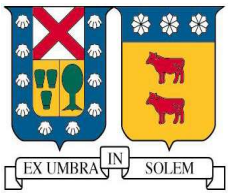


Universidad Técnica Federico Santa María
Departamento de Electrónica

Navegación robótica basada en aprendizaje evolutivo de acciones mediante lógica difusa

Memoria presentada por : **Fernando Ernesto Quirós Retamales**

Como requisito parcial para optar al título de :
Ingeniero Civil Electrónico, mención Computadores y Control Automático

Profesor Guía : **Tomás Arredondo Vidal**

Profesor Correferente : **Wolfgang Freund Grunewaldt**

Valparaíso, 20 de octubre de 2006

Dedicado a mis padres

Índice general

1. Introducción **8**

2. Marco teórico **10**
2.1. Sensores 10
2.2. Redes neuronales artificiales 12
2.2.1. Neuronas reales 12
2.2.2. Neuronas artificiales 13
2.2.3. Redes neuronales artificiales 14
2.2.4. Requerimientos del robot 16
2.3. Algoritmos genéticos 17
2.3.1. Evolución natural y algoritmos evolutivos 17
2.3.2. Definiciones 17
2.3.3. Procedimientos 18
2.3.4. Requerimientos del robot 24
2.4. Lógica difusa 25
2.4.1. Introducción a la lógica difusa 25
2.4.2. Conceptos básicos 25
2.4.3. Control difuso 28
2.4.4. Método Takagi-Sugeno 30
2.4.5. Requerimientos del robot 31
2.5. Redes neuronales SOM 32
2.5.1. Estructura de una red SOM 32
2.5.2. Entrenamiento 32
2.5.3. Funcionamiento 33
2.5.4. Requerimientos del robot 34
2.6. Action based environmental modeling 35
2.6.1. Funcionamiento de AEM 35
2.6.2. Bloque AEM 36
2.6.3. Requerimientos del robot 36
2.7. Chain coding 37

3. Desarrollo **38**
3.1. Procesos 39
3.1.1. Red neuronal artificial 39

3.1.2. Algoritmo genético . . . 40
3.1.3. Fitness difuso . . . 41
3.1.4. Bloque AEM . . . 42
3.1.5. Driver . . . 42
3.1.6. Simulador de entornos . . . 42
3.1.7. Chain coding . . . 42
3.1.8. Reconocimiento de entornos mediante la red SOM . . . 42
3.2. Datos . . . 43
3.2.1. Peso_1, peso_n . . . 43
3.2.2. Mapa . . . 43
3.2.3. Action sequence, environment vector . . . 44
3.2.4. Homing, area, carga batería, entorno . . . 44
3.3. Salidas . . . 45
3.3.1. Motores . . . 45
3.4. Arquitectura de comportamientos . . . 45
3.5. Simulaciones . . . 47
3.6. Criterios para realizar las pruebas . . . 47
3.7. Entrenamiento del robot . . . 48
3.8. Resultados . . . 49

4. Conclusiones **54**

5. Mejoras al comportamiento del robot **56**
5.1. Mejorar el mapa . . . 56
5.2. Realizar el reconocimiento de entornos a partir del mapa . . . 57
5.3. Reconocimiento de entornos en tiempo real . . . 57
5.4. Más comportamientos . . . 57
5.5. Utilizar una red neuronal dinámica . . . 58

6. Apéndice **61**
6.1. SomPak . . . 61
6.1.1. Instalación del programa . . . 61
6.1.2. Comandos . . . 62
6.2. Simulador YAKS . . . 63
6.2.1. Requerimientos para compilar el programa . . . 63
6.2.2. Makefile . . . 64
6.2.3. Archivos . . . 64
6.3. Configuración . . . 66
6.3.1. Opciones generales . . . 69
6.3.2. Forma de hacer evolucionar la población . . . 69
6.3.3. Forma de correr una simulación gráfica, con una red neuronal ya entrenada . . . 70

Índice de figuras

2.1. Foto del Robot Khepera. 10
2.2. Disposición física de los sensores. 11
2.3. Neurona biológica. 12
2.4. Conexión sináptica. 12
2.5. Neurona artificial. 13
2.6. Red neuronal artificial. 14
2.7. Estructura de un cromosoma. 18
2.8. Distribución de población inicial. 19
2.9. Función objetivo. 21
2.10. Población inicial. 21
2.11. Selección mediante probabilidades. 22
2.12. Gráfico de probabilidades tipo torta. 22
2.13. Ejemplo cruce n-puntos. 23
2.14. Ejemplo cruce uniforme. 23
2.15. Ejemplo de mutación. 24
2.16. Algunas funciones de pertenencia. 26
2.17. Mecanismo de inferencia aproximada. 28
2.18. Ejemplo de control difuso. 30
2.19. Estructura de una red SOM. 32
2.20. Ejemplo del funcionamiento de una red SOM. 34
2.21. Mapeo realizado por AEM. 35
2.22. Bloque AEM. 36
2.23. Ejemplo de vector de secuencia de acciones AS. 37

3.1. Arquitectura del sistema que estructura la inteligencia del robot 38
3.2. Estructura de la red neuronal del robot. 39
3.3. Entrenamiento de la red neuronal. 40
3.4. Entrenamiento de la red SOM. 43
3.5. Ejemplo de un mapa y su matriz asociada. 44
3.6. Diagrama de comportamientos del robot. 45
3.7. Entornos utilizados en los experimentos. 47
3.8. Screenshots para set de motivaciones 1, piezas 1 a 6 50
3.9. Screenshots para set de motivaciones 2, piezas 1 a 6 51
3.10. Screenshots para set de motivaciones 3, piezas 1 a 6 52

5.1. Red neuronal dinámica. 58

Índice de tablas

3.1. Parámetros del algoritmo genético y redes neuronales. 48
3.2. Estadísticas para el set de motivaciones 1 53
3.3. Estadísticas para el set de motivaciones 2 53
3.4. Estadísticas para el set de motivaciones 3 53
3.5. Estadísticas para el set de motivaciones 4 53

4.1. Promedios de las tablas de resultados 54

Capítulo 1

Introducción

Hoy en día la robótica móvil es un tema en continuo desarrollo. Con el enorme poder de cómputo que existe en la actualidad se puede dar cada vez más inteligencia a robots móviles para que ejecuten tareas de interés. Además, existe un gran avance en temas relacionados con la inteligencia artificial como redes neuronales, algoritmos genéticos, etc., los cuales pueden ser implementados en un computador, obteniendo un gran potencial para mejorar el desempeño de los robots.

En la actualidad existen numerosos estudios relacionados con la robótica móvil. Se han desarrollado diversos métodos para lograr que un robot se mueva en un entorno, para lograr variados propósitos como: evadir obstáculos, navegar a un sitio especifico, hacer un mapa interno de un entorno, etc. Yamada [1] utiliza modelamiento basado en acciones entrenado con métodos evolutivos para que un robot tipo Khepera se mueva y reconozca habitaciones de distintas formas. Kubota [2] utiliza control difuso y un método llamado Perceptrón-Based GA para que un robot móvil se mueva en entornos con obstáculos dinámicos. Izumi [3] utiliza un comportamiento basado en control difuso para que un robot móvil evada obstáculos. Hagras [4] utiliza un método fuzzy-GA para que un robot aprenda a moverse en tiempo real en entornos no estructurados con obstáculos.

En el presente trabajo se propone desarrollar un sistema para el reconocimiento de entornos en un simulador de un robot móvil tipo Khepera capaz de adquirir comportamientos complejos mediante aprendizaje evolutivo. El robot que se desea implementar, en un principio, posee poca información de su entorno. Sus acciones son consecuencia inmediata de aquello que censan sus sensores, procesado por una red neuronal, que controla la velocidad de los motores. A medida que el robot recorre los entornos va generando un mapa con las zonas del entorno visitadas, que posteriormente permitirá realizar otras funciones.

Para el desarrollo de este trabajo se utilizarán diversas técnicas de soft computing como modelamiento basado en acciones (AEM), lógica difusa (FL), redes neuronales artificiales (ANN), redes auto-organizativas (SOM) y algoritmos genéticos (GA).

El trabajo comienza con el marco teórico, en el que se explican las técnicas utilizadas de soft computing. A continuación, en el desarrollo, se explica como se estructura la inteligencia del robot, utilizando las técnicas previamente explicadas. Luego se presentan los experimentos realizados y las conclusiones obtenidas de ellos. Finalmente se enumeran algunas posibles mejoras que se podrían realizar a la inteligencia del robot. Este trabajo se desarrolló en paralelo con el proyecto IRMA (Investigación Robótica Móvil Autónoma) [5], [6].

Capítulo 2

Marco teórico

2.1. Sensores

Para realizar los experimentos se utilizó un simulador de un robot tipo Khepera [7], el cual mide 55[mm] de diámetro y 30 [mm] de altura. Posee ocho sensores infrarrojos de proximidad, que tienen un alcance de más de 100 [mm]. Se mueve mediante dos ruedas, cada una siendo controlada independientemente. El robot puede estar conectado con el computador y la alimentación a través de un cordón umbilical. En la figura 2.1 se aprecia una fotografía del robot:

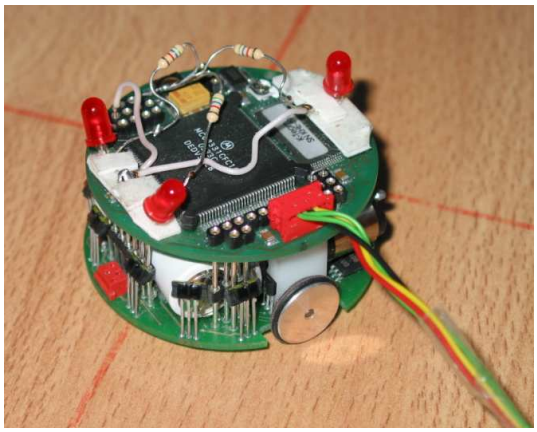


Figura 2.1: Foto del Robot Khepera.

Al ser un robot robusto y de bajo costo no posee cámaras. Las únicas entradas que tiene son sensores infrarrojos de proximidad y de luz. Sin embargo estos últimos no se utilizarán. Se usan ocho sensores infrarrojos con la siguiente disposición:

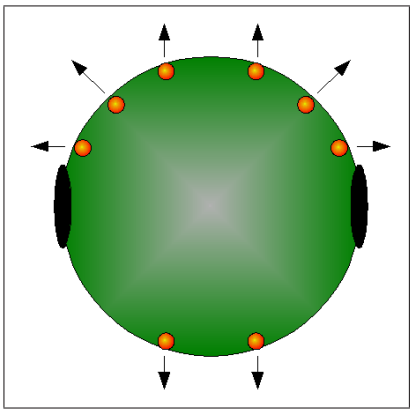


Figura 2.2: Disposición física de los sensores.

Cada sensor puede generar valores entre 0 y 1: 0 indica que no hay obstáculo a más de 100[mm] y 1 que hay un obstáculo a pocos milímetros. De esta forma, el robot sólo conoce su entorno mediante una entrada de dimensión ocho. No es mucha información para navegar en un entorno, sin embargo es suficiente, en un comienzo, para navegar sin chocar con los obstáculos y paredes existentes. Más adelante se verá que el robot, pese a que censa poca información de su entorno, a medida que recorre los entorno será capaz de recolectar y procesar estos datos, lo que le permitirá posteriormente navegar de mejor forma.

2.2. Redes neuronales artificiales

2.2.1. Neuronas reales

Las redes neuronales artificiales imitan el comportamiento de las neuronas reales, que son las células que componen los cerebros de los seres vivos. Una neurona real tiene la siguiente estructura:

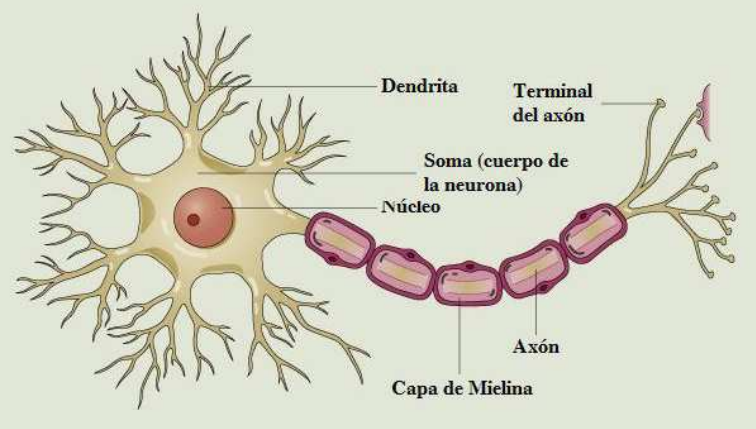


Figura 2.3: Neurona biológica.

Como se aprecia en la figura 2.3, una neurona [8] está compuesta principalmente por: el cuerpo celular y el núcleo, que son los que procesan las señales de entrada y disparan la señal de salida; las dendritas, que son las que reciben estímulos de otras neuronas y el axon, que es el que transmite la señal de salida.

La conexión entre el terminal de un axon y la dendrita de otra neurona se llama sinapsis, la cual puede ser de tipo químico o eléctrico (iónico). No existe una unión física entre las neuronas, sino que existe un espacio llamado espacio sináptico, a través del cual viajan neurotransmisores (o un impulso eléctrico). Estas conexiones son dinámicas, ya que cuando un ser vivo aprende, se crean, eliminan y modifican las uniones entre neuronas. En la figura 2.4 se muestra la sinapsis:

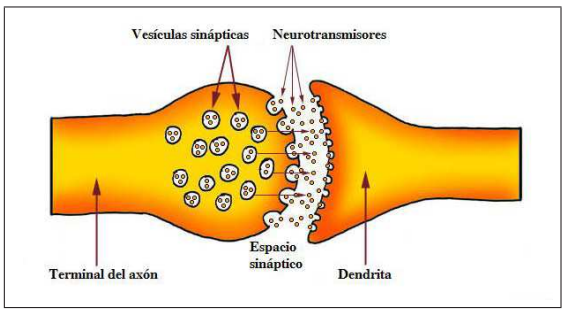


Figura 2.4: Conexión sináptica.

El impulso recibido por una dendrita puede ser exitatorio o inhibitorio y de diversas intensidades, dependiendo de la fuerza de la conexión establecida. La sumatoria de los impulsos que llegan por todas las sinapsis que se relacionan con cada neurona (1000 a 200.000) determina si se produce o no la descarga del potencial de acción por el axon de esa neurona.

2.2.2. Neuronas artificiales

Una neurona artificial corresponde a un modelo que puede ser implementado en un hardware especializado o un segmento de código de programación que emula el funcionamiento de la neurona biológica. Su forma de operar [9] se explica en la figura 2.5:

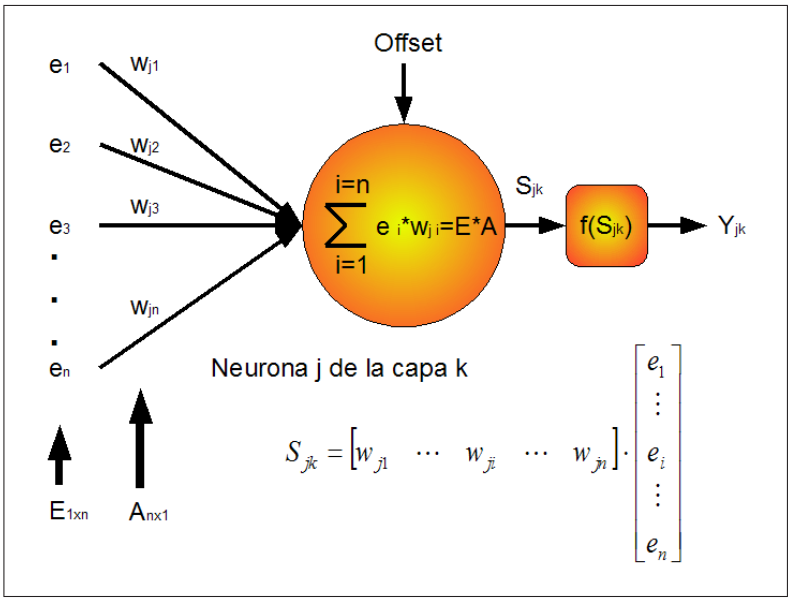


Figura 2.5: Neurona artificial.

En la figura 2.5, las señales de entrada que llegan a la neurona (axones de otras neuronas) están representadas por las variables e_i, mientras que los pesos sinápticos están representados por las constantes [1] $w_{j,i}$. Cada entrada es ponderada por su respectivo peso sináptico. Los valores obtenidos son sumados entre sí y con el valor del offset, obteniéndose $S_{j,k}$. Este resultado corresponde al argumento de la función de salida $f(S_{j,k})$. Existen varias funciones de salida, entre las cuales se destacan la función lineal, tangente hiperbólica, función escalón, función saturación, etc. Esta función amplía el campo de problemas que pueden resolver las redes, como problemas no lineales.

Hay que tener presente que los subíndices i, j, k son utilizados para identificar las señales, constantes y neuronas, de manera unívoca en una red neuronal. En una sola neurona no tienen utilidad.

[1] Los valores de los $w_{j,i}$ son variables mientras la red aprende. Luego de esto, son valores constantes.

2.2.3. Redes neuronales artificiales

Una red neuronal artificial es una agrupación de neuronas artificiales, que operan en conjunto.

Si una neurona se considera como estructura elemental, al realizar agrupaciones de ellas, las posibilidades de interconexiones son ilimitadas. Se debe escoger una arquitectura de red adecuada para el problema a resolver. No existen reglas rígidas para la construcción de redes, aunque existen algunas reglas empíricas para su construcción.

Por ejemplo, para resolver problemas complejos se recomienda aumentar el número de neuronas, pero aumentarlo indefinidamente no mejorará indefinidamente el desempeño de la red. Hay un rango óptimo de número de neuronas para cada problema. Normalmente las neuronas son agrupadas en capas [10], como se ve en la figura 2.6. Cada capa puede tener distinta cantidad de neuronas y las neuronas de una capa pueden o no estar conectadas entre ellas. Además, las capas pueden conectarse entre sí de distintas maneras.

