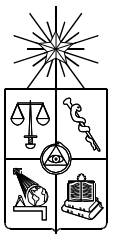

UNIVERSIDAD DE CHILE
FACULTAD DE CIENCIAS FÍSICAS Y MATEMÁTICAS
DEPARTAMENTO DE INGENIERÍA MECÁNICA

SÍNTESIS DE ALGORITMOS PARA LOCOMOCIÓN DE ROBOTS MODULARES RECONFIGURABLES

MEMORIA PARA OPTAR AL TÍTULO DE INGENIERO CIVIL MECÁNICO

FERNANDO FRANCISCO TORRES FAÚNDEZ

PROFESOR GUÍA:
JUAN CRISTÓBAL ZAGAL MONTEALEGRE

MIEMBROS DE LA COMISIÓN:
WILLIAMS CALDERÓN MUÑOZ
PABLO GUERRERO PÉREZ

SANTIAGO DE CHILE
ENERO 2012

Resumen Ejecutivo

Los Robots Modulares Reconfigurables son sistemas que han despertado un gran interés científico durante los últimos años. Ellos corresponden a un caso especial de Materia Programable, donde es posible cambiar la forma y función del sistema modular mediante, por ejemplo, la reconfiguración de sus partes.

Los sistemas robóticos modulares están constituidos por diversas unidades capaces de obtener información autónomamente, procesándola de forma distribuida o centralizada. Algunas de las interrogantes fundamentales de esta área de investigación son cómo distribuir la información y toma de decisiones entre los distintos módulos o cómo contribuir, mediante procesamiento y actuación local, al comportamiento global del sistema robótico; entre otras.

Si bien otros investigadores han desarrollado estrategias de control para generar comportamientos simples (locomoción unidireccional, formación de geometrías simples, auto reparación), no existen en la literatura métodos que permitan la generación automática de dichas reglas.

Para esto, se desarrolló un simulador acoplado a distintos métodos de aprendizaje evolutivo para la síntesis de algoritmos de locomoción para robots modulares reconfigurables genéricos. Los métodos de aprendizaje empleados fueron Algoritmos Genéticos y NEAT (*Neuro Evolution of Augmenting Topologies*).

De esta forma, con algoritmos genéticos se desarrollaron reglas de locomoción explícitas (similares a las disponibles en la literatura), mientras que con NEAT se desarrolló una novedosa propuesta, donde las reglas de locomoción están contenidas implícitamente en redes neuronales.

Finalmente, se estudió el proceso de obtención de los algoritmos de locomoción y, posteriormente, se caracterizaron las distintas soluciones obtenidas ante la variación de distintos parámetros y escenarios de ensayo.

El error asociado al aprendizaje con NEAT fue de 1.93%, mientras que con Algoritmos Genéticos se tuvo un error de 17.87%, de donde se tiene que el entrenamiento con NEAT es más repetible que con Algoritmos Genéticos. Sin embargo, los resultados finales obtenidos con Algoritmos Genéticos superaron a los obtenidos con NEAT en un 17.88%.

Para mejorar el proceso de entrenamiento fue necesario acondicionar el simulador de forma tal que cada algoritmo de locomoción obtuviese siempre el mismo resultado, reduciendo la aleatoridad.

Índice

Capítulo 1

Presentación

1.1. Introducción

La Materia Programable es un concepto que engloba a una serie de sistemas que pueden variar alguna propiedad física (densidad, rigidez, opacidad, forma, etc.) en forma inteligente, lo que implica que sus partes constituyentes son capaces de realizar cómputo. Dentro de este marco destacan los Robots Modulares Reconfigurables [1]. Estos son un caso especial de materia programable, donde sistemas robóticos son capaces de interactuar entre sí para cumplir tareas específicas.

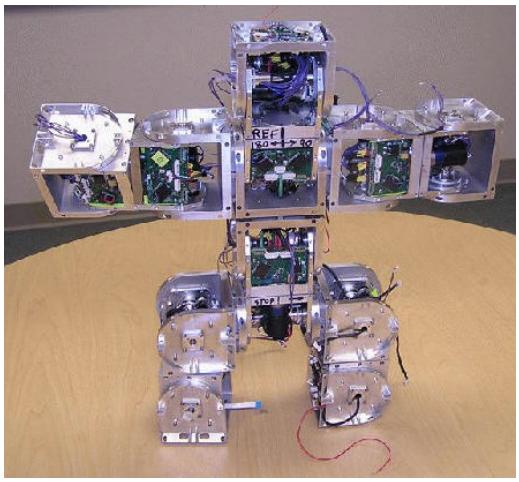

Figura 1.1: SuperBot, robot modular, multifuncional y recofigurable diseñado en el Instituto de Ciencias de la Información de la USC. Ilustración original de [2].

En el campo de los robots modulares reconfigurables es posible distinguir dos grandes grupos: (1) heterogéneos, donde los módulos de un mismo sistema robótico pueden diferenciarse por forma y/o función y (2) homogéneos, donde todos los módulos son iguales y pueden realizar las mismas acciones. En la figura 1.1 se muestra un ejemplo de robot modular homogéneo reconfigurable, cuyo

componentes son capaces de compartir información y energía a través de sus conectores [2].

Los sistemas heterogéneos tienen la ventaja de requerir estrategias de control usualmente más simples [14], en el sentido de que cada grupo de módulos cumple tareas puntuales. No obstante, el nivel de especificidad de cada módulo hace que el sistema sea vulnerable en entornos que no garanticen la disponibilidad de todos los módulos. Es aquí donde radica la ventaja de los robots homogéneos, pues el sistema se vuelve más robusto al no existir módulos irremplazables. Sin embargo, una desventaja es la complejización de los sistemas de control asociados, especialmente en el caso de control distribuido (todos los módulos ejecutan el mismo algoritmo).

