja de poder utilizar un procesador más potente que el propio del robot.

Existe una API detallada para poder interactuar con el robot ya sea en languaje C o en lenguaje ensamblador para M68000 [38].

Apéndice B

Khepera Simulator version 2.0

El simulador utilizado para el desarrollo de esta tesina es el creado por Oliver Michel en la University of Nice-Sophia Antipolis de Francia, en el año 1996. El simulador está completamente escrito en lenguaje C, es de software libre y puede ser descargado en forma gratuita desde la web.

B.1. Descripción del simulador

Al ejecutar el simulador aparece una aplicación con una vista como la figura B.1. Aquí se distinguen tres partes principales: una vista del mundo donde se mueve el robot, una vista de los valores de los sensores del robot junto con los de sus motores, y un cuadro con información de control que el usuario puede manipular con programación específica.

El simulador provee diferentes escenarios precreados que pueden ser utilizados, aunque también provee la funcionalidad de modificarlos o crear nuevos. Estos escenarios constan principalmente de paredes (obstáculos) que el robot debe esquivar, aunque también es posible colocar lámparas para utilizar con los sensores lumínicos del robot.

El simulador permite efectuar los dos modos de funcionamiento del robot Khepera. En primer lugar es posible crear un programa que corra dentro del robot (en este caso dentro del simulador) sin comunicación posible con el exterior. Por otro lado también es posible utilizar el simulador en modo envío de comandos. Este último modo fue el utilizado para este trabajo y el que se detalla a continuación.

Esquema de funcionamiento del envío de comandos

El simulador está formado por diversos módulos agrupados en varios archivos con funcionalidades comunes. El archivo principal es *sim.c*. Aquí se encuentra el código que

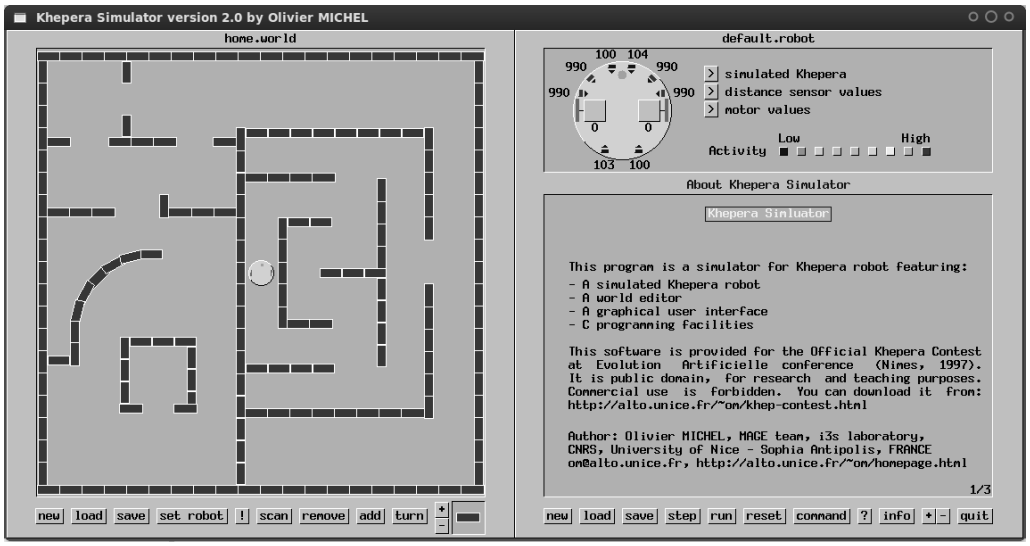

Figura B.1: Khepera Simulator version 2.

abre un nuevo programa y carga algunas configuraciones iniciales, así como el bucle principal que se ejecuta infinitamente (o hasta que muera el proceso). Al ejecutar el simulador con el parámetro *-p* se inicia en modo de envío de comandos (*piped*) como se explica a continuación.