En la siguiente figura se muestra una red neuronal típica, de tres capas, de tamaños p, q y r, feed-forward (no hay realimentaciones entre las capas):

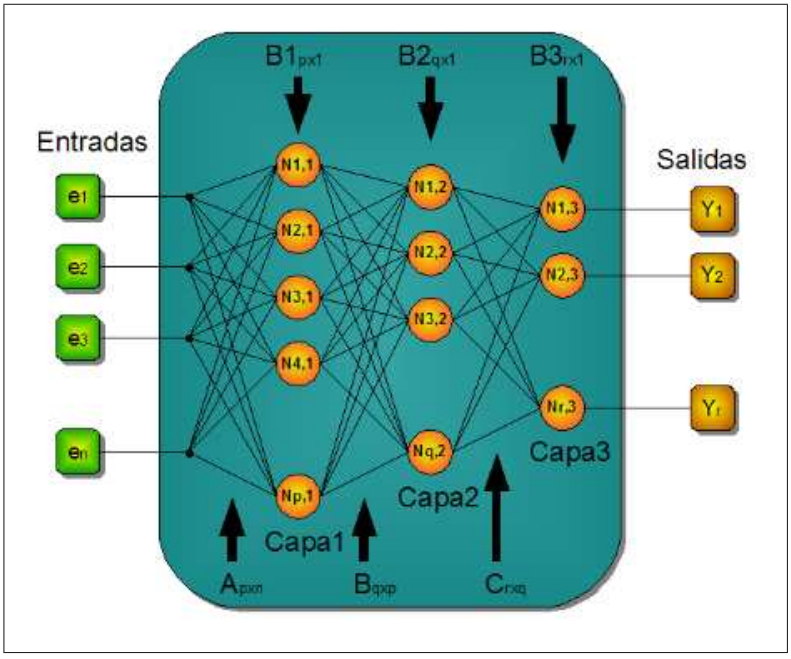


Figura 2.6: Red neuronal artificial.

Como se aprecia en la figura 2.6, los pesos entre capas pueden ser representados como matrices ($A_{p\times n}$, $B_{q\times p}$, $C_{r\times q}$). Por otro lado, los offsets de las capas pueden ser representados como vectores ($B1_{p\times 1}$, $B2_{q\times 1}$, $B3_{r\times 1}$). Es una manera simple y ordenada de manipular estos valores en un programa computacional. La red de la figura 2.6 posee 3 matrices de pesos y 3 vectores de bias, los que representan la inteligencia de la red neuronal.

Existen también otras consideraciones para tener en cuenta al momento de diseñar una red. Por ejemplo hay redes en que las salidas de algunas capas se realimentan en las entradas de otras capas. Otras redes utilizan valores presentes y pasados de las entradas, a través de buffers (redes dinámicas). Se debe escoger la función de salida de las neuronas, se puede utilizar o no bias. Hay redes con entrenamiento supervisado y no supervisado [11]. Cada tipo de red neuronal puede desempeñar diversas tareas, como por ejemplo reconocimiento de patrones, aprendizaje de reglas, modelamiento de procesos, etc.

Definición y arquitectura de la red

Lo primero que se debe hacer es escoger una arquitectura adecuada para la red. Para ello se debe tener presente el tamaño del vector de entrada, la cantidad de salidas, el tipo de problema a resolver, su complejidad, etc. Algunos de los parámetros de diseño a considerar son:

- Número de capas
- Cantidad de neuronas por capa
- Tipo de función de salida de las neuronas
- Red dinámica o estática
- Aprendizaje supervisado o no supervisado
- Algoritmo de entrenamiento
- Si hay o no realimentación entre neuronas.

Existen diversas aplicaciones que permiten implementar redes neuronales. Una de las más potentes es el toolbox de Matlab, que permite definir de manera independiente cada uno de estos parámetros y otros más. También es posible programar redes neuronales en C/C++/Java, o cualquier lenguaje de programación, en cuyo caso habría que implementar todas las funciones a utilizar.

Entrenamiento de la red

Una vez escogida la arquitectura de la red, ésta debe ser entrenada para que aprenda a realizar la tarea que se le asignó. Mediante el proceso de entrenamiento, los pesos sinápticos de las neuronas son continuamente modificados, hasta que se determine, mediante algún criterio, que la red ya ha aprendido.

Existen diversos métodos para entrenar una red. Algunos de ellos son: gradiente descendiente y métodos evolutivos. Para el entrenamiento de esta red se escogió un método evolutivo, el cual se explicará en el capítulo 2.3. Hay que tener presente que el desempeño de la red está estrechamente relacionado con los pesos sinápticos, ya que estos determinan

su funcionamiento, por lo que la etapa de entrenamiento es esencial. Un criterio utilizado para determinar si la red ha aprendido, es el error entre las entradas deseadas y las obtenidas con la red. Muchas veces se utiliza la distancia Euclidiana para medir este error.

Prueba de la red

Para probar una red basta con presentarle valores de entrada y comparar la salida obtenida con los valores deseados para dicha entrada. Si los valores son similares la red aprendió satisfactoriamente. En caso contrario puede que falten iteraciones para que la red aprenda, o puede que la arquitectura propuesta no sea la adecuada, con lo que habría que variar algunas características de la estructura de la red.

2.2.4. Requerimientos del robot

El robot requiere de un método que controle la velocidad de los motores, a partir de las señales de entrada. Para este efecto se utilizó una red neuronal, implementando navegación basada en comportamiento ("Behavior Based Navigation") [12], [13]. Debido a que la dimensión de la entrada no es tan elevada se propondrá una red simple, con pocas neuronas, y según se requiera, se aumentará su complejidad. La idea es implementar un comportamiento del tipo reflejo: las mediciones de los sensores generan una respuesta inmediata, a través de la red, la que controla los motores.

2.3. Algoritmos genéticos

2.3.1. Evolución natural y algoritmos evolutivos

La selección natural es un mecanismo esencial de evolución propuesto por Charles Darwin [14] y generalmente aceptado por la comunidad científica como la mejor explicación para la generación de especies o especiación.

El concepto básico de la selección natural se basa en que las condiciones de un medio ambiente (o naturaleza) determinan (o seleccionan) la eficacia de ciertas características en algunos organismos para su supervivencia y reproducción. Dada la presión evolutiva que ejerce el medio ambiente, las particularidades más exitosas se irán distribuyendo en toda la población.

La selección natural puede ser expresada como la siguiente ley general (tomada de la conclusión de *El origen de las especies*, 1859): si existen organismos que se reproducen, y si la cría hereda características de sus progenitores, y si existen variaciones de características y si el medio ambiente no admite a todos los miembros de una población en crecimiento, entonces aquellos miembros de la población con características menos adaptadas (determinadas por el medio ambiente) morirán y aquellos miembros con características mejor adaptadas sobrevivirán. El resultado es la evolución de las especies.

Los algoritmos evolutivos imitan este lento, pero efectivo proceso natural, a través de códigos de programación, aprovechando el enorme poder de cómputo actual. Lo que se logra es una poderosa herramienta capaz de encontrar mínimos o máximos [2] globales de funciones complejas, multivariables, difíciles o imposibles de derivar, o simplemente desconocidas. Normalmente esta función es llamada función objetivo.

2.3.2. Definiciones

A continuación se presentan algunas definiciones simples [16] para entender mejor el proceso de selección natural:

- Función objetivo. Aquella función que se desea minimizar (o maximizar).
- Individuo. Es la representación de una solución o un punto (coordenadas) dentro del dominio de la función objetivo.
- Padres. Corresponde a un grupo de individuos, escogidos mediante algún proceso de selección, de tal forma que estén bien adaptados al medio (un padre evaluado en la función objetivo da un valor pequeño [3] respecto al resto de la población).

[2] En el desarrollo de este capítulo se considerará que el objetivo es minimizar una función.
[3] Pequeño ya que se está minimizando la función.

- Hijo de dos individuos. Individuo descendiente de dos padres. Sus genes corresponden a la mezcla de los genes de los padres. De esta forma, se espera que los hijos estén mejor adaptados que los padres.
- Población inicial. Es el conjunto de puntos o individuos en el momento en el que se inicia el algoritmo.
- Fenotipo. Representa las características expresadas que posee un individuo.
- Genotipo. Corresponde a la codificación del fenotipo de un individuo. Normalmente se utiliza la representación binaria del fenotipo.
- Adaptación de un individuo al medio. Indica su proximidad al mínimo buscado de la función.
- Número máximo de la población. Es el número de individuos que soporta el medio. Corresponde al punto fijo de la ecuación logística para ese ecosistema.
- Muerte. Corresponde al proceso por el cual se eliminan a los individuos menos adaptados.
- Mutación. Corresponde al proceso por el cual se realizan cambios aleatorios a los genes de algunos individuos, con una cierta probabilidad de ocurrencia.

2.3.3. Procedimientos

Representación de un individuo

El primer paso consiste en escoger la forma en que se representarán los individuos. Cada uno de ellos está representado por un string (cadena) de bits llamado cromosoma. Éste contiene las características del individuo y sobre él se realizarán las operaciones genéticas. En la figura 2.7 se presenta un ejemplo de un cromosoma:

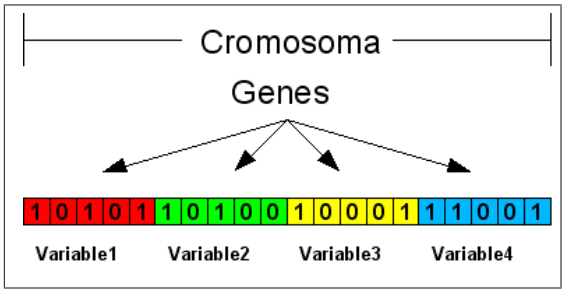


Figura 2.7: Estructura de un cromosoma.

En la figura 2.7, el cromosoma está compuesto de cuatro genes de cinco bits cada uno. Cada gen representa una variable de la función objetivo. El largo de cada gen está relacionado con la precisión que se desea para cada variable.

Por ejemplo, el genotipo del cromosoma de la figura 2.7 está representado por el string binario 10101101001000111001, mientras que el fenotipo, si se considera cada gen como un entero positivo de 5 bits, sería: $(21, 20, 17, 25)$. Hay otras codificaciones posibles, usando alfabetos de diferente cardinalidad. Sin embargo, uno de los resultados fundamentales en la teoría de algoritmos genéticos, el teorema de los esquemas, afirma que la codificación óptima, es decir, aquella sobre la que los algoritmos genéticos funcionan mejor, es aquella que tiene un alfabeto de cardinalidad 2 [15].

Creación de población inicial

Antes de comenzar el algoritmo genético, se debe generar un conjunto de individuos, llamado población inicial. Esta población puede ser generada de manera aleatoria (strings aleatorios de bits) o determinista [17], en cuyo caso se intenta cubrir homogéneamente el dominio de la función objetivo. En la figura 2.8 se presenta un ejemplo de población inicial, para una función de dominio $[0, 4], [0, 3]$:

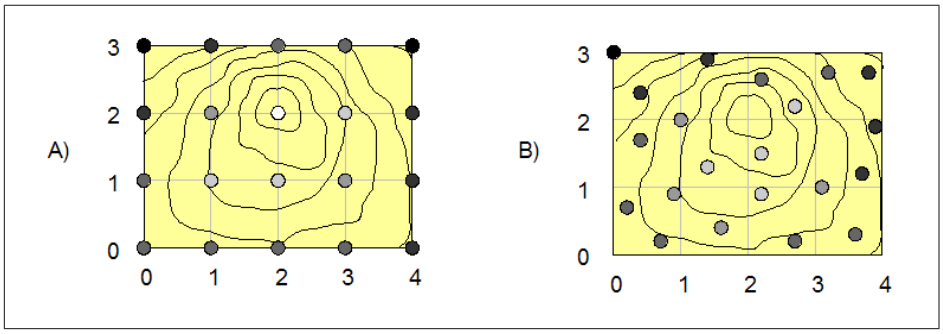


Figura 2.8: Distribución de población inicial.

En la figura 2.8 *A* se escogió la población de manera determinista, mientras que en la figura 2.8 *B* se escogió aleatoriamente. Es importante tener en cuenta que el número de individuos de la población inicial afectará el rendimiento del algoritmo. Un número muy pequeño implica una demora en encontrar el punto mínimo, ya que no se puede cubrir satisfactoriamente todo el dominio de la función. Por otro lado, un número muy grande produce un aumento en el número de cálculos a ejecutar, lo que también disminuye la velocidad del algoritmo. En la figura 2.8 la población inicial es de 20 individuos.

Operaciones básicas

En los algoritmos genéticos existen diversas operaciones que pueden aplicarse a los individuos de una población. En este trabajo analizaremos las siguientes operaciones:

- Selección
  - mediante ranking
  - mediante torneo

    - mediante ruleta

- Cruce
- Mutación
- Elitismo.

a) Selección mediante ranking

En el operador de selección basado en el ranking, los cromosomas se ordenan de acuerdo a sus valores para la función de adaptación (fitness). Luego, se seleccionan para la reproducción a los primeros m (la cantidad que sea necesaria) cromosomas. Este es uno de los métodos más simples de selección, ya que sólo basta ordenar a los individuos según sus valores de fitness.

b) Selección mediante torneo

El operador de selección por torneo permite controlar en forma efectiva la presión selectiva del algoritmo genético, siendo a la vez de fácil implementación. En este esquema, se toman T individuos al azar de la población (donde T es el tamaño del torneo, habitualmente 2 ó 3 individuos), de los cuales se selecciona para la fase de reproducción, con probabilidad p (generalmente entre 0,7 y 0,8), aquel que tenga el mayor valor de la función de adaptación.

Los parámetros T y p permiten regular la presión selectiva. Cuanto más grandes son los valores de T y p, mayor es la presión selectiva. En el caso extremo de que p sea igual a 1 y T igual al tamaño de la población, el algoritmo genético solamente seleccionará al mejor individuo de la población.

En el otro extremo, si T es igual a 1, se logra la presión selectiva más baja (los cromosomas se seleccionan al azar). Manteniendo estos parámetros constantes, se logra una presión selectiva que es independiente de los valores absolutos de aptitud de la población, y sin requerir la aplicación de funciones de escala sobre la función de adaptación.

c) Selección mediante ruleta

A cada individuo se le asigna un valor de adaptación que depende directamente del valor de la función objetivo evaluada en el individuo [18]. Los individuos mejor adaptados tienen mayor probabilidad de seguir vivos y procrear, mientras que los menos adaptados son eliminados. Se utilizará un ejemplo para explicar este método.

Se tiene la siguiente función objetivo:

$$f(x) = 10 + x \cdot \sin(x), x\epsilon[0, 8] \tag{2.3.1}$$

De la función 2.3.1 se obtiene la figura 2.9:

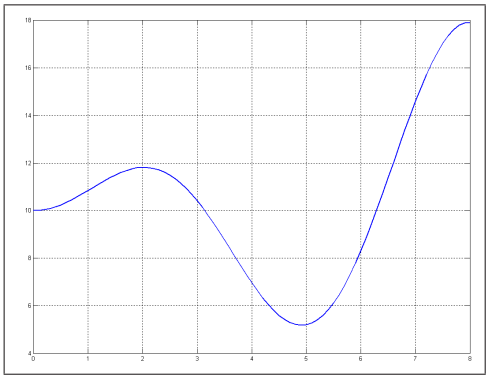


Figura 2.9: Función objetivo.

El mínimo de la función 2.3.1 se encuentra para un x cercano a cinco. Se considera una población inicial de cuatro individuos, cada uno de ellos compuesto por un cromosoma de un gen, ya que la función tiene como dominio los reales entre 0 y 8 (una sola dimensión).

En la figura 2.10 se presenta la población inicial que fue escogida aleatoriamente, los fenotipos, genotipos y valores de la función objetivo para cada individuo:

Individuos	Genotipo						Fenotipo (x)	f(x)
1	1	0	1	0	1	0	5,25	5,4906
2	0	1	0	0	0	0	2	11,8186
3	1	1	1	1	1	0	7,75	17,7081
4	0	0	0	0	1	1	0,375	10,1374

Figura 2.10: Población inicial.

Se escogió genes de largo 6 bits, con 3 bits para la parte entera y 3 para la parte decimal. Con este método se otorga una probabilidad proporcional al fitness de cada individuo. En el ejemplo, debido a que se busca un mínimo, la probabilidad de cada individuo podría ser escogida proporcionalmente al inverso de f(x). De esta forma se obtiene la figura 2.11:

Ind	Genotipo						x	f(x)	1/f(x)	Probabilidad
1	1	0	1	0	1	0	5,25	5,4906	0,1821	0,4217
2	0	1	0	0	0	0	2	11,8186	0,0846	0,2
3	1	1	1	1	1	0	7,75	17,7081	0,0565	0,134
4	0	0	0	0	1	1	0,375	10,1374	0,0986	0,234

Suma	0,4218	1
Escalamiento	2,371	

Figura 2.11: Selección mediante probabilidades.

La probabilidad asociada a cada individuo se obtiene escalando la inversa de $f(x)$, de modo tal que la suma de las probabilidades sea 1. Obtenidas las probabilidades, se distribuyen los individuos en un gráfico tipo torta como se muestra en la figura 2.12:

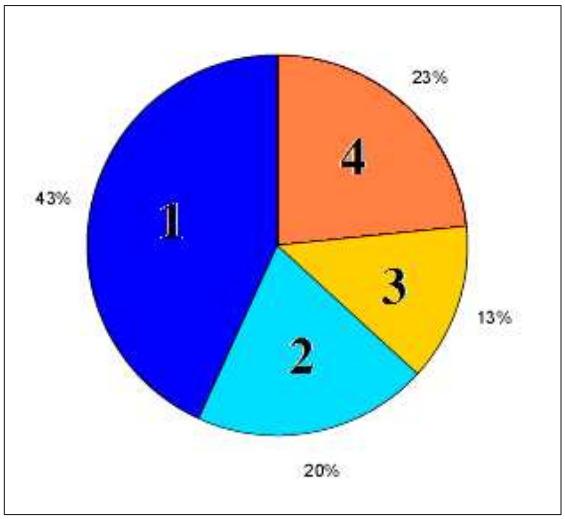


Figura 2.12: Gráfico de probabilidades tipo torta.