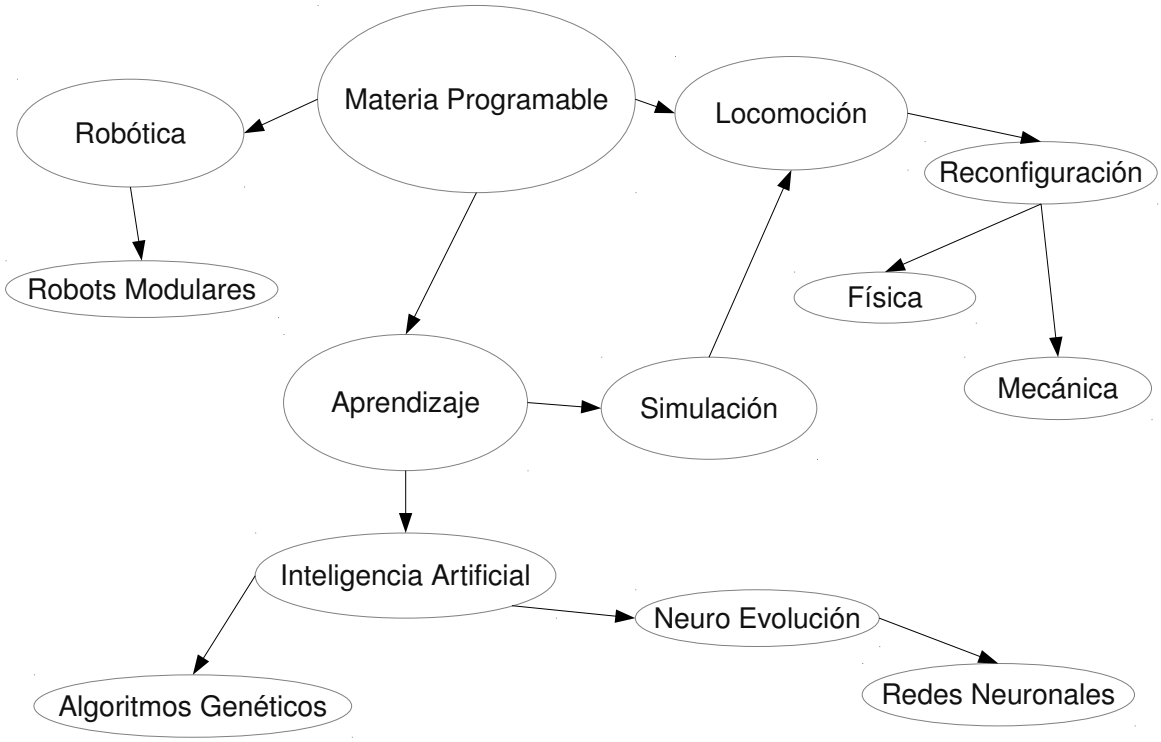

Figura 1.2: Árbol conceptual de algunos temas relacionados con la materia programable. Este trabajo de memoria está orientado al área de entrenamiento del sistema robótico, implementando distintas metodologías de aprendizaje para la generación automática de estrategias de locomoción para robots modulares reconfigurables mediante la reconexión de sus partes.

Este tipo de sistemas modulares tienen gran potencial de aplicación en como estructuras autoreparantes, sistemas autónomos de reconicimiento y vigilancia, multi-herramientas (un solo sistema es capaz de convertirse en varias herramientas), etc.

Trabajos anteriores se han enfocado en el desarrollo de algoritmos de control distribuido para tareas específicas en sistemas homogéneos genéricos. Usualmente estos estudios se han basado en técnicas de control de autómata celular [3, 4], donde un conjunto de reglas de configuración geométrica es aplicado a la vecindad de cada módulo, determinando de esta forma la acción que se realizará.

El trabajo está organizado de la siguiente forma: a continuación se presentan los objetivos planteados, para luego presentar los antecedentes relevantes para el trabajo de memoria, donde se revisan conceptos tales como materia programable, robots modulares, autómatas celulares y algoritmos evolutivos. Se continúa con la presentación de la metodología de trabajo, luego se exponen los resultados de las experiencias realizadas, seguidos del análisis de estos y las conclusiones finales.

1.2. Objetivos

El objetivo de esta memoria es elaborar un método para la síntesis automática de algoritmos de control descentralizado para robots modulares reconfigurables homogéneos genéricos en problemas de locomoción mediante el uso de Algoritmos Evolutivos. Dado un problema de locomoción, se procederá a sintetizar y evaluar, mediante simulaciones, un conjunto de estrategias de control para la reconfiguración local que permitan a un sistema robótico modular cambiar su forma para moverse en un escenario dado. Finalmente se caracterizarán los resultados obtenidos.

A continuación se presentan los objetivos planteados para el desarrollo del presente trabajo de memoria:

1.2.1. Objetivo General

El objetivo general es sintetizar y caracterizar algoritmos para el control descentralizado de robots modulares reconfigurables genéricos mediante el uso de Algoritmos Evolutivos en problemas de locomoción.

1.2.2. Objetivos Específicos

Para alcanzar el objetivo general se desarrollarán los siguientes objetivos específicos, utilizando como plataforma de programación el software Matlab R2010a:

- Desarrollar un simulador para visualizar las estrategias de control y medir parámetros de interés para la síntesis de estos.
- Implementar sistemas de algoritmos genéticos y neuro-evolución como mecanismos de aprendizaje y síntesis.
- Caracterizar los resultados obtenidos con los algoritmos evolutivos (algoritmos genéticos y neuroevolución).

Capítulo 2

Antecedentes

La mayoría de los experimentos empíricos en el área de materia programable se han concentrado en controlar la forma final de un cuerpo mediante el ensamblaje autónomo o supervisado de estructuras modulares estáticas. En gran medida se han dejado de lado las experiencias conducentes a generar locomoción de una estructura como resultado de la reconfiguración de sus partes.

En el ámbito de la simulación se ha explorado exitosamente el control de sistemas modulares mediante algoritmos descentralizados. En los últimos años se han desarrollado algoritmos genéricos que permiten el control para una amplia gama de robots modulares reconfigurables, basándose en técnicas de autómata celular. Estos algoritmos genéricos suponen pocas condiciones respecto de capacidad real de locomoción local de los módulos.

En estos algoritmos genéricos los módulos del sistema toman decisiones de acción basándose en reglas geométricas que se aplican a la vecindad del módulo activo. Esto hace que los algoritmos se independicen de la plataforma de implementación.

La amplia gama de métodos de aprendizaje disponibles en los algoritmos evolutivos dan pie a pensar en la posibilidad de sintetizar automáticamente reglas de control descentralizado para sistemas modulares, eventualmente tan complejos como los necesarios en el control de los robots modulares reconfigurables homogéneos.

A continuación se presenta una revisión de los temas centrales pertinentes al presente trabajo de memoria.

2.1. Materia Programable

La materia programable se caracteriza por la habilidad de poder modificar alguna propiedad de forma inteligente, lo que significa que sus elementos constituyentes deben manipular información en algún nivel. El procesamiento de esta información puede ser llevada a cabo por el mismo sistema, de forma autónoma o por mecanismos externos.

Muchos estudios se han orientado principalmente al problema de ensamblaje de estructuras estáticas en distintos medios y con distintas estrategias. Ejemplo de esto es el trabajo realizado en [15], donde los módulos consisten en una colección de cubos, cuyas caras contienen válvulas neumáticas que se abren al entrar en contacto con otro módulo. El anclaje de las distintas partes para formar una estructura es posible con una fuente de vacío y la propagación de la succión a través de la estructura mediante el uso de las válvulas. En [10, 11, 12, 13] se emplea un concepto similar al anterior, pero utilizando sumideros en un fluido para aproximar los distintos módulos a sus posiciones finales, donde dispositivos eléctricos o mecánicos consolidan las uniones.

En los casos mostrados en [11, 13] se puede reconocer una estrategia centralizada de ensamblaje, pues es un sistema auxiliar quien guía a los módulos a su posición final mediante el uso de fuentes y sumideros en una cámara 2D acondicionada (figura 2.1). Por otra parte, en [10, 12] son los mismos módulos quienes actúan como sumideros. De esta forma un módulo ya ensamblado se activa y atrae a uno libre al punto de anclaje (figura 2.2). Esto también ha abierto lineas de investigación orientadas al estudio de la física y geometrías para optimizar el procedimiento de ensamblaje.

Sin embargo, los sistemas generados bajo este enfoque son funcionales en la medida de que su forma final lo sea. ¿Qué pasaría si estos sistemas tuvieran la libertad de reconfigurarse dinámicamente, pudiendo desplazarse o cambiar su forma continuamente? Pues esto puede lograrse con los robots modulares reconfigurables.

2.2. Robots Modulares Reconfigurables

Los robots modulares reconfigurables son sistemas cooperativos que comparten información para llevar a cabo tareas específicas (e.g. vigilancia, redes sensoriales móviles, etc). Una tarea interesante que se espera de este tipo de sistemas es la reconfiguración en formas arbitrarias, en las que el sistema no necesariamente se mantiene conexo. En [7] se presenta el algoritmo HDM[1] para resolver el problema. Este método supone que cada elemento tiene conocimiento de su ubicación

[1]Siglas para Descomposición Jerárquica de la Mediana (*Hierachical Median Decomposition*).