En primer lugar se crean dos tuberías (*pipes*) del sistema operativo para poder comunicarse con un programa externo. Una tubería funciona como entrada de comandos y la otra como salida. Dentro del bucle principal del simulador, cuando se encuentra que se está en modo piped el flujo de ejecución entra en un nuevo bucle donde constantemente se esperan nuevos comandos y se actúa en consecuencia. Para entender este nuevo bucle se debe analizar el archivo *robot.c* que contiene todas las funciones propias de la simulación del robot. Dentro de este archivo, la función *PipedRobotRun()* controla el arribo de nuevos mensajes (comandos) y la respuesta de los mismos. Existen sólo algunos pocos mensajes implementados de los tantos que trae el robot Khepera, que no se nombran aquí. Dentro de los comandos implementados se pueden destacar el *N*, que permite leer los sensores de proximidad, y el *D* que permite enviar valores a los motores. Estos dos mensajes podrían bastar para realizar un controlador que evade obstáculos. Siempre que se recibe un mensaje, el simulador contesta con el mismo nombre del comando pero en minúscula, del mismo modo que lo hace el robot real.

B.2. Modificaciones realizadas

Para poder llevar a cabo el trabajo presentado en esta tesina de grado tuvieron que realizarse diversas modificaciones o ampliaciones al simulador que permitan ciertas funcionalidades necesarias. A continuación se listan los agregados y modificaciones más importantes.

Nuevos comandos

Uno de los principales agregados llevados a cabo es la incorporación de nuevos comandos necesarios para realizar el proceso evolutivo.

- **X,x,y,a**. Coloca al robot en la posición *(x,y)* del terreno y con un ángulo de orientación dado por *a*. Esta funcionalidad resulta muy necesaria para comenzar cada evaluación, ya que todos los individuos deben comenzar desde el mismo punto de partida.

- **Y**. Pide al robot que indique en qué posición se encuentra. El robot responde con un mensaje indicando las coordenadas *(x,y)* junto con su ángulo *a*. Esta funcionalidad fue muy necesaria para saber la ubicación del robot y conocer su distancia al objetivo en la resolución del problema de alcance de objetivos.

- **Z**. Testea si el robot se encuentra atascado. Retorna 1 en caso afirmativo, 0 en caso negativo. Si bien el simulador ya tenía una función que realizaba esta tarea, no poseía un comando para conocer esa información. El nuevo comando simplemente llama a la función propia del simulador.

Nuevo parámetro

Se creó un nuevo parámetro llamado *-n*. Al ejecutar el simulador con este parámetro se inhabilitan la mayoría de las funciones gráficas con el fin de reducir el uso del procesador. Es decir, el simulador funciona pero no puede verse el robot moviéndose. Esto ahorra considerablemente el tiempo consumido por el procesador, consiguiendo una reducción del tiempo total de simulación, y por ende del proceso evolutivo utilizado para conseguir el controlador.

Modificaciones en la estructura de funcionamiento

Se llevaron a cabo diversas modificaciones sutiles pero importantes a la estructura de funcionamiento del simulador.

En primer lugar, el bucle que espera comandos es un bucle *busy-waiting* que simplemente duerme mientras no haya un nuevo mensaje en cola, atiende el mensaje si llega, responde, y vuelve a dormir. Mientras el bucle duerme, el robot sigue moviéndose de acuerdo a los valores de sus motores. En el robot real esto no es un gran problema porque el robot se mueve extremadamente lento relativo a la velocidad de procesamiento del controlador. Sin embargo la simulación se hace de forma acelerada, es decir que si un mensaje no llega a tiempo por problemas del sistema operativo puede ocurrir que el robot colisione cuando el controlador no hubiese hecho esto en el robot real. Para solucionar esto el robot sólo se mueve un paso a la vez luego de procesar un mensaje. De este modo se anula el riesgo de la no llegada a tiempo de mensajes. Si bien el bucle sigue siendo *busy-wating* ahora la espera es totalmente ociosa, como si el tiempo se detuviese para el robot mientras no llegue un nuevo mensaje.

Se utilizaron diversas escalas para el movimiento del robot de acuerdo a si se quería una velocidad más rápida o más lenta de simulación. Entre estas escalas se utilizó una que simula la velocidad real del robot. Mientras que para realizar la simulación se utilizó una escala 200 veces más rápida.

El simulador utiliza funciones *random* que agregan ruido a las señales de entrada y a otros factores del robot. Para realizar las pruebas de rendimiento de la aplicación paralela hubo que quitar toda función aleatoria ya que era necesario realizar pruebas idénticas. Estos cambios fueron sólo temporales con el fin de llevar a cabo dichas pruebas.