Finalmente, se debe obtener un número aleatorio r entre 0 y 1, el cual decidirá cual individuo seguirá vivo según:

$$\begin{aligned} 0,0000 < r < 0,4217 &\rightarrow \text{Vive el individuo 1} \\ 0,4217 < r < 0,6557 &\rightarrow \text{Vive el individuo 4} \\ 0,6557 < r < 0,8557 &\rightarrow \text{Vive el individuo 2} \\ 0,8557 < r < 1,0000 &\rightarrow \text{Vive el individuo 3} \end{aligned} \tag{2.3.2}$$

Luego, los individuos con mayor fitness tienen mayor probabilidad de sobrevivir que aquellos con menor fitness. Este procedimiento también puede ser utilizado para escoger al sub-grupo de sobrevivientes que darán origen a la nueva generación de individuos (hijos). Este método es normalmente llamado método de la ruleta.

d) Cruzamiento

Consiste en el intercambio de material genético entre dos individuos. El cruzamiento (también llamado crossover o recombinación) es el principal operador genético. Para aplicar el cruzamiento, se escogen aleatoriamente dos miembros cualesquiera de la población. No pasa nada si se emparejan dos descendiente de los mismos padres. Sin embargo, si esto sucede demasiado a menudo, puede crear problemas: toda la población puede aparecer dominada por los descendientes de algún gen, que, además, puede tener características no deseadas. Existen distintos tipos de cruzamiento, entre los cuales se destacan el cruzamiento n-puntos y el cruzamiento uniforme [19].

- Cruzamiento n-puntos. Los dos cromosomas se cortan por n puntos y el material genético situado entre ellos se intercambia de manera alternada. En la figura 2.13 se observa un ejemplo de este procedimiento:

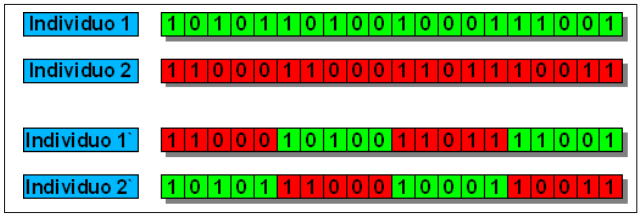


Figura 2.13: Ejemplo cruce n-puntos.

- Cruzamiento uniforme. Se genera un patrón aleatorio de 1's y 0's, y se intercambian los bits de los dos cromosomas que coincidan donde hay un 1 en el patrón. O bien se genera un número aleatorio para cada bit, y si supera una determinada probabilidad se intercambia ese bit entre los dos cromosomas. En la figura 2.14 se observa un ejemplo de este procedimiento:

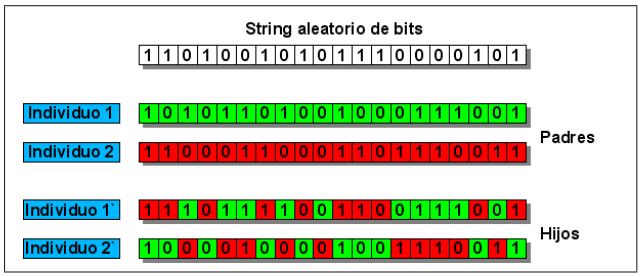


Figura 2.14: Ejemplo cruce uniforme.

e) Mutación

En la evolución, la mutación es un evento de mucha importancia dado que contribuye a que la población no sea homogénea. En muchos casos las mutaciones pueden ser dañinas, pero en promedio contribuyen a la diversidad genética de la especie. En un algoritmo

genético tendrán el mismo papel.

Para determinar qué individuos sufrirán una mutación, se genera un número aleatorio para cada individuo, según una distribución de probabilidad que determinará la frecuencia de ocurrencia de las mutaciones. Cuando uno de estos números supera un valor umbral, se produce una mutación en el individuo.

Las mutaciones son un mecanismo generador de diversidad, y, por tanto, la solución cuando un algoritmo genético está estancado. Pero también es cierto que reduce el algoritmo genético a una búsqueda aleatoria. Siempre es más conveniente usar otros mecanismos de generación de diversidad, como aumentar el tamaño de la población, o garantizar la aleatoriedad de la población inicial.

En la figura 2.15 se ejemplifican tres tipos de mutaciones:

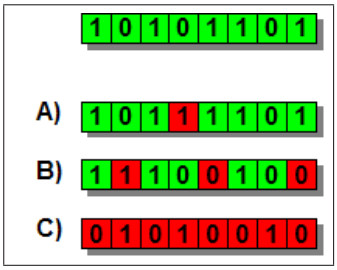


Figura 2.15: Ejemplo de mutación.

En la figura 2.15 se observa el cambio de un bit (A), el cambio de un conjunto de bits (B) y la variación de un individuo completo (C).

f) Elitismo

El elitismo es un criterio que se aplica en un AG con la finalidad de mantener el mejor cromosoma de cada población insertándolo directamente en la siguiente generación [20]. Con esto se asegura que si en determinado momento la heurística de los operadores genéticos no reproduce cromosomas con aptitud superior a la de su población anterior, por lo menos se mantiene el cromosoma con la mejor aptitud hasta esa generación.

2.3.4. Requerimientos del robot

Como se dijo anteriormente, el algoritmo genético se encargará de entrenar la red neuronal, modificando los pesos sinápticos hasta que su desempeño sea el deseado. Por otro lado, la selección se realiza mediante una función fitness implementada con un bloque de lógica difusa, tipo Sugeno [21], que se explicará en el siguiente capítulo.

2.4. Lógica difusa

2.4.1. Introducción a la lógica difusa

El concepto de relación difusa es una generalización del concepto de relación de la teoría clásica de conjuntos. Mientras que una relación entre dos conjuntos clásicos describe la existencia o no de asociación entre los elementos de ambos conjuntos, una relación difusa describe el grado de asociación o interacción entre los elementos de dos o más conjuntos difusos. Los valores de salida de las relaciones difusas no son ambivalentes, 0 ó 1, sino que puede tomar valores reales entre 0 y 1.

Por ejemplo, utilizando la lógica tradicional, los días lunes a viernes pertenecen a los días laborales, mientras que el sábado y el domingo pertenecen al fin de semana, definiéndose los conjuntos *días laborales* y *fin de semana*. Utilizando lógica difusa se definen los mismos dos conjuntos. Luego, el día viernes podría pertenecer un 30 % al fin de semana y un 70 % a los días laborales. Incluso más: el viernes, a las 21:00, se podría considerar que pertenece un 85 % al fin de semana y un 15 % a los días laborales.

La lógica difusa se puede utilizar cuando la complejidad del proceso a estudiar es muy alta o no existen modelos matemáticos precisos, para procesos altamente no lineales y cuando se utilizan definiciones y conocimiento no estrictamente definido (impreciso o subjetivo), con el propósito de cambiar el funcionamiento o el estado del proceso en cuestión. Entonces, con la lógica difusa es posible controlar un proceso por medio de reglas de sentido común, las cuales se refieren a cantidades indefinidas.

Por ejemplo, una regla difusa podría ser: si la temperatura es alta y la presión es baja, entonces apagar la caldera y cerrar la válvula de escape. En este caso las variables son la *temperatura* y la *presión*, mientras que *alto* y *bajo* son los conjuntos a los que pertenecen las variables. Si la temperatura es de 75^oC, el grado de pertenencia de la variable temperatura al conjunto *alto* podría ser de 60 %.

La lógica difusa está relacionada con probabilidades, ya que intenta cuantificar una incertidumbre. Si P es una proposición, se le puede asociar un número $v(P)$ en el intervalo $[0, 1]$ tal que: si $v(P) = 0$, P es falso; si $v(P) = 1$, P es verdadero. La veracidad de P aumenta con $v(P)$.

En este trabajo se utilizará el modelo difuso Takagi-Sugeno [22], el cual utiliza como consecuente una función de las variables de entrada.

2.4.2. Conceptos básicos

Conjuntos difusos y variables lingüísticas

La lógica difusa es una extensión del concepto clásico de conjuntos. El concepto fundamental en que se basa la lógica difusa es el de conjunto difuso, un tipo de conjunto

caracterizado porque los elementos del universo de discurso en el que están definidos pueden pertenecer a él en un cierto grado, representado por una función de pertenencia, como se ve en la figura 2.16.

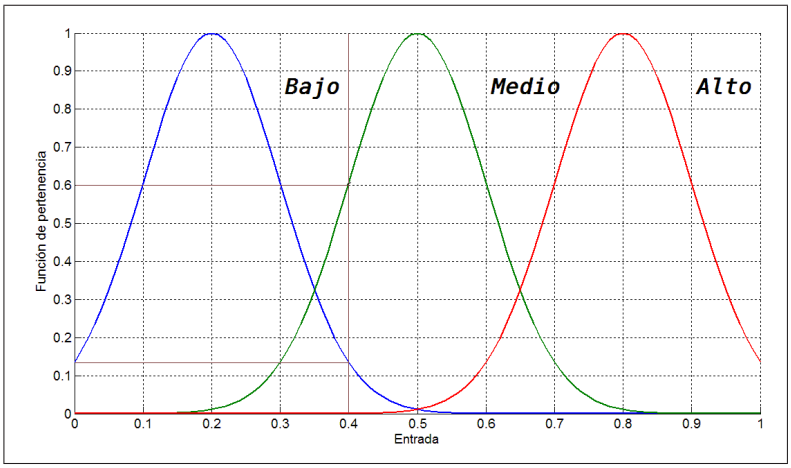


Figura 2.16: Algunas funciones de pertenencia.

Las funciones de pertenencia permiten caracterizar variables lingüísticas, que puedan expresarse en términos del lenguaje natural, como *bajo*, *medio* y *alto*. En la figura 2.16, las funciones de pertenencia caracterizan a la variable *voltaje*. Se pueden utilizar diversas funciones, entre las que se destacan la gaussiana, triangular, trapezoidal, etc. La elección de la forma de la función de pertenencia es subjetiva y dependiente del contexto. Una función de pertenencia puede ser escrita como:

$$F = \{(u, \mu_F(u))|u\epsilon U\} \tag{2.4.1}$$

donde u representa a cada elemento del universo de discurso U y $\mu_F(u)$ es el valor de la función de pertenencia F para el u dado.

Por ejemplo, si en la figura 2.16 el voltaje es $0,4$, el grado de pertenencia al conjunto difuso bajo es $0,14$, al conjunto medio es $0,6$ y al conjunto alto es 0. Utilizando la notación anterior, la expresión para las tres funciones de pertenencia quedaría:

$$[u; \mu_{bajo}(u); \mu_{medio}(u); \mu_{alto}(u)] = [0,4; 0,14; 0,6; 0]. \tag{2.4.2}$$

Operaciones difusas

En la teoría clásica de conjuntos las operaciones elementales son la negación, la unión y la intersección. En la teoría difusa existen estas mismas operaciones, aunque con algunas diferencias [23].

a) Negación

Dado un conjunto F, su complemento está definido por:

$$\mu_{\overline{F}}(u) = c(\mu_F(u)) \tag{2.4.3}$$

donde c es una función $[0,1] \rightarrow [0,1]$ que cumple las siguientes propiedades:

- Concordancia caso nítido: $c(1) = 0$ y $c(0) = 1$
- Estrictamente decreciente: $\forall\ \alpha, \beta \epsilon [0,1] \alpha > \beta \Rightarrow c(\alpha) < c(\beta)$
- Involución: $\forall\ \alpha, \beta \epsilon [0,1] c(c(\alpha)) = \alpha$

b) Union

Dados dos conjuntos A y B, su union está definida por:

$$\mu_{A \bigcup B}(x) = u(\mu_A(x), \mu_B(x)) \tag{2.4.4}$$

donde u es una función $[0,1]x[0,1] \rightarrow [0,1]$ que cumple las siguientes propiedades:

- Concordancia caso nítido: $u(0,1) = u(1,1) = u(1,0) = 1; i(0,0) = 0$
- Conmutatividad: $u(\alpha, \beta) = u(\beta, \alpha)$
- Asociatividad: $u(\alpha, u(\beta, \gamma)) = u(u(\alpha, \beta), \gamma)$
- Identidad: $u(\alpha, 0) = \alpha$
- Monotonía: Si $\alpha \leq \overline{\alpha}, \beta \leq \overline{\beta}$, entonces $u(\alpha, \beta) \leq u(\overline{\alpha}, \overline{\beta})$

Las funciones u que cumplen esta propiedad se llaman normas triangulares o t-normas.

c) Intersección

Dados dos conjuntos A y B, su intersección está definida por:

$$\mu_{A \cap B}(x) = i(\mu_A(x), \mu_B(x)) \tag{2.4.5}$$

donde i es una función $[0,1]x[0,1] \rightarrow [0,1]$ que cumple las siguientes propiedades:

- Concordancia caso nítido: $i(0,1) = i(0,0) = i(1,0) = 0; i(1,1) = 1$
- Conmutatividad: $i(\alpha, \beta) = i(\beta, \alpha)$
- Asociatividad: $i(\alpha, i(\beta, \gamma)) = i(i(\alpha, \beta), \gamma)$
- Identidad: $i(\alpha, 1) = \alpha$
- Monotonía: Si $\alpha \leq \overline{\alpha}, \beta \leq \overline{\beta}$, entonces $i(\alpha, \beta) \leq i(\overline{\alpha}, \overline{\beta})$

Las funciones i que cumplen esta propiedad se llaman conormas triangulares o s-normas.

2.4.3. Control difuso

Un sistema de control difuso clásico está compuesto por tres partes: fuzzificador, motor de inferencia y defuzzificador, como se muestra en la figura 2.17. El núcleo del sistema está formado por una base de conocimiento que permite definir las reglas que describirán la lógica del sistema. El bloque de fuzzificación permite convertir los datos de entrada, que generalmente son valores reales, a variables difusas que puedan ser interpretadas por el motor de inferencia. Por su parte, el defuzzificador permite convertir las variables difusas de salida en valores reales.

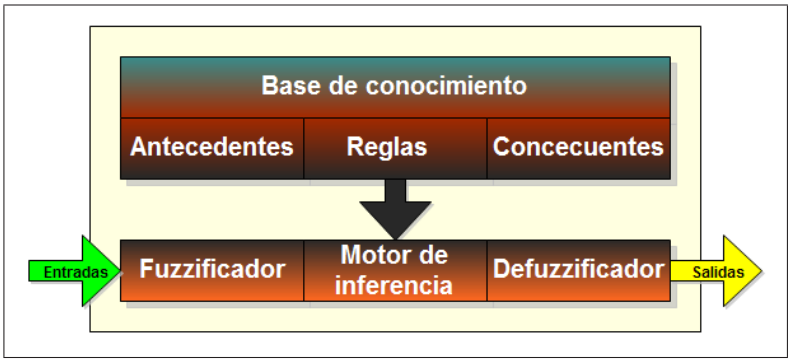


Figura 2.17: Mecanismo de inferencia aproximada.

En un sistema difuso Takagi-Sugeno no se necesita el bloque defuzzificador, ya que se utiliza como consecuente una función de las variables de entrada. De esta forma se obtiene inmediatamente un valor real.

A continuación se explicará con mayor detalle los bloques del sistema fuzzy Takagi-Sugeno.

Fuzzificación

Las mediciones de los fenómenos a controlar se representan con valores reales: rapidez, aceleración, dinero, voltajes, etc. Estos valores no pueden ser utilizados directamente. Deben ser transformados mediante las funciones de pertenencia asociadas a las reglas de inferencia. El motor de inferencia utiliza las salidas de las funciones de pertenencia de las entradas.

Mecanismo de inferencia

El motor de inferencia es el componente principal de un sistema difuso y está compuesto por un conjunto de reglas difusas, como se muestra a continuación:

$$\begin{bmatrix} \text{If } x_1 \text{ is } A_1^1 & \text{and } x_2 \text{ is } A_2^1 & \cdots & \text{and } x_I \text{ is } A_I^1 & \text{Then } z \text{ is } B^1 \\ \text{If } x_1 \text{ is } A_1^2 & \text{and } x_2 \text{ is } A_2^2 & \cdots & \text{and } x_I \text{ is } A_I^2 & \text{Then } z \text{ is } B^2 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ \text{If } x_1 \text{ is } A_1^r & \text{and } x_2 \text{ is } A_2^r & \cdots & \text{and } x_I \text{ is } A_I^r & \text{Then } z \text{ is } B^r \end{bmatrix} \tag{2.4.6}$$

Los x_i son los valores de entrada y los A_i^r son las variables lingüísticas, representadas por conjuntos difusos. La operación lógica z se calcula mediante una función s-norma, mientras que la unión de todas las reglas se realiza mediante una función t-norma. Algunas s-normas y t-normas utilizadas son:

- Método de Mamdani. Considera la función s-norma como el mínimo y la función t-norma como el máximo.
- Método propuesto por Mizumoto. Considera la función s-norma como la multiplicación y la t-norma como la suma.

Defuzzificación

La salida del procedimiento anterior corresponde a una función, por lo que directamente no sirve. Se debe convertir esta función en un valor real representativo. Existen diversos métodos para obtener este valor, entre los que se destacan:

- Centro de gravedad.
- Centro de sumas.
- Primer máximo.
- Último máximo.