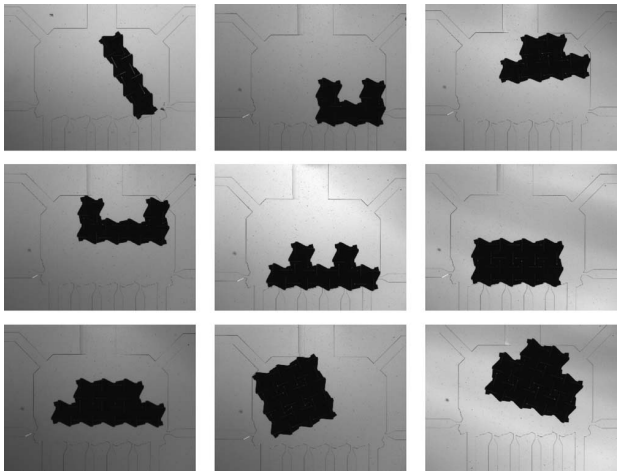

Figura 2.1: Distintas configuraciones finales con sistema de control auxiliar centralizado que maneja las fuentes y sumideros. Los módulos consisten en piezas de silicona de 500x500 μm^2, con un espesor de 30 μm. Estas piezas son llevadas a su posición final gracias al sistema auxiliar, donde se anclan mediante cerrojos pasivos. Figura original de [11].

globlal dentro del sistema, con lo que se genera una biyección entre las posiciones iniciales y finales de cada elemento, tras lo cual se planea el movimiento del grupo (figura 2.3). Sin embargo, esto requiere un alto consumo de memoria que crece con la cantidad de módulos del sistema. También requiere algún mecanismo centralizado de control que, al menos, determine las posiciones de los módulos. Esto último desmedra considerablemente la autonomía del sistema global.

En [6] se presenta un método para la planificación de la reconfiguración de robots modulares. Este método utiliza métricas que miden la similaridad entre distintas formas del espacio Euclideano junto a un algoritmo que es capaz de bisectar estas formas. Con esto, se puede generar la secuencia de movimientos que minimiza alguna función de costo (e.g. energía o cantidad de pasos) para alcanzar una configuración final.

Existen otros enfoques para la reconfiguración, donde sistemas modulares heterogéneos utilizan algoritmos de control local para reconfigurar al conjunto [14]. Este tipo de control tiende a simplificar el tipo de algoritmo que ejecuta cada módulo (figura 2.4), pero limita el uso de este tipo de sistemas a entornos que garanticen la disponibilidad de todos los componentes. Esta condición se puede remediar con el uso de sistemas homogéneos como alternativa para trabajar autónomamente en entornos no controlados, dándole más robustez al conjunto. Una extensión de esto son los métodos de reconfiguración basados en las técnicas de control de autómatas celular, en el que la acción de cada módulo se decide por la interacción de este con su vecindad [3, 4, 8, 27].

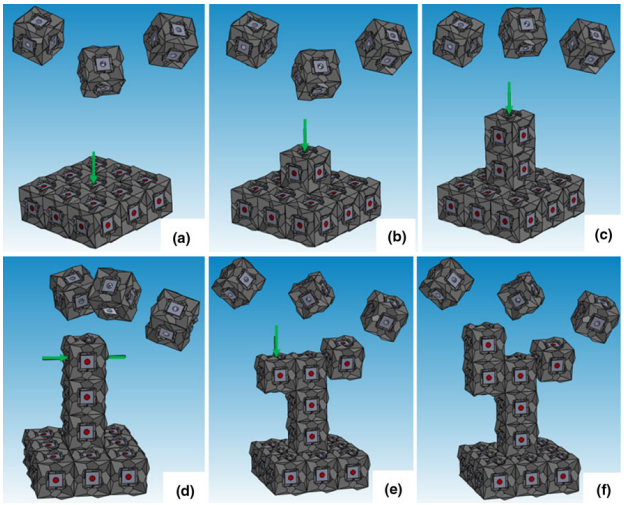

Figura 2.2: Sistema robótico modular 3D de actuación distribuida. La geometría de módulos ha sido optimizada para mejorar el proceso de ensamblaje. Ilustración original de [10].

Un ejemplo de implementación de reglas de control local (o descentralizado) en robots modulares reconfigurables es la plataforma TeleCube[2] [8]. Este sistema consiste en un conjunto de cubos, cuyas caras se extienden telescópicamente en un factor de 2× en relación al tamaño del cubo y que se pueden anclar a las caras de los módulos vecinos. Los movimientos de este sistema están restringidos a las direcciones en las que apuntan las caras del cubo. Para generar el movimiento se consideran varios modos de movimiento, tanto cooperativos como individuales, combinados con la propagación de información y acción local determinadas por reglas simples.

Algunos autores han concetrado sus esfuerzos en el desarrollo de algoritmos para control descentralizados genéricos, que puedan ser usados en una amplia gama de robots modulares reconfigurables, reduciendo los supuestos sobre la capacidad de locomoción de los módulos en el sistema y abstrayéndolos de la plataforma física de implemetación [3, 4, 17, 20]. Estos supuestos consideran que cada módulo es capaz de desplazarse linealmente sobre el sistema robótico y que es capaz de hacer transiciones convexas y cóncavas a un plano diferente (figura 2.5). La transición cóncava no es señalada como un movimiento propiamente tal, sino que como un cambio del punto de anclaje respecto al resto del sistema. Ejemplos de plataformas que cumplen con estos requisitos son el robot TeleCube, Proteo [23] y los módulos presentados en [5].

[2]Desarrollado en el Xerox Palo Alto Research Center.

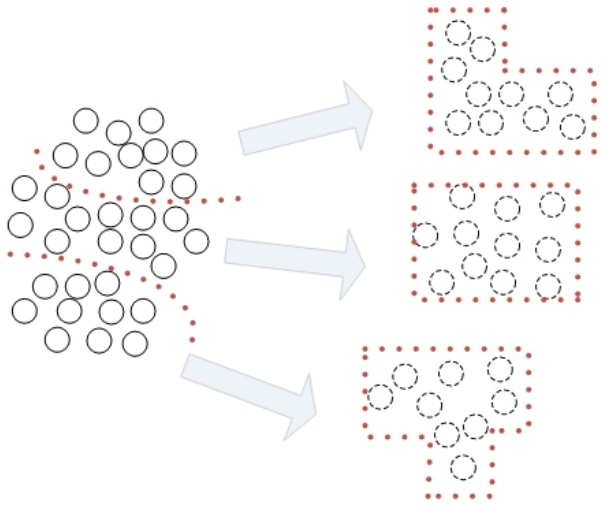

Figura 2.3: Escenario hipotético de distribución inconexa en redes móviles sensoriales. El sistema, en su estado inicial, es reconocido por el mecanismo auxiliar de localización, tras lo cual se genera la biyección entre la posición inicial y final de cada módulo. Ilustración original de [7].

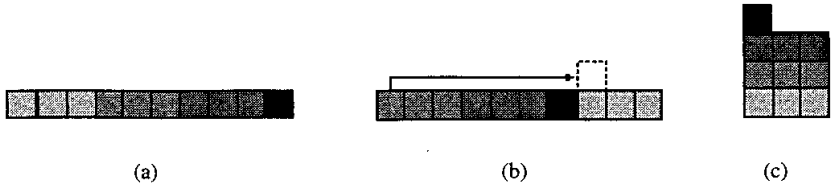

Figura 2.4: Secuencia de estratificación en un sistema modular heterogéneo, donde el tono de color indica el tipo de módulo. Ilustración original de [14].

Basados en las hipótesis anteriores, se han deducido conjuntos de reglas para producir un movimiento unidirecccional con o sin obstáculos en un sistema bidimensional. Estas reglas condensan una serie de configuraciones geométricas similares para las cuales se espera una acción dada, utilizando reglas del tipo de autómatas celulares (figura 2.6) [4]. Además se presentan las demostraciones de los lemas que, para una configuración inicial rectangular: (1) estas reglas siempre pueden aplicarse, (2) el movimiento puede lograrse con cualquier secuencia de activación continua de las reglas y (3) ninguna secuencia de evaluación de las reglas dejará inconexo al sistema. Aunque estrictamente el último lema no es correcto, la implementación del modo de activación D_1 (figura 2.7) garantiza que este se cumple, asegurando que ningún módulo puede activarse más de dos veces seguidas.