Se modificó un archivo de configuración para que la aplicación comience directamente lista para recibir comandos. Originalmente el simulador espera la interacción del usuario desde un entorno gráfico para comenzar a utilizar los distintos modos. Así mismo se cambió el escenario inicial y otras opciones con que comienza la aplicación.

El simulador utiliza una función matemática para calcular los valores de los sensores del robot. Esta función es bastante ideal y no es lo que ocurre en el robot real. Se escalaron los valores de los sensores con el fin de aproximar mejor a la curva de los sensores reales.

Modificaciones varias

En adición, se llevaron a cabo diversas modificaciones que van más allá del alcance de este apéndice, como la reparación de diversos errores que tenía el simulador, o el agregado de pequeñas funciones que ayudan a proveer funcionalidad agregada.

Índice de figuras

Índice de tablas

Bibliografía

[1] E. Alba. *Parallel Metaheuristics: A New Class of Algorithms*. John Wiley and Sons, 2005.

[2] D. Maravall-Gómez Allende. *Reconocimiento de formas y Visión artificial*. Addison Wesley, 1994.

[3] G. R. Andrews. *Foundations of Multithreaded, Parallel, and Distributed Programming*. Addison-Wesley, 2000.

[4] M.J.F. Aparicio and T.C. Cano. *Aplicaciones de las redes de neuronas en supervisión, diagnosis y control de procesos*. Equinoccio, 1999.

[5] B.S. Araujo. *Aprendizaje automático: conceptos básicos y avanzados.* Pearson Prentice Hall, 2006.

[6] B. Barney. *Introduction to parallel computing*. Lawrence Livermore National Laboratory, 2010.

[7] T. Bäck. *Evolutionary algorithms in theory and practice: evolution strategies, evolutionary programming, genetic algorithms*. Oxford University Press, 1996.

[8] T. Bäck, D. Fogel, and Z. Michalewicz. *Handbook of Evolutionary Computation*. IOP Publishing and Oxford University Press, 1997.

[9] E. Cantù-Paz. *Efficient and accurate parallel genetic algorithms*. Genetic algorithms and evolutionary computation. Kluwer Academic Publishers, 2000.

[10] C. A. Coello Coello. *Introducción a la computación evolutiva (notas de curso)*. 2010. http://delta.cs.cinvestav.mx/~ccoello/compevol/apuntes.pdf.

[11] T. Crainicand and M. Toulouse. *Handbook of Metaheuristics*, chapter Parallel Strategies for Metaheuristics, pages 475–513. Kluwer Academic Publishers, 2003.

[12] G. Cybenko. *Approximation by superposition of a sigmoid function*. Mathematics of Control, Signals and Systems, 2. 1989.

[13] C. Darwin. *On the origin of species by means of natural selection or the preservation of favored races in the struggle for life*. Murray, 1864.

[14] H.B. Dong, J. He, H.-K. Huang, and W. Hou. A mixed mutation strategy evolutionary programming combined with species conservation technique. In *Lecture Notes in Computer Science*, volume 0302-9743, pages 593–602, 2005.

[15] J.J. Dongarra, I. Foster, G. Fox, W. Gropp, K. Kennedy, L. Torczon, and A. White. *Sourcebook of parallel computing*. The Morgan Kaufmann Series in Computer Architecture and Design Series. Morgan Kaufmann, 2003.

[16] M. Dorigo. *The Ant Colony Optimization Metaheuristic: Algorithms, Applications and Advances*. Technical Report IRIDIA-2000-32, 2000.

[17] M. Dorigo and V. Maniezzo. Parallel genetic algorithms: Introduction and overview of current research. In J. Stender, editor, *Parallel genetic algorithms: theory and applications*, pages 5–42. IOS Press, 1993.

[18] M. Dorigo and T. St
ützle. *Ant colony optimization*. Bradford Books. MIT Press, 2004.

[19] F. J. Q. Flor and A. G. Solo. *Computadores paralelos y evaluación de prestaciones*. Colección Ciencia y Técnica. Universidad de Castilla-La Mancha, 1996.

[20] R. Flórez López and J.M. Fernández Fernández. *Las Redes Neuronales Artificiales*. Serie MetodologÃa y análisis de datos en ciencias sociales. Netbiblo S.L., 2008.