A continuación se presenta un ejemplo de control difuso MISO (Multiple Input Single Output), de dos entradas [4] de manera gráfica:

[4]Sistema con múltiples entradas y una salida

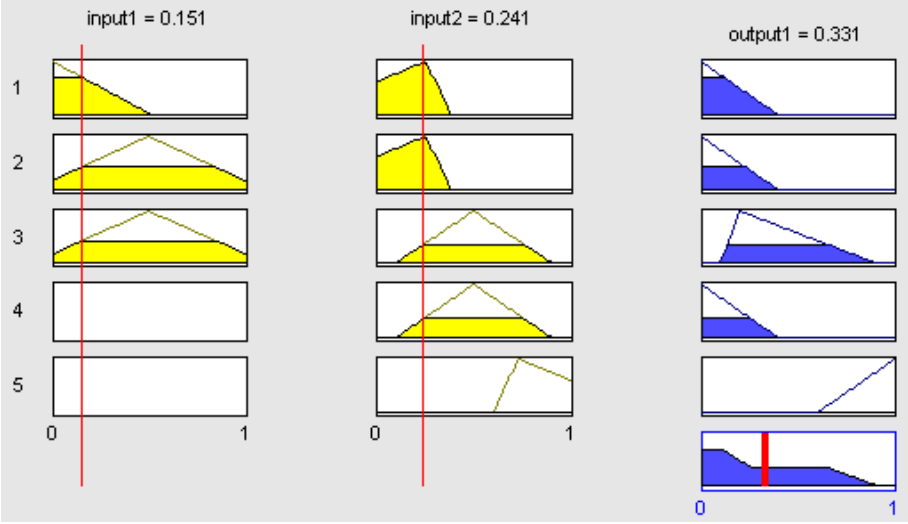


Figura 2.18: Ejemplo de control difuso.

Como se aprecia en la figura 2.18 las entradas valen $x_1 = 0,151$ y $x_2 = 0,241$, con lo que se obtiene una salida de $y = 0,331$. En este ejemplo se usó el método Mamdani. Cada fila representa una regla y cada salida corresponde al mínimo de las dos funciones de pertenencia, para sus respectivas entradas. La salida del sistema se calcula obteniendo el máximo de cada una de las salidas anteriores. La defuzzificación se realiza obteniendo el centroide de la función obtenida.

2.4.4. Método Takagi-Sugeno

En un sistema difuso Takagi-Sugeno, se utiliza como salida para cada regla una combinación lineal de las variables de entrada $x_1 \cdots x_I$, por lo que el motor de inferencia se puede representar de la siguiente forma:

$$\begin{bmatrix} \text{If } x_1 \text{ is } A_1^1 & \text{and } x_2 \text{ is } A_2^1 & \cdots & \text{and } x_I \text{ is } A_I^1 & \text{Then } z \text{ is } B^1 \\ \text{If } x_1 \text{ is } A_1^2 & \text{and } x_2 \text{ is } A_2^2 & \cdots & \text{and } x_I \text{ is } A_I^2 & \text{Then } z \text{ is } B^2 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ \text{If } x_1 \text{ is } A_1^r & \text{and } x_2 \text{ is } A_2^r & \cdots & \text{and } x_I \text{ is } A_I^r & \text{Then } z \text{ is } B^r \end{bmatrix} \tag{2.4.7}$$

Como se aprecia en la figura 2.4.7 la salida está definida como la multiplicación del vector de entrada X y por una matriz de valores reales $A = [a_{1,1} \cdots a_{1,I}] \cdots [a_{r,1} \cdots a_{r,I}]$, con lo que se obtiene un vector de salida. A cada regla le corresponde un elemento de este vector de salida.

2.4.5. Requerimientos del robot

El robot tiene como requerimiento el poder tener una buena interfaz con el usuario. Esto implica que sea intuitivo y capaz de explicar una variedad de comportamientos. Con esto en mente, el bloque de lógica difusa será utilizado para definir la función objetivo del algoritmo genético, que entrena la red neuronal. La ventaja de utilizar una función objetivo difusa está en que se pueden setear parámetros difusos que determinan el comportamiento que tendrá el robot. Dependiendo de estos parámetros el robot se comportará de distintas formas: moviéndose poco para ahorrar batería, moviéndose cerca del punto de partida para luego regresar a éste, maximizando el area cubierta para realizar reconocimiento de entornos, etc. De esta forma, mediante cuatro valores reales entre 0 y 1, se determinará el comportamiento del robot.

2.5. Redes neuronales SOM

Las redes SOM (Self-Organizing Map) son un modelo de redes neuronales artificiales que poseen una utilidad excepcional para visualizar datos de altísima dimensionalidad, en una representación en dos dimensiones [24]. El atractivo que poseen estas redes es que preservan las similitudes entre los datos de entrada. Estas similitudes se reflejan como grupos de elementos cercanos, cuando los datos son mapeados al espacio bidimensional, por lo que son ideales para realizar clustering (formar grupos de elementos con características similares).

2.5.1. Estructura de una red SOM

La estructura básica de las redes SOM es diferente a las redes neuronales tradicionales. La red se representa por un conjunto de n nodos ordenados en un espacio bidimensional. Cada nodo tiene asociado un vector de pesos sinápticos W de dimension q, igual a la dimension de la entrada E. Esta estructura se aprecia en la siguiente figura:

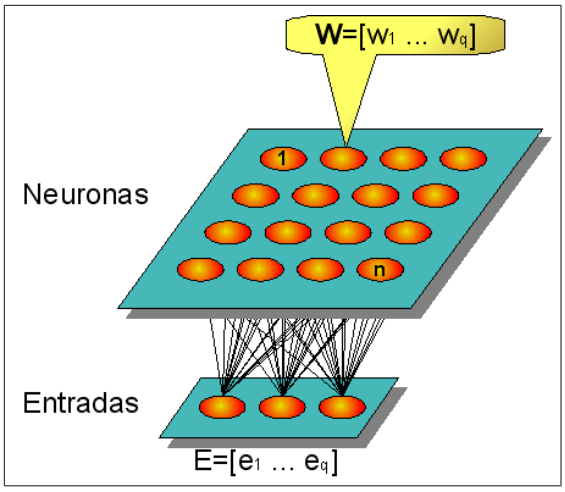


Figura 2.19: Estructura de una red SOM.

Esta red se puede interpretar como una matriz real de tamaño $n \times q$.

2.5.2. Entrenamiento

El entrenamiento consiste en ir modificando los pesos de las neuronas vecinas a una neurona ganadora [26]. La neurona ganadora es aquella que posee la mínima distancia Euclidiana con la entrada actual. La vecindad está determinada por una función de vecindad. También se define la razón de aprendizaje, que determina la velocidad con que los pesos son modificados. Tanto la razón de aprendizaje como la función de vecindad se reducen a medida que transcurren las iteraciones.

El algoritmo a seguir [25] es el siguiente:

- Escoger un conjunto de pesos iniciales de manera aleatoria.
- Presentarle a la red una entrada y obtener la distancia Euclidiana entre la entrada y cada una de las neuronas, según:

$$d_{euc} = \sum_{i=1}^{i=q} (e_i - w_i)^2 \tag{2.5.1}$$

- Buscar la neurona con la mínima distancia Euclidiana y marcarla como la neurona ganadora:

$$d_{euc.min} = min(d_{euc-j}), j = 1, 2, 3 \cdots n. \tag{2.5.2}$$

- Definir la vecindad de neuronas cuyos pesos serán modificados. La función de vecindad debe ser monotónicamente decreciente con la distancia y con el número de iteraciones. Generalmente se utiliza una función exponencial de la forma:

$$\Theta(t) = e^{-\frac{(x-x_0)^2}{\sigma}} \tag{2.5.3}$$

- Los pesos de las neuronas vecinas se modifican según:

$$w(t+1) = w(t) + L(t) \cdot \Theta(t) \cdot (e(t) - w(t)) \tag{2.5.4}$$

  para la iteración $t+1$. La constante $L(t)$ representa la taza de aprendizaje. Hay que tener presente que este valor es constante para una iteración determinada, pero va disminuyendo a medida que aumentan las iteraciones.

- Volver al segundo paso, mientras no se hayan realizado todas las iteraciones.

A medida que la función de vecindad y la razón de aprendizaje disminuyen, la variación de los pesos es menor, por lo que la red tiende a un estado estacionario. Cuando la red llega a ese estado se dice que ha aprendido.

2.5.3. Funcionamiento

El funcionamiento de esta red es el siguiente: se presenta un vector de entrada, se obtienen las distancias euclidianas entre la entrada y todos los nodos. Aquel nodo que tenga la menor distancia euclidiana es el ganador.

Cuando se presentan varias entradas consecutivas, aquellas entradas que posean características similares producirán nodos ganadores topológicamente cercanos, con lo que se pueden definir grupos de nodos ganadores. Así se pueden dividir las entradas en clusters.

La figura 2.20 ejemplifica lo anterior:

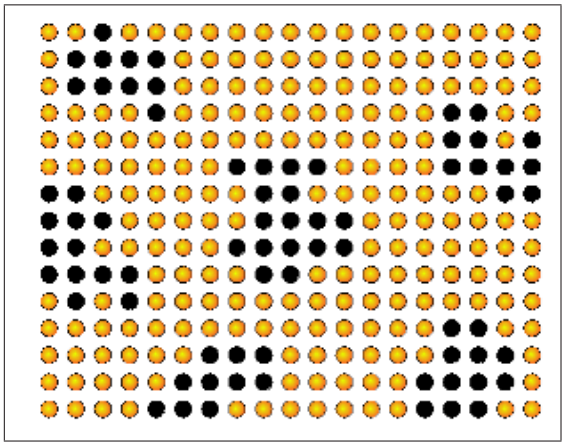


Figura 2.20: Ejemplo del funcionamiento de una red SOM.

En la figura 2.20 los nodos están representados por los círculos anaranjados y los nodos ganadores por círculos negros. Los nodos ganadores forman grupos bien definidos que representan entradas con características comunes.

2.5.4. Requerimientos del robot

El robot tiene la necesidad de reconocer en qué entorno está navegando. Para esto la red SOM será utilizada para el reconocimiento de piezas. Durante su navegación el robot intentará abarcar la mayor cantidad de area, obteniendo así datos representativos de la forma de la pieza. De esta manera se desea entrenar una red SOM con la secuencia de acciones que genere el robot al desplazarse por cada entorno. Se espera que cada pieza sea representada por una agrupación de nodos ganadores.

2.6. Action based environmental modeling

El nombre AEM proviene de las iniciales de *action-based environmental modeling*, lo que quiere decir modelamiento de entornos basado en acciones. Este procedimiento fue desarrollado por Yamada [27] para disminuir el espacio de búsqueda del problema a resolver. A continuación se explica brevemente esta técnica y el funcionamiento del bloque *AEM*.

2.6.1. Funcionamiento de AEM

Para aplicar la técnica AEM se considera lo siguiente: se posee un robot con 8 sensores de proximidad, que tienen como salida 0 si no hay obstáculo a una distancia umbral U ó 1 si hay un obstáculo a una distancia menor que dicho umbral. Como consecuencia de la entrada, el robot puede realizar una de cuatro acciones: avanzar 5[mm], girar 30º a la derecha, girar 30º a la izquierda o girar 180º a la derecha. Por lo tanto este problema se puede considerar como un mapeo de $2^8 = 256$ estados a 4 acciones, como se ve en la siguiente figura:

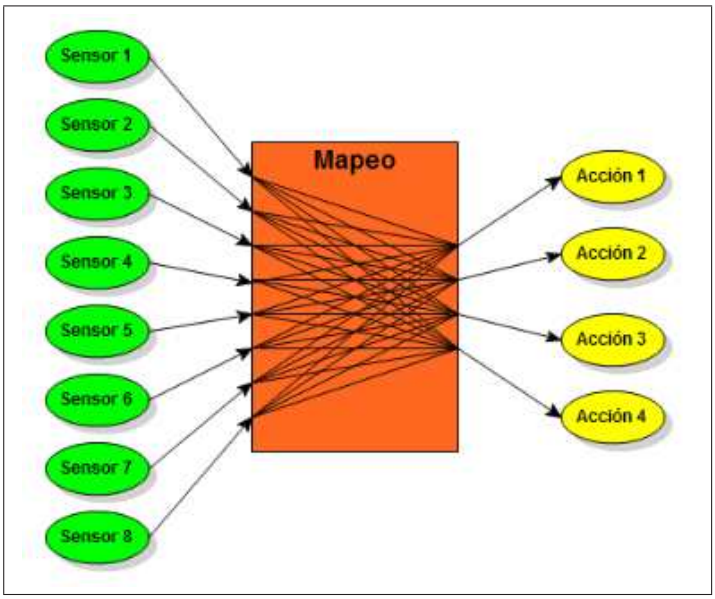


Figura 2.21: Mapeo realizado por AEM.

La complejidad de realizar el mapeo óptimo, tal que el robot cumpla una tarea específica es muy elevada. Las posibilidades son $4^{256} \approx 10^{154}$. Por otro lado, las entradas utilizadas no son discretas, sino valores reales entre 0 y 1, por lo que la complejidad del problema es aun mayor.

2.6.2. Bloque AEM

El trabajo del bloque AEM es traducir la salida de la red neuronal a una de las cuatro acciones recién mencionadas, como se muestra en la siguiente figura:

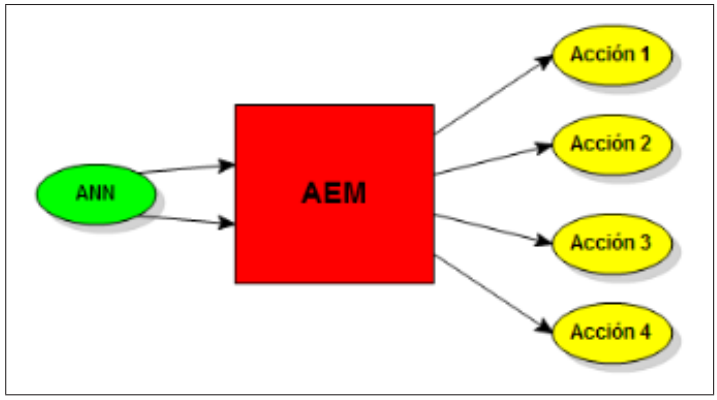


Figura 2.22: Bloque AEM.

Como se ve en la figura 2.22, la salida de la red neuronal se conecta a la entrada del bloque AEM. Este bloque, dependiendo de las salidas de la red (que en un principio controlarían directamente la velocidad de los motores) son transformadas a una de las cuatro posibles acciones. Posteriormente, cada una de estas cuatro acciones serán traducidas a señales que puedan ser utilizadas por los motores.

2.6.3. Requerimientos del robot

El robot requiere poder reconocer entornos. Por ende necesita generar la secuencia de acciones para el reconocimiento de entornos. Este bloque se encarga de generar las acciones, de manera individual.

2.7. Chain coding

Mediante el bloque AEM se codifica la salida de la red neuronal en cuatro posibles acciones, a partir de las cuales el robot podría navegar en una habitación. Estas acciones se codifican con números enteros, de la siguiente manera:

$$\begin{array}{lcl} \text{Acción1:} \quad \text{Avanza 5[mm]} & \Rightarrow & 0 \\ \text{Acción2:} \quad \text{Giro 30}^o \text{ Derecha} & \Rightarrow & 1 \\ \text{Acción3:} \quad \text{Giro 30}^o \text{ Izquierda} & \Rightarrow & 2 \\ \text{Acción4:} \quad \text{Giro 180}^o \text{ Derecha} & \Rightarrow & 3 \end{array} \tag{2.7.1}$$

El robot, a medida que navega, va almacenando la secuencia de acciones que realiza, codificada, guardándolas en un vector llamado *secuencia de acciones* ($\overrightarrow{AS}$). A continuación se presenta un ejemplo de un vector de secuencia de acciones:

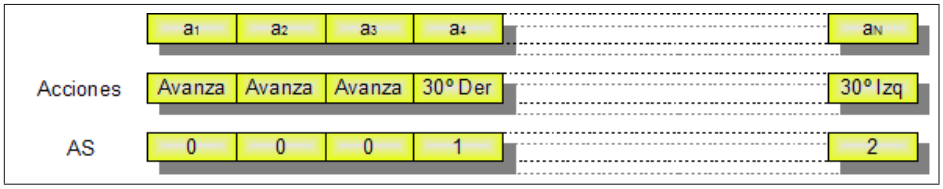


Figura 2.23: Ejemplo de vector de secuencia de acciones AS.

Los vectores $\overrightarrow{AS}$ son los que permiten el reconocimiento de entornos, ya que cada uno de ellos está asociado a las acciones realizadas en una habitación. Sin embargo, entregarle este tipo de datos a una red SOM no es muy significativo, simplemente por cómo está representada la información. Por eso es usual codificar los datos de alguna forma más adecuada. Una forma es mediante codificación de cadenas (chain coding), con lo que se obtiene un vector de salida llamado *vector de entorno* ($\overrightarrow{EV}$).

Si se tiene un vector de acciones de la forma $\overrightarrow{AS} = [a_1, a_2, \cdots a_i, \cdots a_N]$, se define un vector de entorno $\overrightarrow{EV} = [e_0, e_1, \cdots e_i, \cdots e_N]$ asociado a $\overrightarrow{AS}$, según el siguiente algoritmo:

$$\begin{array}{l} e_0 = 0 \\ \text{Si } a_i = 0 \quad \text{entonces} \quad e_i = e_{i-1} \\ \text{Si } a_i = 1 \quad \text{entonces} \quad e_i = e_{i-1} + 1 \\ \text{Si } a_i = 2 \quad \text{entonces} \quad e_i = e_{i-1} - 1 \\ \text{Si } a_i = 3 \quad \text{entonces} \quad e_i = -e_{i-1} \end{array} \tag{2.7.2}$$

De esta forma se obtienen los $\overrightarrow{EV}$, cada uno de los cuales proviene de la ejecución de un individuo en un entorno.

Para la identificación de entornos es necesario obtener un $\overrightarrow{EV}$ para cada entorno, que sea representativo del mismo. Esto quiere decir que el robot debe tener alta curiosidad al momento de generar estos vectores, de manera que abarque la mayor cantidad de area. Luego, son utilizados para entrenar la red SOM.