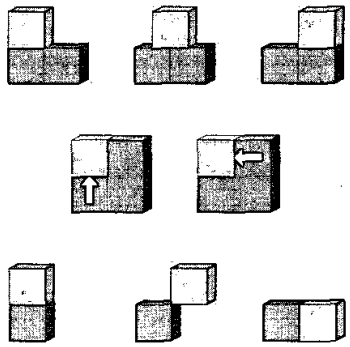

Figura 2.5: Movimientos básicos de un módulo genérico. (arriba) Traslación lineal sobre un plano de módulos, (centro) transición cóncava y (abajo) transición convexa. Figura original de [3].

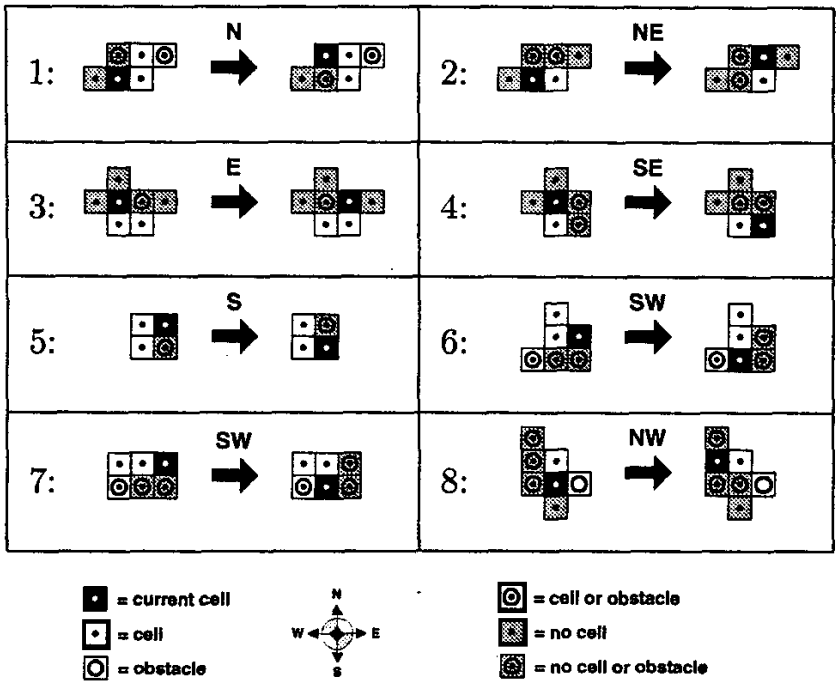

Figura 2.6: Conjunto de reglas geométricas locales para generar moviento hacia la derecha con obstáculos. Figura original de [4].

También se han desarrollado conjuntos de reglas genéricas descentralizadas que permiten la reconfiguración desde un sistema modular completamente extendido a un cubo y la reparación de agujeros (figura 2.8).

Algorithm 1 D_1 module activation model

```
1: while true do
2:     Generate module list
3:     while list not empty do
4:         Choose module at random from list
5:         for rule in Rules do
6:             if rule applies to module then
7:                 use it and break
8:             end if
9:         end for
10:        remove module from list
11:    end while
12: end while
```

Algorithm 2 D_∞ module activation model

```
1: while true do
2:     Choose module at random
3:     for rule in Rules do
4:         if rule applies to module then
5:             use it and break
6:         end if
7:     end for
8: end while
```

Figura 2.7: Algoritmos para la activación de módulos genéricos. El algoritmo D_1 es semi-aleatorio, pues garantiza que una celda no puede activarse más de dos veces seguidas, lo que sucedería en el caso de que un mismo módulo fuese el último en ser seleccionado en una lista y el primero en ser seleccionado en la lista siguiente. Por otra parte, el modelo D_∞ consiste en una activación completamente aleatoria de los módulos. Un modelo D_0 correspondería a una activación completamente secuencial de los módulos.

En general, estos conjuntos de reglas se complejizan rápidamente con la dificultad de las tareas requeridas, por lo que es interesante automatizar la generación de estos algoritmos de control descentralizado, pudiendo aspirar a reglas para control en sistemas tridimensionales y/o comportamientos de alta complejidad.

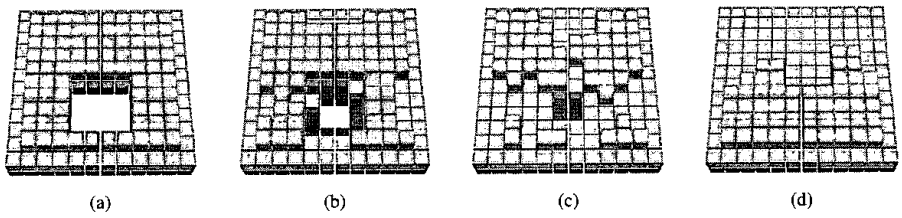

Figura 2.8: Cuatro instantes de la aplicación del algoritmo de reparación de agujeros con reglas geométricas basadas en autómata celular. Figura original de [3].

2.3. Autómatas Celulares

Es relativamente normal ver sistemas en la naturaleza cuyos comportamientos son complejos, pero cuyos componentes son simples. Estos comportamientos son generados por la cooperación de los elementos básicos. Los autómatas celulares son sistemas matemáticos construidos con varios

componentes simples e idénticos, cuyo trabajo en conjunto es capaz de lograr resultados de extrema complejidad [27].

Un autómata celular uni-dimensional consiste de una linea de espacios, cada uno conteniendo un valor 0 o 1 (o, en general, un valor de $0, ..., k-1$). El valor a_i de cada sitio se actualiza en pasos temporales discretos, de acuerdo a una única regla determinística que depende del estado de los vecinos en torno a un radio r:

$$a_i^{(t+1)} = \phi[a_{i-r}^{(t)}, a_{i-r+1}^{(t)}, ..., a_i^{(t)}, ..., a_{i+r-1}^{(t)}, a_{i+r}^{(t)}] \tag{2.1}$$

Incluso con valores de $k = 2$ y $r = 1$, el comportamiento global de este simple sistema puede ser altamente complejo. Algunas reglas locales ϕ pueden generar comportamientos simples, mientras otras pueden generar complejos patrones. Estudios empíricos han demostrado que este tipo de autómatas celulares desarrollan básicamente 4 tipos de patrones cualitativos [27]:

1. Patrones que se desvanecen.
2. Evolucionan hasta llegar a un tamaño fijo.
3. Crecen indefinidamente a velocidad constante.
4. Crecen y se contraen irregularmente.

Usualmente, los modelos matemáticos de sistemas naturales se basan en ecuaciones diferenciales que describen comportamientos suaves de un parámetro en función de otros. Los autómatas complementan este punto de vista, añadiendo una descripción evolutiva discreta de varios componentes (idénticos entre sí). Los modelos basados en autómatas celulares suelen ser más adecuados para la descripción de sistemas físicos altamente no-lineales, como también en sistemas químicos y biológicos, donde la desencadenación de los fenómenos depende de algún valor umbral. Los autómatas celulares también son modelos adecuados cuando los efectos de la inhibición del creciemiento son importantes.