[21] L. Fogel, J. Owens, and M. Walsh. *Artificial Intelligence Through Simulated Evolution*. 1966.

[22] M. R. Garey and D. S. Johnson. *Computers and intractability; a Guide to the theory of NP-completeness*. W. H. Freeman, 1979.

[23] F. Glover. Heuristics for integer programming using surrogate constraints. *Decision Sciences*, 8:156–166, 1977.

[24] F. Glover. Future paths for integer programming and links to artificial intelligence. *Computers and Operations Research*, 13:533–549, 1986.

[25] F. Glover, M. Laguna, and R. Martí. *Fundamentals of Scatter Search and Path Relinking*, pages 658–684. Control and Cybernetics, 29(3), 2000.

[26] D.E. Goldberg and K. Deb. A comparative analisys of selection schemes used in genetic algorithms. In *Foundations of Genetic Algotithms*, volume 1, pages 69–93, 1991.

[27] D.E. Goldberg and B. L Miller. Genetic algorithms, tournament selection, and the effects of noise. In *Complex Systems*, volume 9, pages 193–212, 1995.

[28] T. Gomi and A. Griffith. Evolutionary robotics-an overview. In *Evolutionary Computation*, pages 40–49, 1996. Proceedings of IEEE International Conference on 20-22 May 1996.

[29] A. Grama, Gupta A., Karypis G., and Kumar V. *Introduction to parallel computing*. Pearson Education. Addison-Wesley, 2003.

[30] P. Hansen and N. Mladenovic. Variable neighborhood search. *Computers Operation*, 24:1097–1100, 1997.

[31] P. Hansen and N. Mladenovic. Variable neighborhood search: Principles and aplications. *European Journal of Operational Research, 130*, pages 449–467, 2001.

[32] J. Holland. *Adaptation in Natural and Artificial Systems*. The MIT Press, 1975.

[33] K. Hornik, M. Stinchcombe, and H. White. Multilayer feedforward networks are universal approximators. *Neural Networks*, 2(5):359–366, 1989.

[34] P. Isasi Viñuela and I. Galván León. *Redes de neuronas artificiales: un enfoque práctico*. Pearson Educación, 2004.

[35] Jain. *Handbook of Pattern Recognition and Image Processing*, chapter Cluster Analysis, pages 33–57. Academic Press, 1986.

[36] B. Johanson and R. Poli. Gp-music: An interactive genetic programming system for music generation with automated fitness raters. In *Genetic Programing 98*, pages 181–186. Morgan Kaufmann Publishers, 1998.

[37] A. K. De Jong. *An Analysis of the Behavior of a Class of Genetic Adaptive Systems*. PhD thesis, University of Michigan, 1975.

[38] K-Team. *Khepera II User Manual. Version 1.1*, 2002.

[39] J. Kennedy and R. Eberhart. Particle swarm optimization. *In Proceedings of the IEEE International Conference on Neural Networks*, 4:1942–1948, 1995.

[40] J. Kennedy, R. Eberhart, and Y. Shi. *Swarm Intelligence*. Morgan Kaufmann Publishers, 2001.

[41] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi. *Optimization by Simulated Annealing*, pages 671–680. Science, 200 (4598), 1983.

[42] T. Kohonen. *Self-organizing maps*. Springer series in information sciences. Springer, 2001.

[43] R. Lahoz-Beltrá. *Bioinformática: simulación, vida artificial e inteligencia artificial*. Díaz de Santos, 2004.

[44] L. Lanzarini and L. Corbalán. Evolving neural arrays. a new mechanism for learning complex action sequences. *CLEI Electronic Journal. Special Issue of Best Papers presented at CLEI'2002*, 6(1), 2003.

[45] L. Lanzarini, G. Osella Massa, and H. Vinuesa. Modular creation of neuronal networks for autonomous robot control. *Revista Iberoamericana de Inteligencia Artificial*, 11(35):43–53, 2007.

[46] L. Lanzarini, C. Sanz, M.Naiouf, and F. Romero. Mixed alternative in the assignment by classes vs conventional methods for calculation of individuals lifetime in gavaps. *Proceedings of the 22nd International Conference on Information Technology Interfaces, ITI 2000*, 953-96769-1-6:383–389, 2000.