Capítulo 3

Desarrollo

En este capítulo se explicará cómo se fusionaron los conceptos vistos en los capítulos anteriores para generar la inteligencia del robot. Para lograr esto se presenta la figura 3.1 [28], en donde se ve la configuración de los bloques que componen la inteligencia del robot:

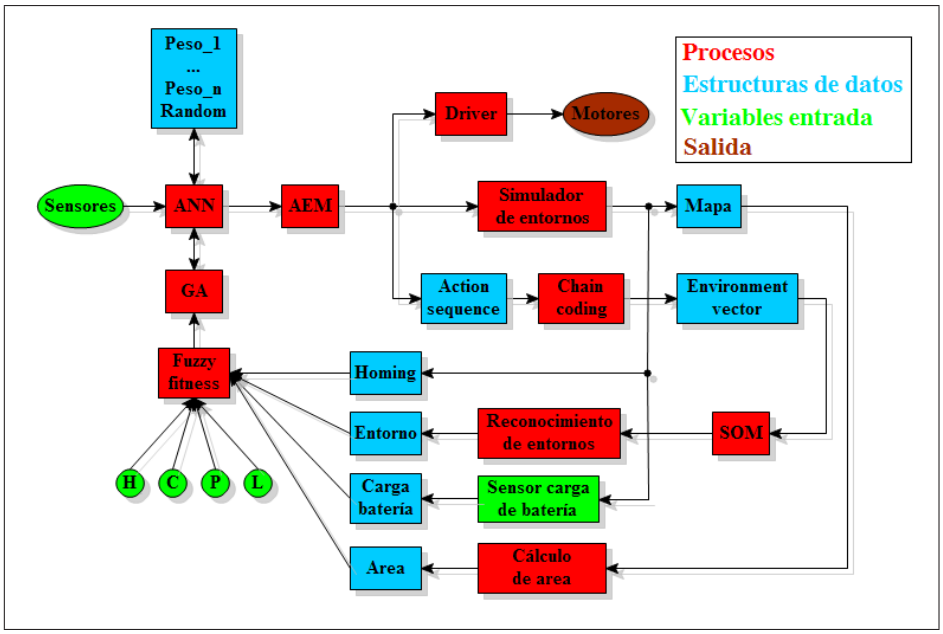


Figura 3.1: Arquitectura del sistema que estructura la inteligencia del robot

En la figura 3.1 los procesos se representan con rectángulos de color rojo, las entradas con elipses verdes, los datos con rectángulos azules y la salida con una elipse café. El proceso comienza con la captura de datos a partir de los sensores. Estos datos son procesados por la red neuronal, la cual utiliza un set de pesos neuronales determinado. Si se está en proceso de entrenamiento los pesos cargados en la red son variados con el algoritmo genético, de manera iterativa, mientras que si se está en modo de operación, los pesos no se varían. La salida de la red neuronal alimenta el bloque AEM, el que determina

qué acción debe realizar el robot. A partir de la salida del bloque AEM se confecciona la secuencia de acciones, la que es transformada en el vector de entorno, el que permite el reconocimiento de entornos junto con la red SOM. Por otro lado, se utilizan las señales de control generadas por la red neuronal para alimentar el simulador, el que se encarga de simular las acciones realizadas por el robot. A partir de este bloque se puede obtener el mapa de las zonas visitadas por el robot en el entorno. Por el momento se asume que se conoce la posición del robot, de forma de ir marcando los sectores del mapa ya visitados. Finalmente se cierra el lazo mediante cuatro conjuntos de datos, obtenidos de la simulación, del mapa, del reconocimiento de entornos y del medidor de carga de la batería. Estos datos llegan al bloque que calcula el fitness difuso, el que los procesa para definir el valor de la función objetivo que representará a cada individuo. Este fitness depende directamente del seteo de las perillas HCPL, las que se definen más adelante.

Es importante señalar que todos los procesos y datos se presentan de manera paralela. O sea, en todo momento se puede tener acceso a los datos y los procesos están en ejecución de manera simultanea. No es un diagrama de estados.

A continuación se presenta la explicación de los bloques de la figura 3.1 de manera detallada.

3.1. Procesos

3.1.1. Red neuronal artificial

La red neuronal es la parte principal de la inteligencia del robot, ya que es la que se encarga de procesar la información proveniente de los sensores para generar las señales que controlarán los motores. La red propuesta tiene la siguiente arquitectura:

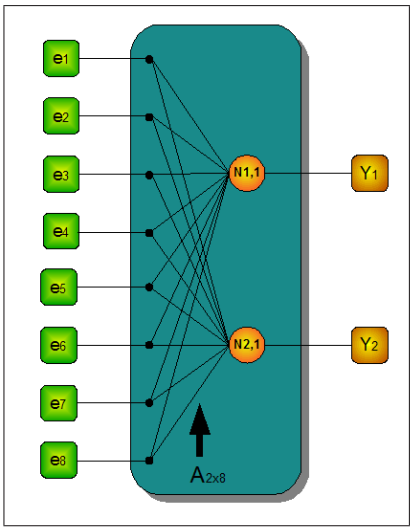


Figura 3.2: Estructura de la red neuronal del robot.

Como se aprecia en la figura 3.2 la red neuronal propuesta posee una capa de dos neuronas, ambas conectadas a todos los sensores, por lo que se puede representar por una matriz real de tamaño 2×8. Se decidió no utilizar bias, por lo que este vector se considera con valor cero. Además, la función de salida de las neuronas es una tangente hiperbólica. El sensor de batería puede o no ser considerado en la red [5].

Tanto las entradas de la red como las salidas tienen un rango de 0 a 1, para tratar a todas las señales del mismo modo.

3.1.2. Algoritmo genético

El algoritmo genético se encarga de entrenar a la red neuronal, modificando la matriz de pesos de manera iterativa hasta que la red aprenda. Cabe señalar que el algoritmo genético utilizado tiene las operaciones de mutación y elitismo. Se dice que los individuos son asexuados, ya que no se realizan cruces. Sin embargo es suficiente para entrenar la red neuronal, ya que es bastante sencilla. se considera un conjunto de matrices de pesos, cada una de las cuales representa a un individuo. Las operaciones genéticas se realizan sobre estas matrices. En la figura 3.3 se presenta el ciclo para una iteración del entrenamiento:

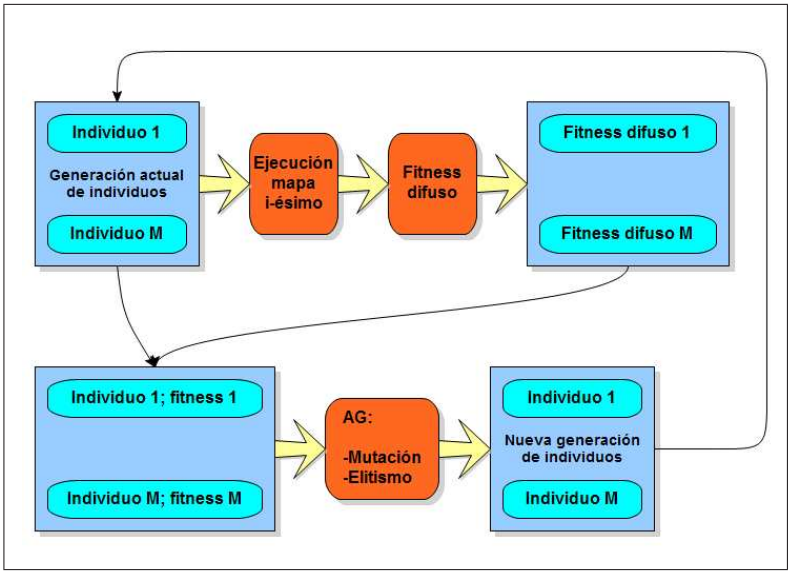


Figura 3.3: Entrenamiento de la red neuronal.

El algoritmo comienza con una población inicial escogida aleatoriamente. Cada individuo debe navegar en la habitación elegida, con lo que se obtiene un conjunto de valores que miden el desempeño de cada individuo:

Area (a) . Corresponde al area normalizada que cubre el robot, incluyendo el alcance de sus sensores. Se habla de area normalizada ya que en realidad es una relación entre el area máxima que el robot puede abarcar, respecto al area que realmente abarcó.

La mayor area que podría abarcar en teoría es cuando realiza todos sus movimientos en línea recta.

Batería (b) . Indica el porcentaje de carga de batería que queda luego de haber realizado todos los pasos [1] en una habitación. Hay que tener presente que las acciones que realiza el robot tienen distinto gasto de energía.

Homing (h) . Especifica la distancia a la que queda el robot de la vecindad de partida, al momento de finalizar los 1000 pasos.

Reconocimiento (r) . Describe si el robot fue capaz de reconocer correctamente el entorno en que se encuentra. Este parámetro se obtiene mediante la red SOM.

A partir de estos valores se obtiene el fitness difuso para cada individuo, con los que se ejecuta el algoritmo genético.

Este algoritmo posee las siguientes operaciones genéticas:

- Mutación.
- Elitismo.

A partir de estos operadores se producen las nuevas generaciones.

3.1.3. Fitness difuso

El fitness difuso posee cuatro parámetros de entrada, cada uno de los cuales representa una motivación para que el robot se comporte de una manera específica. Cada motivación está representada por un número real entre 0 y 1. Estos cuatro valores están agrupados en un vector llamado $\overrightarrow{HCPL}$. Las motivaciones utilizadas son:

Homing (H). Determina las ganas que tiene el robot de llegar de vuelta a su hogar, definido por una vecindad al punto de partida.

Curiosity (C). Corresponde a la curiosidad del robot, o sea las ganas de moverse por el entorno, tratando de abarcar la mayor area posible y descubriendo nuevos lugares.

Pressure (P). Esta motivación hace que el robot se sienta presionado a realizar su tarea lo más rápido posible para recargar la batería.

Localization (L). Representa las ganas que tiene el robot de saber en que entorno está. Para utilizar esta motivación se requiere saber si los individuos son capaces de identificar correctamente las habitaciones, por lo que se debe integrar SOM al simulador.

[1] En las simulaciones se limitó la cantidad de acciones a 1000.

Luego de haber escogido un set de motivaciones, estos valores son normalizados para que la suma de ellos de 1.

Hay que tener presente que cada una de los valores de motivación está relacionada con uno de los parámetros de desempeño enumerados en el punto anterior: Homing con Homing; Curiosity con Area; Pressure con Batería y Localization con Reconocimiento.

3.1.4. Bloque AEM

Este bloque se encarga de mapear las salidas de la red neuronal a una de las cuatro acciones posibles explicadas anteriormente (av, 30ºizq, 30ºder, 180ºder). De esta forma se pretende generar los vectores de entorno para reconocer la habitación. También se utiliza esta transformación para seguir los pasos que realizó Yamada [1.1], posibilitando que el control de los motores se realice con señales AEM, a través del mux.

3.1.5. Driver

Es necesario traducir esta información a señales que puedan controlar a los motores. La señal utilizada para realizar el control son del tipo PWM.

3.1.6. Simulador de entornos

El simulador de entornos corresponde al programa que se desea implementar. Este programa funcionará en paralelo con el robot, de manera de ir procesando la información obtenida desde los sensores y lograr el comportamiento deseado.

3.1.7. Chain coding

Este bloque se encarga de convertir la secuencia de acciones en un vector de entorno. Este proceso se ejecuta sólo cuando la secuencia de acciones está lista, o sea, cuando el robot ha finalizado su recorrido.

3.1.8. Reconocimiento de entornos mediante la red SOM

La red SOM permite que el robot reconozca en que entorno está. Para ello es necesario entrenar previamente la red. Se necesita que el robot recorra una vez cada habitación, generando en cada caso un vector de entorno distinto. Es esencial que el recorrido que realice el robot sea representativo para cada habitación. Esto quiere decir que el robot debe abarcar la mayor cantidad de area posible, por lo que en este punto la motivación *curiosidad* debe ser elevada.

Una vez obtenidos los vectores de entorno se procede a entrenar la red. En la figura 3.4 se muestra lo explicado anteriormente:

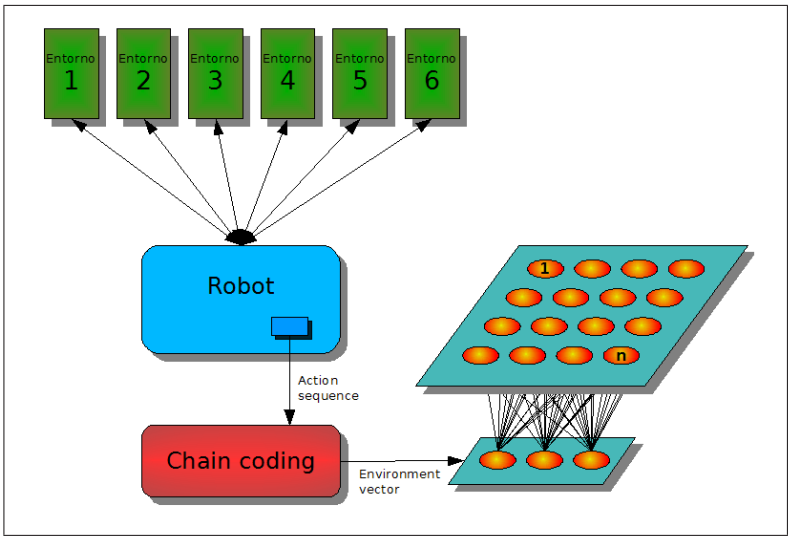


Figura 3.4: Entrenamiento de la red SOM.

La red utilizada posee 400x1 nodos de entrada y 128x1 nodos en la capa de competición. Se utilizó una función de vecindad gaussiana de radio inicial 64, 10000 iteraciones y factor de aprendizaje de 0,05.

En la etapa de reconocimiento, el robot es puesto en una habitación cualquiera. Luego de haber realizado todos los pasos, la red SOM es capaz de analizar el vector de entorno recién generado.

3.2. Datos

3.2.1. Peso_1, peso_n

Este bloque consiste en una base de datos con los sets de pesos neuronales que se han obtenido al entrenar la red de diferentes formas, cada una de ellas apropiada para una situación específica. También se cuenta con un algoritmo que genera sets de pesos aleatorios, para entrenar redes desde cero.

Cada set de pesos es una matriz de 16 valores flotantes.

3.2.2. Mapa

En este momento el mapa sólo permite calcular una aproximación del area que el robot ha abarcado en su recorrido. Para ello se divide el mapa en cuadros de igual tamaño y se

asocia cada uno a un elemento de una matriz. Luego, cuando el robot pasa por un cuadro se considera ese recuadro como descubierto, por lo que se marca como un lugar conocido (matriz con valores binarios). En la figura 3.5 se muestra un mapa de ejemplo:

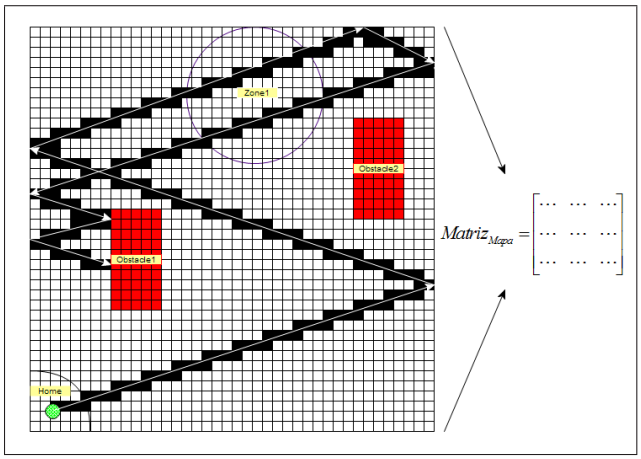


Figura 3.5: Ejemplo de un mapa y su matriz asociada.

La figura 3.5 corresponde al mapa de una habitación cuadrada, con dos obstáculos y una zona. La red neuronal ya fue entrenada, por lo que el robot se mueve de manera adecuada. A medida que el robot pasa por los recuadros, éstos van siendo marcados como reconocidos, por lo que la escritura de la matriz es on-line.

Hay que destacar la importancia del tamaño de la cuadrícula: recuadros pequeños permiten una mejor resolución del entorno, pero su número aumenta rápidamente, por lo que aumenta la cantidad de datos. Por otro lado, recuadros grandes reducen el tamaño de la matriz, pero no se logra una buena resolución del entorno. De todas formas, al ser una matriz binaria, su tamaño puede ser reducido, almacenando los datos a nivel de bits en vez de enteros o flotantes, por lo que se podría hacer una cuadrícula más fina sin tanto gasto de memoria.

3.2.3. Action sequence, environment vector

Corresponden al vector de secuencia de acciones y el vector de entorno.

3.2.4. Homing, area, carga batería, entorno

Estos datos son la información que se extrae de las simulaciones realizadas y son los que representan finalmente cuan adaptados están los individuos a su entorno, dado un set de parámetros $\overrightarrow{HCPL}$.

3.3. Salidas

3.3.1. Motores

Representa a los motores que impulsan al robot, los que son controlados mediante una señal PWM generada por el bloque *driver*. Tanto este bloque como el driver no son parte del simulador, ya que son parte de la implementación a nivel de hardware del robot.

3.4. Arquitectura de comportamientos

Los procesos y datos mostrados en la figura 3.1 son a nivel de programación. Se muestran las funciones y los datos de entrada y salida que contiene el programa, además de las relaciones que existen entre ellos. Sin embargo existe otra forma de mostrar el comportamiento del robot, a través de una red de comportamientos [28]. Este tipo de representación es más abstracta, ya que se obvían las funciones utilizadas, mostrando tareas con un nivel de abstracción más alto. También se muestran los datos de entrada y salida, además de los sensores.