Como ejemplo, los autómatas celulares pueden proveernos de modelos de creciemiento de cristales dendríticos, como los copos de nieve [37]. Empezando de una sola semilla, se agregan sitios que representan la fase sólida de acuerdo a una regla bi-dimensional que considera la inhibición del crecimiento en torno a sitios recién creados, de donde resulta un crecimiento tipo fractal. Otro ejemplo son los sistemas de fluidos turbulentos, donde se puede modelar la interacción local entre

vortices discretos.

Los auḿatas celulares pueden servir en una amplia variedad de sistemas biológicos. En particular, pueden servir como modelos de formación de patrones. Por ejemplo, se han desarrollado modelos del creciemiento de algunos patrones de pigmentación [38].

En general, hay dos formas de concebir a los autómatas celulares. La primera es como sistemas dinámicos discretos, o idealizaciones discretas de ecuaciones diferenciales. La segunda forma es considerarlos como sistemas de procesamiento de información, o como computadoras simples de procesamiento paralelo. La información estaría representada por el estado inicial, siendo esta procesada por la evolución del autómata celular.

Quizás la aplicación de autómatas celulares más difundida es el *Juego de la Vida* [40, 39]. El juego de la vida es en realidad un juego que no requiere jugadores, pues la evolución del sistema está determinada sólo por el estado inicial de la malla y no necesita ninguna entrada de datos posterior. Este juego es capaz de generar una serie de patrones dinámicos o estáticos (figura 2.9).

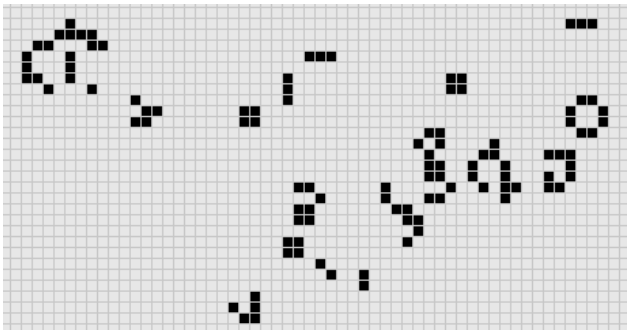

Figura 2.9: Instantánea del Juego de la Vida, donde se reconocen algunos de los patrones recurrentes del juego.

Una forma de potenciar las capacidades de los autómatas celulares es acoplarlos con sistemas de entrenamiento automatizados como, por ejemplo, los algoritmos evolutivos. En [19] se implementó una plataforma de entrenamiento para desarrollar reglas de autómata celular para resolver el *Majority Problem*. Este es un problema de clasificación, en el que el objetivo es desarrollar, desde una población inicial uni-dimensional, binaria y aleatoria, una población final uniforme. El problema se considera resuelto correctamente si el estado final de la población coincide con el estado con mayor presencia en la población inicial (figura 2.10).

La herramienta de aprendizaje usada para entrenar al autómata celular es la programación genética estandar [41]. En la experiencia se trabajó sobre una población de 149 estados binarios, para la cual varios autores ya habian propuesto reglas.

Figura 2.10: Comportamiento espacio-tiempo de la mejor solución rondas de aprendizaje con programación genética. La población inicial corresponde a la fila superior, y la población se desarrolla, según las reglas evolucionadas, hacia abajo. Fuente [19].

Al momento de la experiencia, la mejor solución disponible habia sido desarrollada en 1995 por Rajarshi Das, que consistía en modificaciones a las reglas propuestas en 1978 por Gacs-Kurdyumov-Levin. Esta solución tiene una tasa de éxito de 82.178 %, ensayadas 10^7 veces sobre una población de 149 estados binarios aleatorios. La mejor solución, obtenida en 1996 con programación genética, obtuvo una precisión del 82.362 %, medidos en 10^7 poblaciones iniciales aleatorias de 149 estados binarios.

Estos resultados reafirman la alternativa de utilizar algoritmos evolutivos para el desarrollo de reglas de autómata celular, puesto que se obtuvo una mayor precisión que con las desarrolladas por humanos.

2.4. Algoritmos Evolutivos

Una de las carácteristicas más admirables de la naturaleza es la existencia de organismos que se adaptan constantemente en los entornos más dispares, que a menudo llegan a ser extremadamente hostiles. Esto implica que ciertas formas de vida se extinguen, mientras que otras sobreviven

evolucionando en estos escenarios dinámicos. También es sorprendente que estos organismos no realizan ningún esfuerzo explícito en adaptarse, sino que este proceso es llevado por un mecanismo intangible que conocemos por evolución natural.

La comunidad de optimización ha tomado este ejemplo para desarrollar varias técnicas que pueden agruparse bajo el concepto de *Algoritmos Evolutivos*. De hecho, esta es un área que en sí está constantemente evolucionando, lo que puede verse en el creciente número de publicaciones sobre el tema. Sin embargo, todos estos métodos comparten una base común que se podría resumir en:

> *El algoritmo mantiene una colección de potenciales soluciones para un problema. Algunas de estas soluciones son usadas para crear nuevas soluciones mediante el uso de ciertos operadores. Estos operadores actúan en las posibles soluciones y crean nuevas. La soluciones potenciales en las que el operador actúa son seleccionadas en base de su calidad como solución para con el problema objetivo. El algoritmo usa este proceso repetidamente para generar nuevas soluciones hasta que se alcanza algún criterio de convergencia.*[25]

Esta misma metodología se suele encontrar en la literatura, pero haciendo alusión al origen inspirado en la biología, utilizando conceptos como *genes, alelos, cromosomas, población, generación, recombinación*, etc.

Como veíamos, la evolución no actúa directamente sobre los organismos, sino que en sus cromosomas (codificación que contiene toda la información que define al individuo). Y es esta información la que se pasa de una generación a otra en el momento de la reproducción del los individuos. La naturaleza posee una gran variedad de estrategias de reproducción, donde las más esenciales son la *mutación* (variaciones aleatorias en el cromosoma) y la *recombinación* (intercambio de información genética entre los individuos). La selección natural es el mecanismo que relaciona los cromosomas con el desempeño de los individuos que representan, favoreciendo la reproducción de los mejor adaptados y generando la exclusión de los más débiles.

Este es el proceso que los investigadores han intentado reproducir, con la salvedad de que en la naturaleza vemos una evolución constante en función de los requerimientos del entorno y no dirigida a ningún objetivo en particular, mientras que los sistemas diseñados por humanos están orientados a completar tareas específicas. Esto genera dos enfoques claros a la hora de la construcción de sistemas basados en la naturaleza:

1. Intentar reproducir fielmente los procesos de evolución natural.
2. Usar los principios naturales como inspiración para desarrollar sistemas eficientes para lograr tareas determinadas.