[47] L. Lanzarini and H. Vinuesa. Neural networks elitist evolution. In *29th International Conference on Information Technology Interfaces. ITI 2007*, 2007. Artículo completo – Publicado por IEEE Computer Society Press – Pág. 457-462.

[48] E. Lawer, J. K. Lenstra, A. H. G. Rinnooy Kan, and D. B. Shmoys. *The Traveling Salesman Problem*. John Wiley and Son, New York, NY, 1985.

[49] J.-P. Li, X.-D. Li, and A. Wood. Species based evolutionary algorithms for multimodal optimization: A brief review. In *WCCI 2010 IEEE World Congress on Computational Intelligence*, 2010. Barcelona, Spain.

[50] L.H. López. *Predicción y optimización de emisores y consumo mediante redes neuronales en motores diesel*. Reverté, 2006.

[51] K.F. Man, K.S. Tang, and S. Kwong. *Genetic algorithms: concepts and designs*. Number v. 1 in Advanced textbooks in control and signal processing. Springer, 1999.

[52] MathWorks. R2010b Documentation, Matlab. Random Numbers. Website, 2010. http://www.mathworks.com/help/techdoc/math/brnuahp.html.

[53] D. E. Moriarty and R. Miikkulainen. Efficient reinforcement learning through symbiotic evolution. In *Machine Learning*, volume 22, pages 11–33. Department of Computer Sciences, The University of Texas at Austin, 1996.

[54] A.D. Muñoz, J.J.P. Fernández, and M.G. Carrillo. *Metaheurísticas*. Ciencias experimentales y tecnología. Dykinson, 2007.

[55] N.J. Nilsson. *Artificial Intelligence: a new synthesis*. The Morgan Kaufmann Series in Artificial Intelligence Series. Morgan Kaufmann Publishers, 1998.

[56] C. H. Papadimitriou and K. Steiglitz. *Combinatorial Optimization - Algorithms and Complexity*. Dover Publications, Inc., New YorkY, 1982.

[57] M.M. Raghuwanshi and O.G. Kakde. Genetic algorithm with species and sexual selection. In IEEE, editor, *2006 IEEE Conference on Cybernetics and Intelligent Systems*, pages 1–8. 2008.

[58] I. Rechenberg. *Evolutionsstrategie: Optimierung Technischer Systeme Nach Prinzipien der Biologischen Evolution*. Fromman-Holzboog Verlag, 1973.

[59] C. B. Reeves. *Modern Heuristic Techniques for Combinatorial Problems*. John Wiley and Sons, 1993.

[60] G.V. Reina and E.M. García. *Sistemas evolutivos y selección de indicadores*. Universidad de Sevilla, 2004.

[61] R. Rosenfeld and J. Irazábal. *Teoría de la computación y verificación de programas*. Edulp-Universidad Nacional de La Plata, 2010.

[62] G. Sotolongo-Aguilar and M. V. Guzmán-Sánchez. Aplicaciones de las redes neuronales, el caso de la bibliometría. *Ciencias de la Información*, 32(1), 2001.

[63] W.M. Spears. *Evolutionary algorithms: the role of mutation and recombination*. Natural computing series. Springer, 2000.

[64] K.O. Stanley and R. Miikkulainen. Evolving neural networks through augmenting topologies. *Evolutionary Computation*, 10:99–127, 2002.

[65] T. Stützle. *Local Search Algorithms for Combinatorial Problems Analysis, Algorithms and New Applications*. Technical report, DISKI Dissertationen zur Künstliken Intelligenz, 1999.

[66] M. O. Tokhi, M. A. Hossain, and M. H. Shaheed. *Parallel computing for real-time signal processing and control*. Advanced textbooks in control and signal processing. Springer, 2003.

[67] B. Wilkinson and M. Allen. *Parallel Programming. Techniques and Applications Using Networked Workstations and Parallel Computers*. Pearson Prentice Hall, 2da edición edition, 2005.

[68] R. Wright and N. Gemelli. Adaptive state space abstraction using neuroevolution. In *Agents and Artificial Intelligence: International Conference, ICAART 2009, Porto, Portugal, January 19-21, 2009. Revised Selected Papers*, volume 67 of *Communications in Computer and Information Science*, pages 84–96. Springer, 2010.