Al igual que en la figura 3.1 los bloques trabajan en paralelo. En la figura 3.6 se presenta este diagrama y las explicaciones de cada uno de los bloques que lo componen:

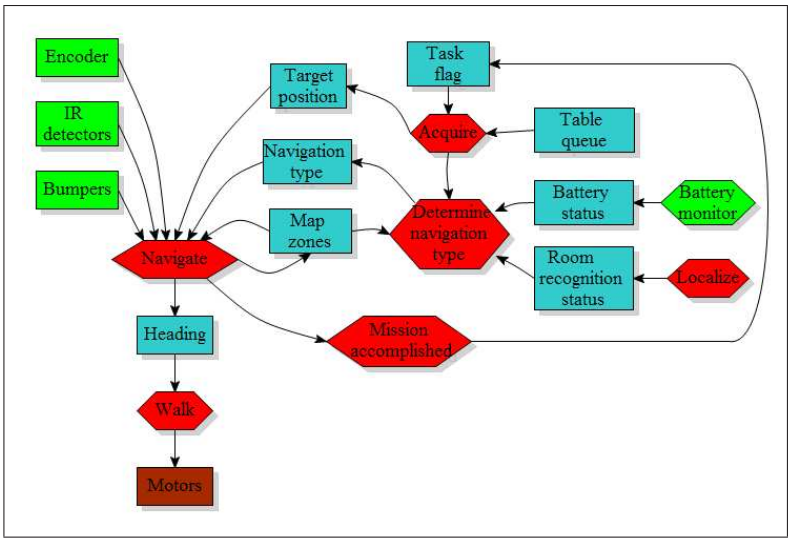


Figura 3.6: Diagrama de comportamientos del robot.

Comportamientos:

Navigate. Este es el proceso principal. Es el que determina cómo se moverá el robot, decide la dirección en que se moverá, a partir de las entradas y otros datos internos.

Walk. Es el proceso que controla los motores, decidiendo de que forma giran, para mover al robot hacia delante o hacerlo girar, a partir de la información almacenada en la cabecera. Como salida genera una señal PWM que va directamente a los motores.

Determine navigation type. Es el proceso encargado de determinar la forma de navegación del robot. Por ejemplo si el robot acaba de comenzar a desplazarse por una habitación lo más probable es que este bloque decida navegar en modo exploración, de forma de comenzar a adquirir datos del entorno.

Acquire. Este proceso es el que administra la cola de tareas que debe realizar el robot. Cuando se activa la bandera *task flag* trae al frente la siguiente tarea para que sea ejecutada. Este proceso puede establecer una coordenada de llegada para el robot (si la tarea es llegar de un punto A a un punto B) o afectar el tipo de navegación que adoptará el robot.

Mission accomplished. Este proceso determina cuando se logró la tarea actual, a partir de los datos de navegación, advirtiendo este suceso con una bandera.

Battery monitor. Corresponde a un sensor que detecta la carga de la batería. Este sensor escribe su medición en el bloque *battery status*.

Localize. Este es el proceso encargado de reconocer el entorno en que se está, con su respectivo grado de certeza.

Datos:

Encoder, IR detectors y bumpers. Estos son los sensores del robot. Representan las entradas del sistema.

Heading. Contiene la información de cual es la siguiente acción a seguir. Esta información es de alto nivel. Indica si el robot debe avanzar o girar para algún lado.

Navigation type. Este bloque contiene el tipo de navegación que se desea ejecutar. por ejemplo el robot puede estar en modo de exploración, búsqueda de una estación de recarga de batería, llegar de un punto A a un punto B, etc. A nivel de programación definirá la red neuronal a utilizar, los parámetros difusos, etc.

Target position. Si el tipo de navegación es llegar de un punto A a un punto B, este bloque contiene las coordenadas del punto B.

Map zones. A medida que el robot recorre el entorno va reconociendo zonas (por ejemplo las zonas de recarga). Este bloque contiene la información acerca de las zonas reconocidas.

Task flag. Cuando se finaliza la tara actual se levanta esta bandera para que el bloque *acquire* sepa que debe pasar a la siguiente tarea.

Task queue. Corresponde a la cola de tareas a realizar por el robot.

Battery status. Contiene la información de la carga actual de la batería, lo que afecta el comportamiento del robot.

Room recognition status. Contiene los datos acerca de las posibles habitaciones en que puede estar, con sus respectivos grados de certeza.

Salida:

Motors. Las únicas salidas del sistema son las señales de voltaje que controlan los motores. Estas señales son normalmente del tipo PWM, siendo el bloque *walk* el que las genera a partir de la cabecera (header).

3.5. Simulaciones

Utilizando el simulador YAKS con las modificaciones hechas, se realizaron numerosos experimentos para comprobar el funcionamiento del robot. Se realizaron simulaciones con distintos sets de motivaciones fuzzy para observar de qué manera se desempeñaba el robot y en distintos entornos. Además, se utilizó el programa SomPak para probar el reconocimiento de entornos.

Un resumen de todos los parámetros que definen el comportamiento del robot se muestran en la tabla 3.1:

A continuación se explicará el tipo de pruebas a realizar, la forma de entrenar el robot y los resultados obtenidos.

3.6. Criterios para realizar las pruebas

Antes de probar el robot se deben establecer cuales son los objetivos que se buscan y la manera de lograrlos. Se plantean dos objetivos principales: demostrar que el robot cambia su comportamiento al variar las motivaciones difusas de una manera consecuente al set de motivaciones y lograr un reconocimiento de entornos exitoso cuando la curiosidad es alta. Para realizar las pruebas se utilizan seis entornos cuadrados, con diferentes obstáculos y murallas, como se muestra en la figura 3.7:

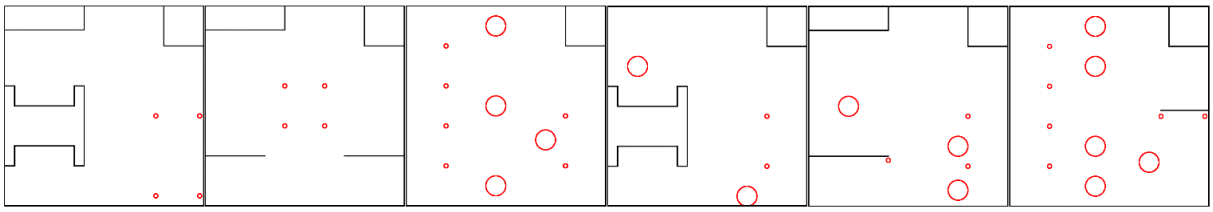


Figura 3.7: Entornos utilizados en los experimentos.

Las piezas mostradas en la figura 3.7 son cuadradas y tienen un area total de 1000×1000 unidades^2. Además se limitó la cantidad de acciones a mil.

Tabla 3.1: Parámetros del algoritmo genético y redes neuronales.

Algoritmo genético	
Tipo de población inicial	Aleatoria
Número de individuos	200
Número de generaciones	90
Mutación	1 %
Elitismo	Si
Entrenamiento	En habitación 1
Red neuronal	
Número de entradas	8
Número de capas de la red neuronal	1
Número de neuronas	2
Función de salida de las neuronas	Tangente hiperbólica
Rango de las entradas y salidas	0 a 1
Realimentaciones	No
Tipo de red	Estática
Red SOM	
Número de entradas	1000
Número de nodos en capa de competición	128×1
Función de vecindad	Gaussiana
Radio inicial de vecindad	64
Número de iteraciones	10000
Razón de aprendizaje	0.05

3.7. Entrenamiento del robot

Para entrenar la red neuronal se debe escoger un set de parámetros difusos, ya que durante el entrenamiento éstos se mantienen constantes. De los numerosos experimentos realizados se escogieron cuatro sets de motivaciones interesantes. Estas motivaciones son las que se utilizarán para entrenar la red que luego se probará en los distintos entornos. Cada set genera una red neuronal distinta. Estas redes son luego almacenadas y posteriormente probadas en cada una de las habitaciones.

Las motivaciones escogidas son:

- $\overrightarrow{HCPL_1}$=[0,00; 1,00; 0,00; 0,00]. Se espera que el robot abarque la mayor cantidad de area posible.
- $\overrightarrow{HCPL_2}$=[0,45; 0,55; 0,00; 0,00]. Se espera que el robot abarque bastante area, pero que su posición final sea cercana ala vecindad de partida.
- $\overrightarrow{HCPL_3}$=[0,25; 0,50; 0,25; 0,00]. Se espera que el robot cubra una menor area,

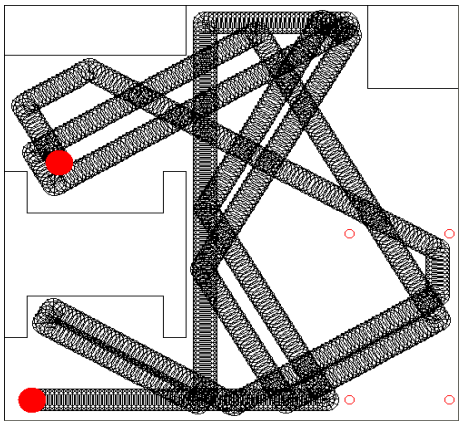
finalizando su recorrido en una posición cercana a la vecindad de partida.

- $\overrightarrow{HCPL_4}$=[0,15; 0,70; 0,15; 0,70]. Se espera que el robot se desplace de manera similar que en 3, pero que además sea capaz de reconocer entornos.

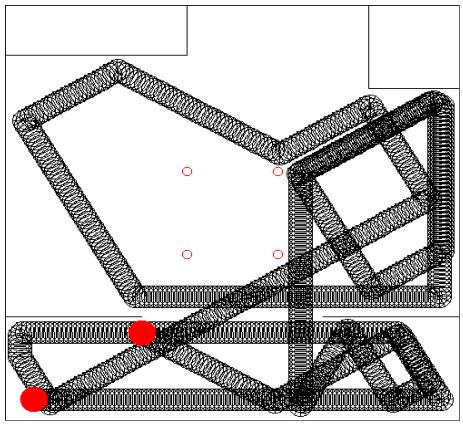
Es importante mencionar que el entrenamiento se realizó sólo en la pieza 1, aunque es posible realizar un entrenamiento secuencial, en donde se evolucionan algunas generaciones por habitación.

3.8. Resultados

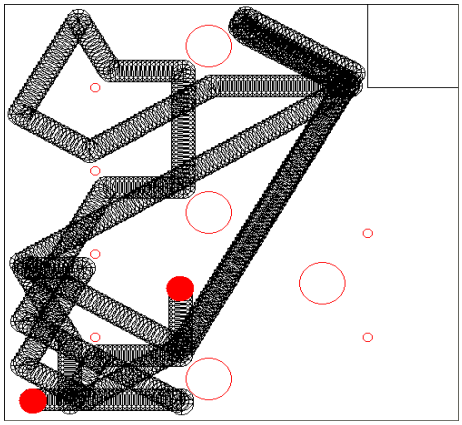
En las figuras 3.8 a 3.10 se muestran los screenshots y tablas de estadísticas obtenidos para los cuatro sets de motivaciones difusas escogidos:



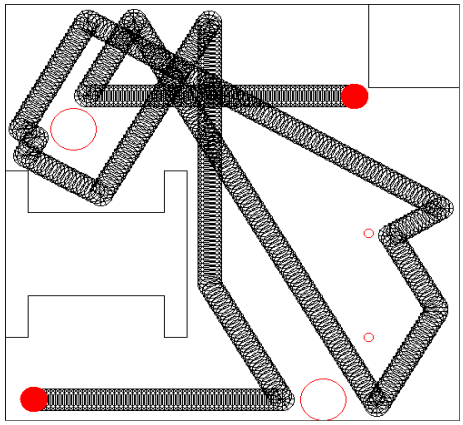
(a) Pieza 1



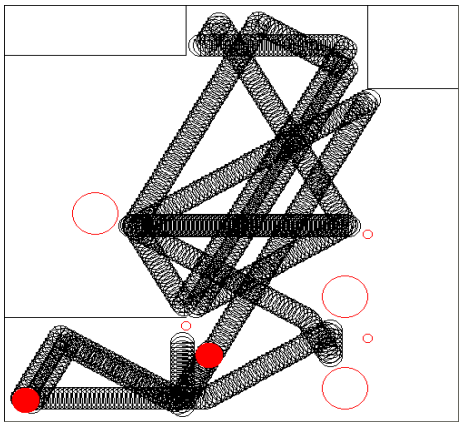
(b) Pieza 2



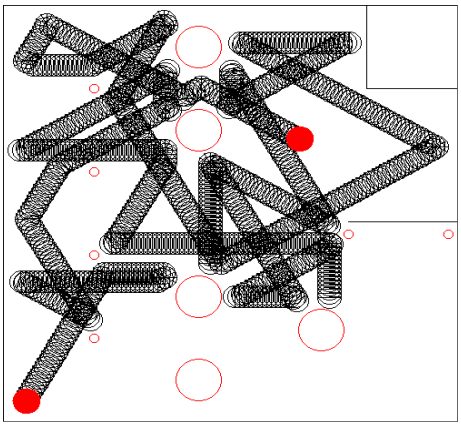
(c) Pieza 3



(d) Pieza 4

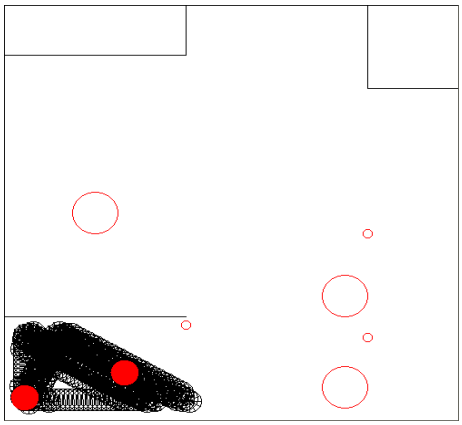


(e) Pieza 5

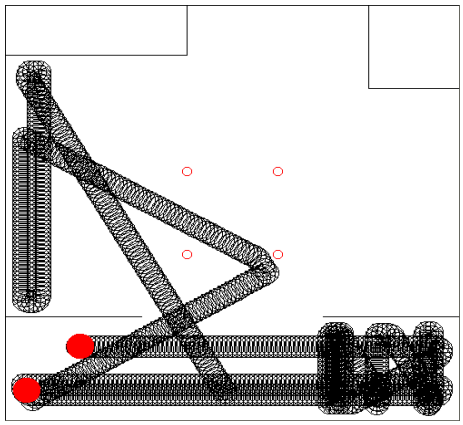


(f) Pieza 6

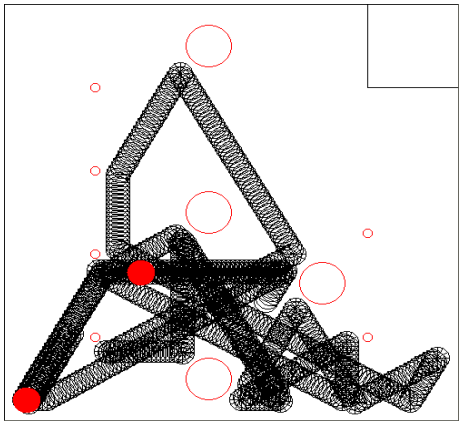
Figura 3.8: Screenshots para set de motivaciones 1, piezas 1 a 6



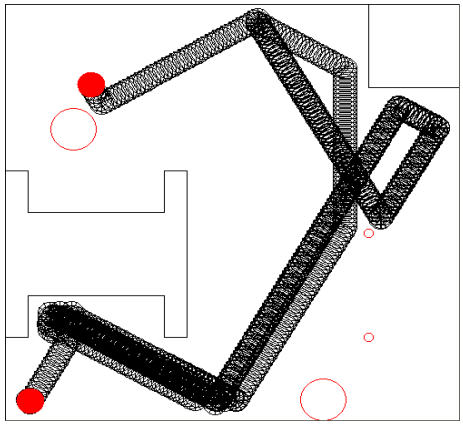
(a) Pieza 1



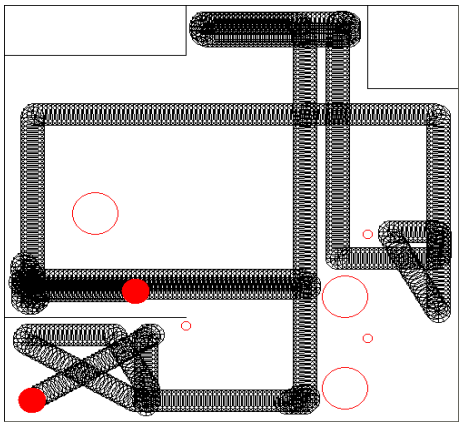
(b) Pieza 2



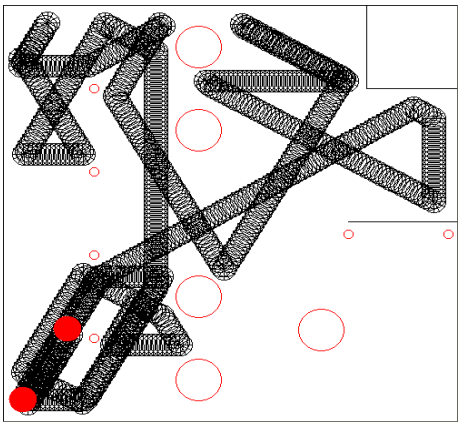
(c) Pieza 3



(d) Pieza 4

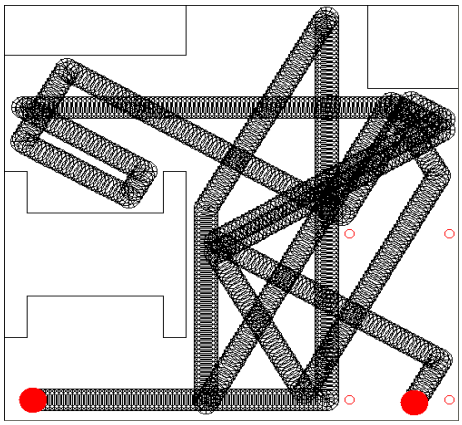


(e) Pieza 5

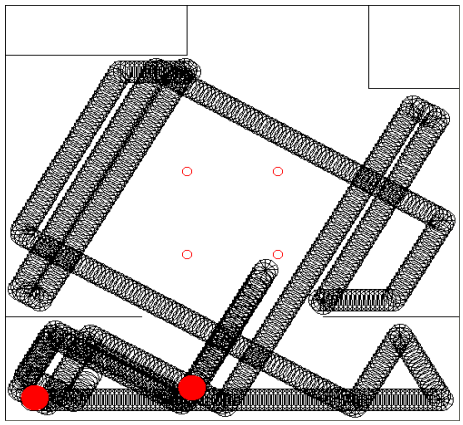


(f) Pieza 6

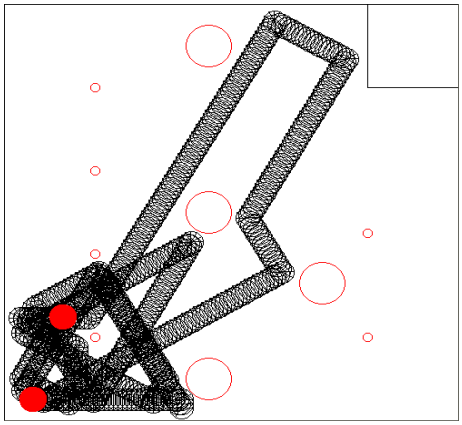
Figura 3.9: Screenshots para set de motivaciones 2, piezas 1 a 6