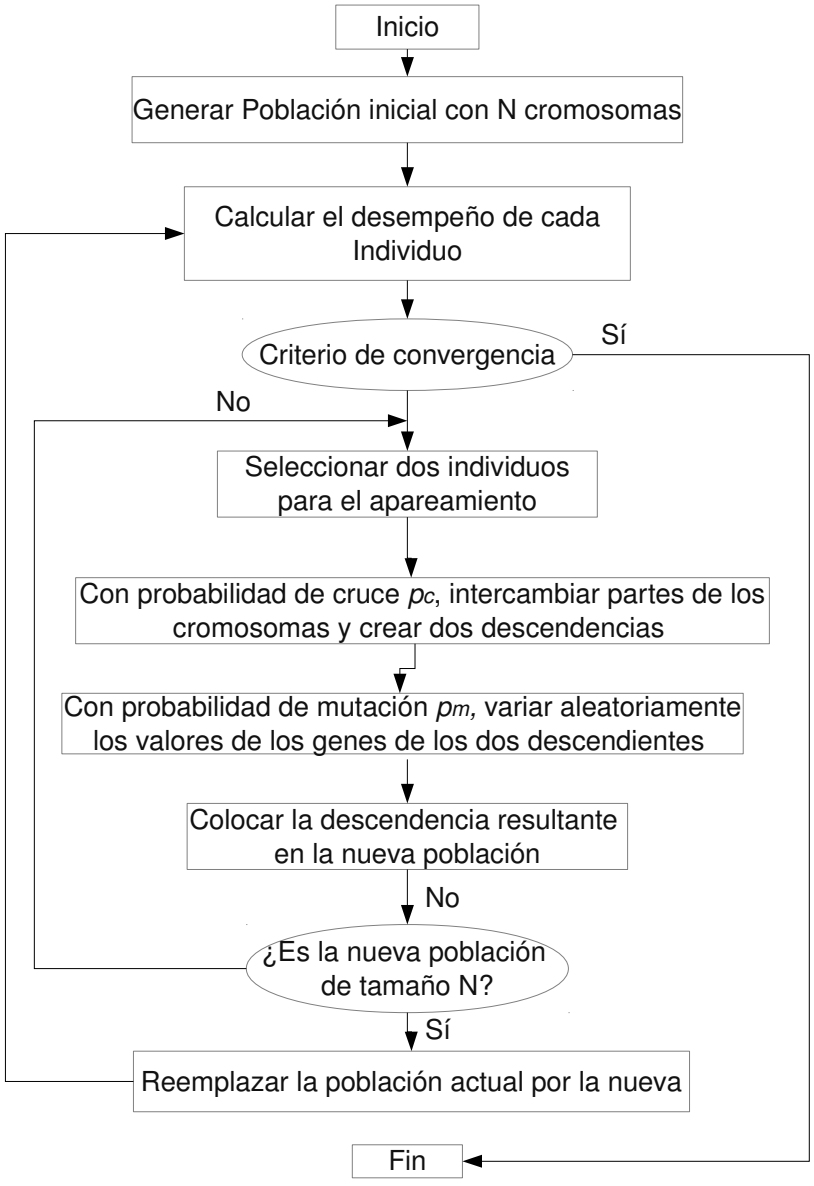

Figura 2.11: Pseudo código típico de un algoritmo genético simple.

El primero se ve más en el campo de la *Inteligencia Artificial*, y es interesante pues reproduce fenómenos naturales como el parasitismo, relaciones de depredador-presa, etc., permitiendo el estudio de estos. Sin embargo, es el segundo enfoque el más interesante, y constituye el origen de los

algoritmos evolutivos.

Un algoritmo evolutivo es escencialmente una metodología estocástica iterativa que genera soluciones tentativas para un problema dado. El algoritmo manipula una población P de individuos, donde cada cual consiste, básicamente, de uno o varios cromosomas. Un proceso de decodificación permite relacionar los cromosomas contenidos en la población con las soluciones correspondientes. Los cromosomas se dividen en pequeñas unidades llamadas *genes*, que puede tomar ciertos valores llamados *alelos*. A su vez, las soluciones se relacionan con una medida de desempeño, que se conoce como *fitness*.

Inicialmente, se genera una población aleatoria de N individuos, a los cuales se les evalúa sus respectivos fitness. Con esto se inicia el proceso de *selección*, en el que se elige un número de individuos promisorios para reproducirse mediante la acción de los distintos *operadores genéticos*. Esto se hace hasta completar una nueva *generación* de individuos, a la cual se aplica el mismo proceso que puede verse en la figura 2.11.

Los primeros tipos de algoritmos evolutivos comenzaron su existencia a mediados de los años 60'. Durante estos años, y casi simultáneamente, científicos de distintas partes del mundo empezaron a desarrollar algoritmos basados en la naturaleza con el objetivo de resolver problemas. Estas fuentes originaron tres modelos de algoritmos evolutivos [24]:

- *Programación evolutiva*: Se enfoca en la adaptación de individuos en vez de la evolución de su material genético [28]. Esto implica una visión más abstracta del proceso evolutivo, en el que se modifica directamente el comportamiento del individuo (en vez de alterar sus genes). Típicamente este comportamiento es modelado con el uso de complicadas estructuras de datos tales como autómatas finitos o grafos. Tradicionalmente, esta metodología sólo utiliza reproducción asexuada (e.g. mutación), introduciendo pequeños cambios en una solución existente, y técnicas de selección basadas en la competencia directa entre los individuos.

- *Estrategias Evolutivas*: Esta técnica fue desarrollada para resolver problemas ingenieriles y se caracteriza por la manipulación de arreglos de números de punto flotante, aunque también existen versiones discretas [29, 30]. Como en el caso anterior, la reproducción asexuada es, casi siempre, el único operador genético utilizado. Una característica importante del método es que suele utilizar métodos auto-adaptativos para controlar la aplicación de los operadores genéticos. Estos mecanismos están orientados a optimizar el progreso de la búsqueda al

evolucionar, además del individuo en sí, parámetros que controlan los operadores genéticos, como por ejemplo las probabilidades con que actúan los operadores, las distribuciones probabilísticas, etc.

- *Algoritmos Genéticos*: Este método es posiblemente el más extendido de todas las formas de algoritmos evolutivos, al punto de que varios de sus postulados se consideran practicamente obligatorios [31]. La principal característica de los algoritmos genéticos es el uso de la recombinación de individuos como operador genético principal. La razón de este postulado es que diferentes partes de la solución óptima pueden ser descubiertas independientemente entre los individuos, para ser luego reunidas tras el proceso de selección. También se utiliza la reproducción asexuada, o *mutación*, pero en un nivel secundario, cuyo único propósito es mantener la diversidad de la población.

Estas escuelas de algoritmos evolutivos han sido combinadas en varias ocaciones por varios autores, generando nuevas y novedosas variantes, entre las que se pueden nombrar la *programación genética*, *neuro-evolución*, etc. A continuación se profundiza en los modelos implementados en el presente trabajo de memoria.

2.4.1. Algoritmos Genéticos

Los algoritmos genéticos son métodos de búsqueda basados en la selección natural. Combinan la superviviencia de los individuos más aptos con el intercambio de información guiada probabilísticamente. En cada generación se crea una nueva población de individuos usando partes de los individuos más aptos de la generación anterior. Ocasionalmente, también se introducen innovaciones aleatorias en las estructuras.

Generalmente, se pueden agrupar los métodos de optimización en dos grandes categorias:

- *Basados en cálculos*: Estos métodos han sido desarrollados extensamente en los últimos siglos. Sin embargo, suelen estar limitadas a la vecindad de punto inicial de búsqueda, siendo más bien de alcance local. Esto es porque seleccionan la dirección de exploración en función del gradiente, lo que además supone una serie de condiciones sobre la derivabilidad de la función objetivo, que en muchos casos pueden no darse.
- *Enumerativos-Aleatorios*: Estos consisten básicamente en probar, aleatoria o secuencialmente, todas las posibilidades de solución en un espacio de búsqueda finito o infinito discretizado. Si

bien la implementación de este tipo de algoritmos parece simple, son ineficientes en espacios de búsqueda grandes.

Si bien es imposible negar la utilidad de estos métodos, debemos reconocer que son pocos robustos. Por ejemplo, si tomamos una función suave y continua, es esperable que los métodos basados en cálculos sobrepasen en desempeño a las enumerativas-aleatorias, alcanzando al punto óptimo de forma mucho más rápida. Pero en el caso contrario, si tenemos una función acentuadamente oscilante y errática, con infinidad de máximos locales, es casi evidente que los métodos basados en cálculos no serán tan eficientes, en favor de los métodos enumerativos-aleatorios. Dado esto, sería deseable un método que fuese robusto independientemente del tipo de función con la que se quiera trabajar.