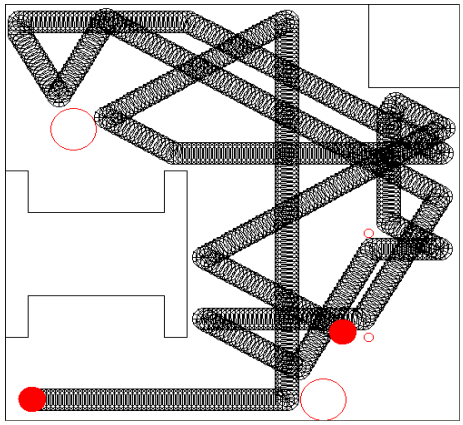
(a) Pieza 1



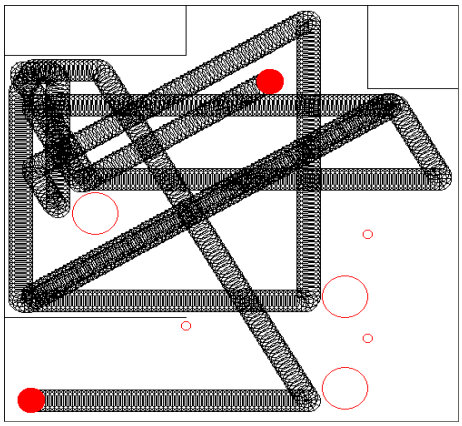
(b) Pieza 2



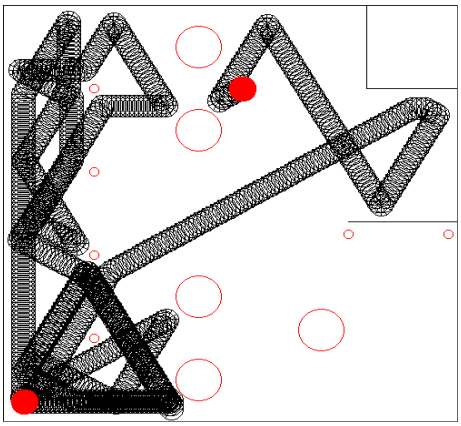
(c) Pieza 3



(d) Pieza 4



(e) Pieza 5



(f) Pieza 6

Figura 3.10: Screenshots para set de motivaciones 3, piezas 1 a 6

Tabla 3.2: Estadísticas para el set de motivaciones 1

Pieza	Fitness promedio	Homing	% Exploración	% Batería usada
Pieza1	0,895	0,623	89,500	80,820
Pieza2	0,881	0,512	88,090	81,340
Pieza3	0,873	0,369	87,330	81,050
Pieza4	0,887	0,347	88,740	80,790
Pieza5	0,884	0,558	88,450	80,790
Pieza6	0,862	0,551	86,240	81,160

Tabla 3.3: Estadísticas para el set de motivaciones 2

Pieza	Fitness promedio	Homing	% Exploración	% Batería usada
Pieza1	0,844	0,914	80,090	77,910
Pieza2	0,836	0,782	87,020	78,520
Pieza3	0,880	0,970	82,650	79,330
Pieza4	0,816	0,844	79,790	78,220
Pieza5	0,833	0,852	82,210	78,660
Pieza6	0,853	0,904	82,100	79,000

Tabla 3.4: Estadísticas para el set de motivaciones 3

Pieza	Fitness promedio	Homing	% Exploración	% Batería usada
Pieza1	0,781	0,927	88,760	79,030
Pieza2	0,786	0,833	91,900	78,990
Pieza3	0,768	0,808	89,810	79,440
Pieza4	0,746	0,773	87,350	79,210
Pieza5	0,683	0,500	84,290	76,920
Pieza6	0,722	0,719	84,870	78,170

Tabla 3.5: Estadísticas para el set de motivaciones 4

Pieza	Fitness promedio	Homing	% Exploración	% Batería usada	% Reconocimiento
Pieza1	0,759	0,576	84,960	80,590	100
Pieza2	0,442	0,578	88,790	80,730	0
Pieza3	0,433	0,508	87,240	81,050	0
Pieza4	0,439	0,588	88,110	80,260	0
Pieza5	0,774	0,561	88,160	80,900	100
Pieza6	0,419	0,550	84,130	80,990	0

Capítulo 4

Conclusiones

Se puede ver como el robot es capaz de variar su comportamiento al variar las motivaciones difusas, de la manera deseada. Por ejemplo, cuando el robot sólo está motivado con curiosidad ($\overrightarrow{HCPL_1}$), el área que cubre es mayor y los movimientos son menos predecibles. Al aumentar el homing, el robot no abarca tanta área y su posición al finalizar los 1000 pasos es más cercana a la vecindad de partida. Al aumentar la batería ($\overrightarrow{HCPL_3}$), el robot realiza un recorrido corto y muchas veces cercano a la vecindad de partida.

Hay que recordar que el robot no posee información global del entorno y que se mueve de manera refleja, según lo que censa del entorno. Esto quiere decir que su comportamiento promedio será consecuente con las motivaciones. No siempre se obtienen los mismos resultados. Por eso es que las tablas 3.2 a 3.5 tienen los valores promedio de 10 ejecuciones de la simulación, por set de motivaciones, por habitación.

De las tablas 3.2 a 3.5 se obtienen los siguientes valores promedio:

Tabla 4.1: Promedios de las tablas de resultados

Pieza	Fitness promedio	Homing	% Exploración	% Batería usada	% Reconocimiento
$[0,00; 1,00; 0,00; 0,00]$	0,880	0,493	88,058	80,992	—
$[0,45; 0,55; 0,00; 0,00]$	0,844	0,878	82,310	78,607	—
$[0,25; 0,50; 0,25; 0,00]$	0,748	0,760	87,830	78,627	—
$[0,15; 0,70; 0,15; 0,70]$	0,544	0,560	86,898	80,753	33,333

En la tabla 4.1 se ve que el fitness, al aumentar la diversidad de motivaciones, disminuye. Seguramente, cuando el robot tiene asignada una sola tarea, la realiza bien, por lo que su fitness es alto. Sin embargo, cuando se le dan múltiples tareas, las realiza todas, pero no de la mejor forma, por lo que el fitness es menor.

También se puede ver en la segunda columna que el mayor *homing* se produce cuando la motivación *homing* es la más alta ($\overrightarrow{HCPL_2}$) y es menor cuando esta motivación es la

mínima ($\overrightarrow{HCPL_1}$).

Los porcentajes de exploración son bastante similares para los cuatro casos. La mayor exploración se produce cuando sólo hay *curiosidad*, que es lo esperado. Por otro lado, la menor exploración se produce cuando el homing es alto, lo que también es coherente con lo esperado, ya que el robot tratará de mantenerse cercano a la vecindad de partida, no preocupándose de recorrer todo el entorno.

Los porcentajes de batería usada también son bastante similares entre si. El mayor gasto energético se produce cuando sólo hay *curiosidad*, lo que es razonable. Sin embargo el mínimo gasto se produce cuando el *homing* y la *presión* son los más altos (set 2 y 3, respectivamente), lo que es coherente con las motivaciones.

Por último está el porcentaje de reconocimiento. Sólo existe este valor cuando la motivación *localización* no es cero. El reconocimiento no fue muy satisfactorio, ya que sólo fue capaz de reconocer las piezas uno y cinco (un tercio del total de piezas), como se aprecia en la tabla 3.5. El problema está en el hecho de que el robot realiza el reconocimiento a través de la secuencia de acciones, las que pueden variar mucho para una pieza determinada. Sin embargo se propone una solución para mejorar el reconocimiento, la que se plantea en el capítulo 5.2.

Las aplicaciones de robots móviles autónomos son ilimitadas. Por ejemplo pueden diseñarse robots para el reconocimiento de areas geográficas, explorando areas desconocidas y realizando automáticamente mapas que puedan ser de utilidad para su propia navegación como para otras aplicaciones. También podría ser equipado con una cámara para realizar tareas de vigilancia, deambulando en lugares donde se desee saber qué pasa. Además, un robot así puede ser equipado con otros tipos de sensores, como sensores de temperatura, sensores sensibles a ciertas sustancias químicas, de radiación, etc.

Los métodos de soft computing utilizados demostraron su enorme potencia en la implementación de robots móviles autónomos. A partir de una red neuronal bastante simple se logró controlar el movimiento de dos motores de manera tal que el robot se comportara de distintas formas deseadas. Esto sólo fue posible con un entrenamiento adecuado, proporcionado por el algoritmo genético y su función objetivo, del tipo difusa.

Capítulo 5

Mejoras al comportamiento del robot

En un comienzo se decidió que el robot no posea información general del entorno y que sus movimientos sean un reflejo de lo que censa. Luego de haber logrado hacer funcionar el simulador, se plantearon nuevas formas de razonamiento, en donde el robot ya posee información de su entorno, a medida que lo censa y la almacena en un mapa interno (que no sólo sirve para calcular el area), que luego puede utilizar para mejorar su desplazamiento en la habitación.

Por ejemplo, el robot podría utilizarse para reconocer áreas extensas, de acceso difícil, etc., en las que se quieran realizar mediciones, o simplemente explorar para generar mapas. Si el robot está motivado para ser curioso, podría barrer grandes areas. Además podría ser capaz de encontrar de manera autónoma las estaciones de carga de batería, para asegurar una altísima autonomía.

Se proponen cinco mejoras para el comportamiento del robot: mejorar el mapa, realizar el reconocimiento a partir del mapa, reconocimiento de entornos en tiempo real, crear más tipos de comportamientos y utilizar una red neuronal dinámica para el control de los motores.

5.1. Mejorar el mapa

Actualmente el mapa generado consiste en una matriz binaria que representa la topografía del mapa: cero para un sector desconocido, uno para uno conocido. El mapa podría también poseer información acerca de las zonas de recarga de batería.

Eventualmente, si el robot fuera utilizado para realizar mediciones de algún tipo (concentración de gases, campos electromagnéticos, ruido, etc.) el mapa podría guardar esa información, y algoritmos especializados buscar información relevante en ellos.

5.2. Realizar el reconocimiento de entornos a partir del mapa

En este momento se utilizan los vectores de entorno para el reconocimiento de entornos [1]. Este método tiene la ventaja de no necesitar información general del entorno. Sólo necesita la secuencia de pasos codificada. Sin embargo este método es poco robusto, ya que un corrimiento (desplazamiento, offset, etc) en el vector de entorno generado produce una reducción drástica en la precisión del reconocimiento.

Una alternativa es utilizar información general del entorno, o sea el mapa. En este momento el mapa se utiliza sólo para obtener el area reconocida. Puede utilizarse este mismo mapa como entrada a una red SOM para realizar el reconocimiento.

Hay que tener presente que la información almacenada en una secuencia de acciones es la misma que la del mapa. Sólo es una forma de representación distinta. En el caso del mapa, la representación requiere de más datos, pero a su vez es una representación más robusta, ya que es del tipo topográfica (la información está directamente relacionada con la forma del mapa). Así se pretende lograr un mejor reconocimiento.

5.3. Reconocimiento de entornos en tiempo real

El reconocimiento se realiza cada vez que el robot realiza una acción, al igual que el chain coding. Así se pretende tener en todo momento una posible habitación, con un cierto grado de certeza en la respuesta dada. A medida que el robot recorre el entorno aumenta la cantidad de valores en el vector de acciones, por lo que el grado de precisión del reconocimiento aumenta y la salida tiende a un valor estable.

5.4. Más comportamientos

En la figura 3.1 se puede introducir un mux en el punto A), de tal forma que hayan otros sistemas que no sea la red neuronal que puedan controlar el movimiento del robot. Así se pueden desarrollar comportamientos dinámicos, que cambien en el tiempo según las circunstancias. De esta forma se pueden ejecutar tareas más complejas, que requieran otro tipo de comportamientos. Por ejemplo, un comportamiento podría ser el necesario para posicionar el robot en una estación de recarga, otra para llegar de un punto A a un punto B, etc. Estos comportamientos debieran ser manejados mediante una cola de misiones a cumplir.

[1] Se escogió este tipo de reconocimiento para seguir con la idea utilizada por Yamada [1]

5.5. Utilizar una red neuronal dinámica

En un comienzo, se decidió que el robot no poseyera memoria, lo que quiere decir que el robot no posee información del pasado. Sus acciones sólo son un comportamiento reflejo, dependiendo de las mediciones de sus sensores. Esto limita las posibilidades de adquirir comportamientos más complejos, en los que se utilicen datos pasados como argumentos para las acciones futuras. Una red neuronal dinámica utiliza valores presentes y pasados de las entradas, a través de un buffer, como se muestra en la figura 5.1:

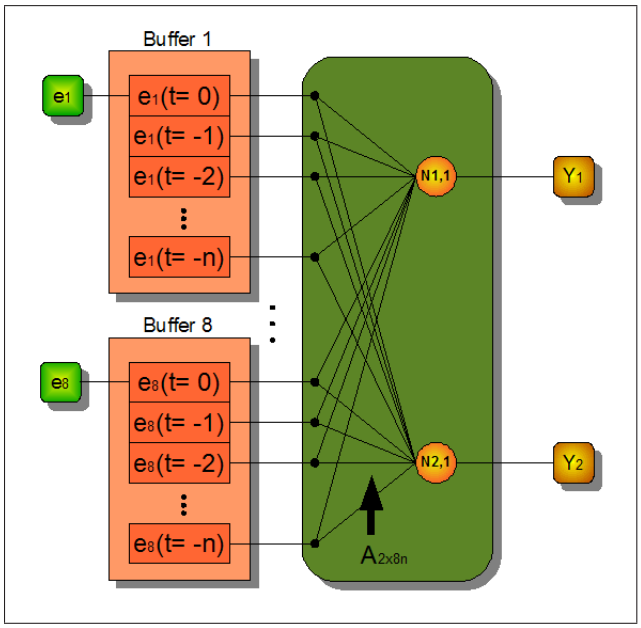


Figura 5.1: Red neuronal dinámica.

Como se ve en la figura 5.1 cada neurona de la capa de entrada está conectada al valor presente de las entradas y a una cierta cantidad de valores pasados de las mismas. Así su comportamiento puede ser más complejo.

Teniendo presente el comportamiento de estas redes, considero que puede mejorar enormemente el comportamiento del robot, pudiendo quizá diferenciar esquinas de murallas rectas y obstáculos redondos, con otras formas, etc. Incluso, si se usara un buffer suficientemente grande es posible que el robot pueda recordar distancias grandes entre obstáculos o murallas.

Por último, hay que tener presente que el tamaño de la red crece al aumentar el tamaño del buffer: al usar un buffer de tamaño n, la matriz aumenta su tamaño n veces.

Bibliografía

[1] Yamada, S. Evolutionary behavior learning for action-based environment modeling by a mobile robot. Applied Soft Computing, Volume: 5, Issue: 2, (2005), p. 245-257.

[2] Kubota N., Morioka T., Kojima F., Fukuda T.: Learning of mobile robots using perception-based genetic algorithm. Measurement, Volume 29, Number 3, April 2001, pp. 237-248(12).

[3] Izumi K., Watanabe K.: Fuzzy behavior-based control trained by module learning to acquire the adaptive behaviors of mobile robots. Mathematics and Computers in Simulation archive Volume 51, Issue 3-4, January 2000, p 233-243.

[4] Hagras H., Challaghan V., Colley M.: Learning and adaptation of an intelligent mobile robot navigator operating in unstructured environment based on a novel online Fuzzy-Genetic system. Fuzzy Sets and Systems, Volume 141, Issue 1, 1 January 2004, Pages 107-160.

[5] Arredondo, T., Freund,W., Muñoz, C., Navarro, N., and Quirós, F.: "Fuzzy Motivations for Evolutionary Behavior Learning by a Mobile Robot". In: Ali, M., Dapoigny, R.(eds): Innovations in Applied Artificial Intelligence. Lecture Notes in Artificial Intelligence, Vol. 4031. Springer-Verlag, Berlin (2006), p. 462-471.

[6] Arredondo, T., Freund,W., Muñoz, C., Navarro, N., and Quirós, F.: Real-Time Adaptive Fuzzy Motivations for Evolutionary Behavior Learning by a Mobile Robot". In: LNCS/LNAI, Vol. TBD. Springer-Verlag, Berlin (2006), p. TBD.

[7] http://diwww.epfl.ch/lami/robots/K-family/Khepera.html

[8] http://iibce.edu.uy/uas/neuronas/abc.htm

[9] http://www.gc.ssr.upm.es/inves/neural/ann2/anntutorial.html

[10] http://es.wikipedia.org/wiki/Red_neuronal_artificial

[11] http://www.monografias.com/trabajos12/redneur/redneur.shtml

[12] Brooks, R.: A Robust Layered Control System for a Mobile Robot. IEEE Journal of Robotics and Automation, Vol. RA-2, No. 1, (1986), p. 14-23.

[13] Arkin, R.: Behavior-Based Robotics. MIT Press, Cambridge, (1998), 491 pp.

[14] http://es.wikipedia.org/wiki/Evolución_biológica

[15] http://geneura.ugr.es/ jmerelo/ie/ags.htm

[16] http://www.cpdee.ufmg.br/j̃ramirez/disciplinas/otimizacao/Trab1.pdf

[17] http://www.tierradelazaro.com/mates/alggen.htm

[18] http://dis.unal.edu.co/f̃gonza/courses/2004- I/CompEvol/

[19] http://geneura.ugr.es/j̃merelo/ie/ags.htm

[20] http://www.itnuevolaredo.edu.mx/maestros/sis_com/takeyas/Tesis_Maestria /articulo.PDF

[21] http://www.mathworks.com/access/helpdesk/help/toolbox/fuzzy/fp49243.html

[22] http://www.mathworks.com/access/helpdesk/help/toolbox/fuzzy/fp49243.html

[23] http://www.lcc.uma.es/ẽva/doc/materiales/slides1.ppt

[24] http://www.dsc.ufcg.edu.br/w̃esley/GHSOM/+++dit_ijcnn2000.pdf

[25] http://www.lsi.upc.edu/ãvellido/teaching/som-gtm_web.pdf

[26] http://www.mlab.uiah.fi/t̃imo/som/thesis-som.html

[27] http://www.cis.hut.fi/research/som_lvq_pak.shtml

[28] Rotenstein A., Rothensteion A., Tsotsos J.: Mission-Directed behaviour-based robots for planetary exploration. Proc. i-SAIRAS'03, Nara, Japan, May 2003.

[29] http://freshmeat.net/projects/yaks/

[30] http://users.cs.dal.ca/t̃t/CSCI650805/papers/internalModel.pdf

Capítulo 6

Apéndice

6.1. SomPak

Para trabajar con redes SOM se utilizó el programa de código abierto SomPak [7.1], el cual genera y entrena redes SOM a partir de archivos en formato ASCII con los datos de entrenamiento. La salida que genera el programa es otro archivo, en el mismo formato, que contiene los pesos sinápticos de las neuronas ya entrenadas. La interfaz del programa es mediante una consola Unix. Cuando se hable de la carpeta de trabajo se referirá a la carpeta donde se desempaquetarán los archivos fuente y donde estarán los archivos ejecutables del programa. El procedimiento para instalar el programa y los comandos utilizados se explican a continuación.

6.1.1. Instalación del programa

Lo primero que se debe hacer es descargar el paquete *som_pak-3.1.tar* de la página:

http://www.cis.hut.fi/research/som_pak/

Para compilarlo basta situarse en la carpeta de trabajo y teclear los siguientes comandos:

- tar xovf som_pak-1.2.tar
- cd som_pak-1.2
- cp makefile.sv makefile
- make

De esta forma se crean los ejecutables del programa.

6.1.2. Comandos

Antes de utilizar el programa hay que contar con un archivo que contenga los datos de entrada. La primera línea contiene un número que indica el tamaño de los vectores de entrada. Las siguientes líneas contienen los vectores de datos. Los datos van separados entre sí por espacios y cada vector se encuentra en una línea distinta. Este archivo debe estar en la carpeta de trabajo.

Inicialización de la red

Para que el programa inicialice la red se utiliza el comando *randinit*, de la siguiente forma:

randinit -din archivo_entrada.dat -cout archivo_salida.cod -xdim 128 -ydim 1 -topol hexa -neigh gaussian -rand 4861

Las opciones utilizadas son:

-din archivo_entrada.dat . Mediante este parámetro se escoge el archivo con los datos de entrada. Este archivo, en formato ASCII, contiene en la primera línea la cantidad de datos que contiene cada vector de entrada. En las siguientes líneas y separados por un espacio simple están los números que componen a cada entrada. Cada línea contiene un vector de entrada, con la cantidad de datos especificada en la primera línea.

-cout archivo_salida.dat . Este parámetro especifica el nombre del archivo de salida. Este archivo, en formato ASCII contiene los pesos de la red SOM. Cada fila i contiene los pesos de la *i-ésima* neurona, separados por un espacio simple. Existe una fila para cada neurona. Además, en la primera línea del archivo aparece el número de entradas, la topología de la red, número de neuronas y el tipo de función de vecindad.

-xdim 128 . Este parámetro define el número de neuronas en el eje x.

-ydim 1 . Este parámetro define el número de neuronas en el eje y.

-topol hexa . Este parámetro define la topología de la red. Se puede escoger una topología rectangular (rect) o hexagonal (hexa).

-neigh gaussian . Este parámetro define la función de vecindad. Se puede escoger una función gaussiana (gaussian) o escalón (bubble).

-rand 4861 . Este parámetro sirve para escoger la semilla, para obtener los valores aleatorios que tendrán los pesos de la red cuando se inicialice.

Entrenamiento de la red

Para entrenar la red recién definida, se utiliza el comando *vsom*, de la siguiente forma:

vsom -din archivo_entrada.dat -cin archivo_salida.cod -cout archivo_salida.cod -rlen 10000 -alpha 0.05 -radius 64

Las opciones utilizadas son:

-din archivo_entrada.dat, -cout archivo_salida.cod . Estos son los archivos de entrada (datos) y salida (pesos) que utiliza el programa.

-rlen 10000 . Indica la cantidad de iteraciones que se realizarán.

-alpha 0.05 . Indica el factor de olvido de la función de vecindad.

-radius 64 . Define el radio inicial de la función de vecindad.

Pruebas

Las pruebas del programa se realizaron mediante el comando *qerror*, que permite obtener el error de cuantización de la red:

qerror -din archivo_entrada.dat -cin archivo_salida.cod

Según los experimentos realizados, errores de cuantización menores que 10 son aceptables.

Por otro lado, se obtuvieron datos reales, con los que se entrenó la red, y mediante un programa que calcula la distancia euclidiana se calcularon los nodos ganadores. Luego, con datos de prueba, se puede ver si la red aprendió correctamente a reconocer los entornos.

6.2. Simulador YAKS

Yaks es un simulador para un robot tipo Khepera. Su nombre proviene del acrónimo *Yet Another Khepera Simulator* [29]. Este simulador, de código abierto, está escrito en lenguaje C++ y se puede descargar de la página http://freshmeat.net/projects/yaks/. A continuación se verán los requerimientos para compilarlo, las variantes para compilarlo y la función de los archivos que contiene el paquete.

6.2.1. Requerimientos para compilar el programa

GTK La librería GTK permite crear interfaces gráficas para usuarios (GUI). Esta librería es usada para generar los botones, menús, etc. Las librerías y documentación se pueden encontrar en la página http://www.gtk.org.

Xlib La librería GTK y todas las salidas gráficas del simulador dependen de esta librería, la cual se encarga de traducir el flujo de datos gráficos para usar el protocolo *X Protocol*.

optional libGGI Esta librería es necesaria si se desea utilizar GGI (General Graphics Interface) como salida gráfica para el simulador.

6.2.2. Makefile

Existen distintas variantes para compilar el simulador, que difieren tanto en como se construye el lazo principal y si el programa tiene o no una interfaz gráfica. Las variantes son:

sim Si se usa **sim.c** como loop principal, el simulador se compila sin ninguna interfaz gráfica X11.

gsim El archivo **gsim.c** utiliza la función principal de **sim.c**, pero además define las librerías gráficas para GUI.

tsim Esta es una variante de gsim, en donde se genera una red de segundo orden. Esta variante fue desarrollada por el Dr. Tom Ziemke.

csim En esta variante se genera un simulador limpio (**c**lean). Esto quiere decir que no se utilizarán las clases ga y ann. La red neuronal debe ser dada de manera externa, con lo que se pueden probar redes ya entrenadas.

De estas variantes se utilizarán *sim* y *gsim* para las simulaciones. El ejecutable *sim* permitirá realizar experimentos sin interfaz gráfica [30], aumentando la velocidad de ejecución, por ejemplo, para realizar los entrenamientos o realizar estudios estadísticos. Por otro lado, el ejecutable *gsim* permite ver de manera gráfica el comportamiento del robot, con lo que se pueden realizar análisis cualitativos de su desempeño.

6.2.3. Archivos

A continuación se explica la función de los archivos fuente del paquete **yaks.tar.gz**.

ann.cc Define la clase *ann*, la que utiliza la clase *neurona*.

buildn.c Construye la red que será utilizada en la simulación.

csim.c Este archivo incluye el lazo principal, sin la red neuronal ni el algoritmo genético.

environ.cc Este archivo permite implementar los mundos, usando todas las clases de objetos y las clases del robot.

ga.cc En este archivo está implementado el algoritmo genético mediante una clase.

geom.c Aquí están implementadas las funciones geométricas utilizadas en el simulador.

gripperrobot.cc Esta es una subclase del robot, que permite utilizar el gripper (pinza).

gui.c Aquí están implementados los botones de control y menús de la interfaz gráfica.

light.cc En este archivo está implementado el objeto luz.

monitor.cc Clase que permite monitorear gráficamente las variables. Utiliza la clase *value*.

motor.cc

neuron.cc Define la clase neurona, base para generar la red neuronal.

obst.cc Implementa los objetos redondos estáticos.

param.c Este archivo es usado por la rutina principal para analizar el archivo de opciones.

real.cc Este archivo es utilizado por la rutina principal para trabajar con un robot real conectado mediante una línea serial.

robot.cc Esta es la super clase del robot.

sim.c Esta es la rutina principal estándar, independiente si se utiliza o no la interfaz gráfica.

simt.c Rutina principal estándar que implementa la red de segundo orden.

sobst.cc Implementa objetos redondos dinámicos.

test.opt Este es un archivo de opciones de ejemplo para el simulador.

value.cc Clase de valores para ser monitoreados.

wall.cc Define el objeto muralla.

zone.cc Implementa las zonas del mundo.

Para el desarrollo del trabajo hay archivos que no serán utilizados, por ejemplo, aquellos relacionados con el control de un robot real. Además, otros archivos serán modificados para insertar las líneas de código escritas.

6.3. Configuración

Para definir los parámetros de la simulación se utiliza el archivo **test.opt**, en el cual se definen todas las características del robot, del programa (tanto para cálculos como para la interfaz gráfica) y las motivaciones difusas. A continuación se muestra el contenido de este archivo, con los valores utilizados en una de las tantas simulaciones realizadas. Las líneas están numeradas para hacer referencias a ellas. Además, hay varios comentarios en el mismo archivo, de tal forma que estén claras las opciones más relevantes, muchas de las cuales se deben configurar una sola vez.

```
001 ##############################################################################
002 # This files includes the parameters that are adjustable in the simulator,#
003 # use # in the beginnig of lines to comment them out.                      #
004 # The file is not case sensitive.                                          #
005 ##############################################################################
006
007 # Evolutions parameters
008 A_#ROBOTS 1
009 #B_#ROBOTS 2
010
011 #ROBOTs
012 # Bot Code
013 #  g = gripper
014 #  b = back infrared
015 #  f? = 2,4,6 front infrared
016 #  h = hand, gripper sensor
017 #  z = ground sensor
018 #  c = compass sensor
019 #  e = energy sensor
020 #  r = rod sensor
021 #  l = light sensor
022
023 #ROBOTD1 b010101010 (0.0,0.0) # Sane test comment
024 ROBOTD1 [f6b]
025 #ROBOTD2 [f6be]
026 #ROBOTD2 [f6bl]
027
029 INDIVIDS 200 # Cantidad de individuos que componen la población.
030 EPOCHS 2
031 NR_OF_STEPS 1000 # Número de pasos que realiza el robot.
032
033 RUN_REAL 0
034 BAUD_RATE 38400
035 SERIAL_LINE /dev/ttyS0
036 RADIO_TURRETS 0
037 PARENTS 40
038 OFFSPRINGS 5
039
041 # Method            Nr
042 # None              -1  (Used for only watching a stored run)
```

```
043 # Elite             0
044 # Tournament        1
045 # Tournament+Elite  2
046
047 BIT_MUTATION 1
048 FITNESS_FUNCTION 3
049 #FFITNESS 1 paseo
050 #FFITNESS 2 zonas visitadas
051 #FFITNESS 3 fuzzy fitness with aem
052 NR_OF_INDIVIDS_TO_LOG 1
053 SAVE_WEIGHTS_EVERY_x_GENERATION 25 # Indica cada cuantas generaciones se
054 # guardarán los pesos sinápticos.
055 TEST_SAME_START_POSITION 1
056 LOG_PATH     ./log # Nombre del directorio donde se guararán los logs.
057 WORLD_PATH   ./worlds # Nombre del directorio donde se guararán los mapas.
058 CAMERA_PATH  ./cam
059 BIG_CAMERA   round.cam
060 ROBOT_CAMERA round.cam
061 SMALL_CAMERA small.cam
062 WALL_CAMERA  wall.cam
063 MOTOR_CAMERA motor.cam
064 LIGHT_CAMERA smalllight.cam
065
066 FITNESS_FILE micai3SOM.log
067 PICTURE_ROBOT robot.xpm
068 PICTURE_WORLD world.xpm
069 WEIGHT_FILE  micai3SOM   #prefix for weight files
070
071 UPDATE_DELAY 10
072
073 VERBOSE_LEVEL 1
074
075 # Graphical options
076 SCALE 0.5 # Escala interfaz gráfica.
077
078 # Simulator options
079 ##########################################################
080
081 ROD_NOISE         0
082 SENSOR_NOISE 0 # Si se desea usar ruido poner 1, sino 0.
083 NOISE_PERCENTAGE  3 # Porcentaje de ruido 0-100
084 GRIPPER_NO_NOISE  0 # If you use noise - you problary dont want to
085 # add noise to the gripper since it only
086 # return 0 or 1024
087 #GRIPPER 0 # Si se desea utilizar la pinza poner 1, sino 0.
088 MOVE_OBSTACLES    0
089
090 # Own parameters
091 # Add your own parameters here (not that it mathers where you put them)
092
093 ##########################################################
094 # This file was created by the author of the simulator #
```

```
095 # Johan Carlsson, johanc@ida.his.se, 1999              #
096 ##########################################################
097
098 SELECTION_METHOD 0 # Si se desea usar algún comportamiento almacenado
099 # poner 1, en caso contrario poner 0.
100 GENERATIONS 151 # Número de generaciones.
101 WORLD_FILE oficina2.world # Nombre del archivo que contiene el mapa.
102 START_GENERATION 150 # Generación en la que se comienza la simulación.
103 IRMA_HOMING 0.0 # Motivación Homing.
104 IRMA_CURIOSITY 0.95 # Motivación Curiosidad.
105 IRMA_BATTERY 0.0 # Motivación Batería.
106 IRMA_LOCALIZATION 0.0 # Motivación Localización.
107 IRMA_POWER_REVERSE    0.4
108 IRMA_POWER_LEFT       0.2
109 IRMA_POWER_RIGHT      0.2
110 IRMA_POWER_FORWARD    0.4
111 IRMA_POWER_FREEZE     0.1
112 IRMA_POWER_CHANGE     0.2
113 IRMA_POWER_VINIT    500.0
114 IRMA_POWER_VMAX     500.0
115
116 IRMA_ACT_FILE /home/otros/irmabot2/micai/ejecutable1/results/actseq.txt
117 # Archivo donde se guardará la secuencia de acciones realizada.
118 IRMA_LOG_FILE /home/otros/irmabot2/micai/ejecutable1/results/logfile.txt
119 # Nombre del archivo log.
```

6.3.1. Opciones generales

(101) WORLD_FILE oficina1.world

(103) IRMA_HOMING 0.0

(104) IRMA_CURIOSITY 0.95

(105) IRMA_BATTERY 0.0

(106) IRMA_LOCALIZATION 0.0

(116) IRMA_ACT_FILE /home/otros/irmabot2/micai/ejecutable1/results/actseq.txt

(118) IRMA_LOG_FILE /home/otros/irmabot2/micai/ejecutable1/results/logfile.txt

Estas opciones se configuran en un comienzo y quizá no sea necesario modificarlas. Por ejemplo, las líneas 116 y 118 seguramente no se deban modificar. La habitación es probable que se cambie en algún momento. Por otro lado, las motivaciones sólo afectarán a la red neuronal si se está realizando el entrenamiento.

6.3.2. Forma de hacer evolucionar la población

Cuando se utiliza el algoritmo genético para modificar los pesos de la red neuronal conviene usar el programa **sim**, ya que se ejecutan muchas generaciones y no es necesario ver los recorridos del robot. Los parámetros que hay que modificar en el archivo **test.opt** son:

(082) SENSOR_NOISE 0

(083) NOISE_PERCENTAGE 0

(100) GENERATIONS 150

(102) START_GENERATION -1

De esta forma se elimina el ruido en los sensores. Además se escoge que el número de generaciones a ejecutar sea 150. El -1 indica al programa que las iteraciones parten desde el comienzo, con una población aleatoria.

Para iniciar el entrenamiento se debe teclear el siguiente comando:

sim test.opt

6.3.3. Forma de correr una simulación gráfica, con una red neuronal ya entrenada

Si se desea ver el comportamiento de un robot ya entrenado gráficamente, se deben modificar los siguientes parámetros del archivo **test.opt**:

(082) SENSOR_NOISE 1

(083) NOISE_PERCENTAGE 5

(100) GENERATIONS 151

(102) START_GENERATION 150

Para las simulaciones conviene utilizar un poco de ruido para simular la realidad y para tener más diversidad en los comportamientos y trayectorias. Se escogió 5 % de ruido. Por otro lado la generación inicial es la 150, ya que es la generación más evolucionada [1]. El parámetro $GENERATIONS$ siempre debe debe tener un valor igual al parámetro $START_GENERATION + 1$.

Para iniciar la simulación se debe teclear el siguiente comando:

gsim test.opt

[1] Esto dado que se realizaron 150 generaciones.