Para solucionar el problema de robustez presente en los métodos tradicionales, los algoritmos genéticos se diferencian cuatro puntos fundamentales:

1. Los algoritmos genéticos no trabajan directamente sobre los parámetros, sino sobre una representación codificada de estos.
2. La búsqueda se realiza desde una colección de puntos, en vez de sólo uno a la vez.
3. Funcionan en base a información sobre el desempeño de los candidatos a soluciones, no sobre derivadas u otras condiciones de la función objetivo.
4. Emplea reglas de transición probabilísticas, no deterministicas.

Los algoritmos genéticos, al menos como fueron concebidos inicialmente, requieren que las posibles soluciones sean codificadas en cadenas de largo finito y con un alfabeto también finito (posibles valores para las posiciones de esta cadena). Por ejemplo, una cadena binaria de 3 bits puede representar números enteros entre 0 y 7. Estrategias más modernas de representación permiten que estas cadenas sean de tamaño variable, representando vectores de números reales cuyos valores se modifican utilizando distribuciones probabilísticas, pero esto se verá en más profundidad en el capítulo de neuro-evolución.

La mayoría de los métodos de optimización se mueven de un punto a otro en el espacio de búsqueda empleando alguna regla de transición que lo determina. Este enfoque de moverse punto a punto es riesgoso pues es propenso a encontrar máximos relativos en espacios multimodales (de varios máximos). Por el contrario, los algoritmos genéticos trabajan en función de una colección de puntos simultáneamente, explorando varios focos de atracción en forma paralela, reduciendo la

posibilidad de concentrarse en un solo máximo local.

Otro punto a favor de los algoritmos genéticos es que funcionan de forma independiente de la función objetivo, mientras que los métodos tradicionales suelen requerir una gran cantidad de información extra, como las derivadas en el caso de las técnicas basadas en el gradiente. Para el correcto funcionamiento de los algoritmos genéticos solo es necesario una medida de desempeño de los candidatos a solución. Esto convierte al método en un mecanismo más estandarizado, pues la única adaptación que se debe hacer entre el algoritmo y el problema es la evaluación de esta medida de desempeño para cada individuo (además, claro, de la codificación inicial de los parámetros). De hecho, es esta independencia del método para con el problema que han hecho posible que los algoritmos genéticos sean implementados no solo como métodos de optimización, sino que también como algoritmos para el aprendizaje de comportamientos en robots, aprendizaje de lógica difusa, diseño automatizado, etc. Un trabajo destacable en el diseño asistido por algoritmos genéticos es el desarrollo de intrincadas geometrías para antenas tipo "crooked-wire" (alambre retorcido) [18].

Los algoritmos genéticos son un ejemplo de método que utiliza la elección aleatoria como herramienta para guiar probabilísticamente una búsqueda altamente explotable en un espacio parametrizado. Las reglas de transición probabilísticas seleccionan a los individuos con mejor desempeño para compartir las características contenidas en su codificación, propagando información valiosa sobre una solución potencial.

Se han nombrado varias de las características principales que diferencian a los algoritmos genéticos de los métodos tradicionales. A continuación se presentan los tres operadores básicos que hacen posible este comportamiento: selección, recombinación y mutación.

Selección

La selección es el proceso en el que las cadenas que representan a los individuos son copiados en relación con su función de desempeño. Es natural pensar en este desempeño como una función de ganancia que se quiere maximizar, por lo que individuos con mayor valor de desempeño deben tener mayor probabilidad de contribuir con una o más descendencia a la próxima generación. Este operador es, claro está, una versión artificial de la selección natural, donde sobrevive el individuo más fuerte (o de mejor desempeño).

Este operador puede ser implemetado de diferentes formas, pero la más común es utilizar el

método de la *ruleta*. Este método consiste en generar una plataforma de selección en el que cada espacio (asociado a cada individuo) es proporcional a su desempeño. De esta forma, si se utiliza un generador de números aleatorios uniforme, la probabilidad de que un individuo sea elegido es:

$$p_i = \frac{f_i}{\sum_{i \in I} f_i} \tag{2.2}$$

Donde f_i es el desempeño del individuo i, e I es la población. Con este operador se determina que individuos son seleccionados para entrar al *pozo de reproducción*, donde trabajan el resto de los operadores genéticos. Este proceso se repite hasta que se completa la nueva generación de individuos.

Existen otras formas de seleccionar a los individuos que entran a pozo de reproducción, a continuación se presentan algunas [26]:

- *Ranking*: En esta metodología se ordena a toda la población según su desempeño. Luego, a cada individuo se le asigna un número de descendientes en función de su posición dentro del ranking. Este método ha recibido críticas pues disocia el desempeño del individuo con su reproducción.
- *Torneo*: Se selecciona aleatoriamente un conjunto de k individuos de la población, de los cuales el mejor entra al pozo de reproducción. Este proceso se repite hasta completar la población de la siguiente generación.

Recombinación

Luego de la selección, dos individuos son seleccionados para recombinarse con probabilidad p_r. Este operador genera dos descendencias producto del intercambio de información genética entre los individuos seleccionados. Si la recombinación procede, se elige un punto al azar dentro de la cadena de codificación de cada individuo, donde se intercambia el material contenido desde el punto hacia el fin (o principio) de la cadena codificada. En caso de no proceder la recombinación (no se logra vencer la probabilidad) se tiene que la descendencia es igual a los individuos seleccionados. Generalmente se recomienda que la probabilidad de recombinación p_r oscile entre 0.6 y 0.8, esto dado que éste operador es el principal mecanismo de esparcimiento de información en los algoritmos genéticos.

Existen otros enfoques para abordar este operador, donde la principal variante consiste en declarar más de un punto de intercambio de información. Este enfoque ha demostrado mejores resultados en problemas multimodales [21].

Mutación

Este operador, a diferencia de los anteriores, no actúa sobre los individuos, sino que sobre cada uno de los genes en la cadena codificada. Con una probabilidad p_m el gen cambiará arbitrariamente de valor. Si bien la implementación de este operador es clara en sistemas de codificación binaria, también se ha implementado con éxito en representaciones con números reales, donde se suma o resta al gen un valor generado aleatoriamente con alguna distribución probabilística. La literatura recomienda que la probabilidad de mutación sea tal que sólo un gen en la cadena sea modificado [21].

Otros Operadores

A pesar de la exploración del espacio de búsqueda multipunto característica de los algoritmos genéticos, en ciertas circunstancias se puede presentar el problema de *convergencia prematura* (figura 2.12). Éste problema consiste en que la población se concentra en torno a un máximo local (la población se uniformiza), impidiendo la exploración efectiva del resto del espacio de búsqueda y perdiendo de alcance al máximo global. Esto se puede producir, entre otros, por los siguientes motivos:

- Población inicial: Esto es cuando no hay mucha diversidad en los individuos de la población inicial, o incluso pueden estar agrupados en un espacio pequeño del espacio de búsqueda, por lo que la búsqueda se concentra solo en éste.
- Super-individuos: Esto sucede cuando dentro de la población existe un individuo, o más, cuyo desempeño destaca considerablemente por sobre el resto, sin que esto signifique que sea una solución promisoria. De esta forma, este super-individuo genera una gran cantidad de descendencia que termina por acaparar la mayor parte de la población. Este problema es característico de las primeras etapas del algoritmo.

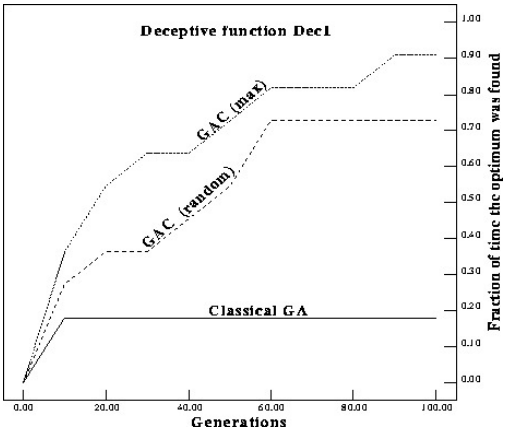

Figura 2.12: Curvas de aprendizaje para distintas implementaciones de algoritmos genéticos sobre un mismo problema. Se puede ver que algunos métodos se estancan tempranamente en valores que no corresponden al máximo global. Esto se produce por una mala exploración del espacio de búsqueda.

Para enfrentar este problema se han generado varios operadores que buscan mantener la diversidad de individuos en la población. Usualmente la población inicial se genera aleatoriamente, por lo que no hay mayor control sobre la concentración de individuos. Al respecto se han desarrollado varias estrategias de reinicio de la población, incluso la evolución de poblaciones paralelas, con intercambios de individuos cada cierto tiempo. Otra estrategia recurrente es explotar el concepto de especiación, donde los individuos se agrupan según criterios de semejanza del genotipo, limitando la reproducción a individuos de la misma especie. Un enfoque reciente que ha obtenido resultados destacables, conocido como ALPS[3], consiste en desarrollar paralelamente varias capas de poblaciones diferenciadas entre ellas por la edad del material genético de los individuos, donde la primera reinicia constantemente la población [9].

Un operador que ataca el problema de los super-individuos es conocido como *escalamiento de fitness*. Este método es fácilmente integrable en las rutinas de selección, pues toma los desempeños de los individuos y los reajusta. Luego, estos desempeños modificados son procesados por el algoritmo de selección. La idea de este mecanismo es prevenir que el desempeño de los super-individuos sea excesivamente alto con respecto al resto de la población en las etapas tempranas de la evolución, para que el resto de los individuos tengan la oportunidad de desarrollarse antes de ser descartados definitivamente. Este mismo operador cumple otro rol importante en las etapas tardías de la evolución, donde la población tiene desempeño más bien homogéneo. En las etapas tardías este operador efectúa la operación opuesta, es decir, acentúa las diferencias de los desempeños de los individuos, aumentando las probabilidades de selección de los individuos que sean superiores al resto [21]. Este

[3]Siglas para Estructura de Población Separada por Edades (*Age-Layered Population Structure*).

comportamiento se logra manipulando el concepto de *presión de selección*, que consiste básicamente en modificar el número esperado de descendencia del mejor individuo. Por ejemplo, una presión de selección de 2 corresponderia a que el desempeño del mejor individuo sea cercano a dos veces el desempeño promedio de la población. En la literatura se recomienda utilizar un valor de presión entre $1{,}2 - 2$ [21, 22].

Si bien que un individuo sea seleccionado garantiza que éste generará descendencia, no garantiza necesariamente que este se conserve íntegramente. Esto puede producir la pérdida de material genético valioso. Para solucionar esto se desarrolló el concepto de *elitismo*, que consiste básicamente en traspasar a los mejores individuos (usualmente no más de 2) directamente a la generación siguiente, saltándose el proceso de selección. Esto ha probado mejorar considerablemente el rendimiento de los algoritmos genéticos.

2.4.2. Neuro-Evolución

Para entender que es la neuro-evolución primero hay que entender qué son las redes neuronales. Las redes neuronales son modelos artificiales del cerebro y el sistema nervioso, capaces de desarrollar cálculos múltiples en forma paralela, siendo capaces de aprender a desarrollar comportamientos complejos mediante cálculos simples.

Las redes neuronales están compuestas por dos elementos básicos; nodos y conexiones, analogías de las neuronas y sinápsis del sistema nervioso. Los nodos están distribuidos en distintas capas, de forma tal que los nodos de una capa envían información a la siguiente mediante las conexiones. La estructura típica de una red consiste en una capa de entrada, que recibe información del exterior, una serie de capas escondidas, que procesan la información, y una capa de salida.

Las neuronas son las unidades de procesamiento de información. Usualmente consisten en funciones de activación con un valor umbral, que pueden ser lineales, logísticas, binarias, hiperbólicas, etc. Estas reciben la información proveniente de los nodos de la capa anterior y generan una salida que es enviada a la siguiente capa. Las conexiones suelen tener un valor asociado, llamado *peso sináptico*. De esta forma, la entrada de cada neurona suele ser la suma ponderada de las salidas de los nodos de la capa anterior con los pesos de las conexiones que las unen.

El tipo de red neuronal más simple es conocida como *feed forward*, en la que la información recorre la red unidireccionalmente. Sin embargo, existen también redes con conexiones recurrentes, lo

que significa que las conexiones pueden ser multidireccionales, desde capas de neuronas superiores hacia las inferiores. Esto le da a la red una noción de memoria, pues se rescatan estados anteriores de una capa dentro de una misma evaluación de la red.

Una de las características más notables de las redes neuronales es su capacidad de entrenamiento y aprendizaje. Usualmente, estos mecanismos de aprendizaje manipulan los valores de los pesos sinápticos, quedando invariables las funciones de activación de las neuronas. Para esto existen varias técnicas, como por ejemplo:

- *Entrenamiento Supervisado*: Al momento del entrenamiento se conoce un par de entrada y salida deseado, con lo que se genera una medida de error del funcionamiento de la red. Con esta medida de error se pueden ajustar los valores de los pesos sinápticos de la red iterativamente hasta lograr el comportamiento deseado.
- *Entrenamiento Reforzado*: Los valores de los pesos sinápticos son ajustados en función de indicadores de desempeño.
- *Entrenamiento No-Supervisado*: No existe ningún tipo de guía. En este caso la red neuronal genera categorias en función de patrones en el conjunto de entrada. De esta forma, la red actuará como catalogador.

Con estos antecedentes se puede entender lo que es la *neuro evolución*. La neuro evolución es la evolución de redes neuronales empleando algoritmos genéticos como recurso en tareas complejas de aprendizaje reforzado. Este enfoque ha demostrado ser más veloz en la búsqueda de soluciones que la mayoría de los métodos de aprendizaje reforzado [32, 33].

En las estrategias de neuro-evolución tradicionales, la cantidad de nodos y capas de la red neuronal son determinadas antes de la realización del experimento. Típicamente, estas topologías consisten de sólo una capa escondida, donde todas las conexiones están habilitadas. De esta forma, la evolución sólo opera recombinando vectores que representan los pesos sinápticos y mutando cada valor de estos. Así, el objetivo de este tipo de neuro-evolución es optimizar los pesos sinápticos que determinan la funcionalidad de la red.

Sin embargo, los pesos sinápticos no son la única parte de las redes neuronales que participan de su comportamiento. La topología, o estructura de nodos, también contribuye en su funcionalidad. Aunque, en principio, una red neuronal completamente conexa es capaz de aproximar cualquier función continua, existen varias ventajas en la evolución de las topologías, partiendo porque ya no

es un humano quién tiene que determinar la topología final de la red, lo que usualmente consiste en un proceso de ensayo y error. Además, evolucionar topologías de complejidad creciente, partiendo desde estructuras simples, ahorra tiempo y recursos en el entrenamiento de los pesos sinápticos.

La evolución de la topología presenta, sin embargo, varias complicaciones:

1. El problema de utilizar una representación genética tal que la recombinación de estructuras dispares sea significativa.
2. Cómo proteger las innovaciones estructurales, que necesitan algunas generaciones para ser optimizadas, de ser descartadas prematuramente.
3. Cómo minimizar las topologías a través de la evolución sin la necesidad de generar una función de fitness que mida la complejidad de las estructuras.

Estas interrogantes son resueltas por un método de neuro evolución conocido como NEAT[4] [22]. Este método será uno con los cuales se trabajará en este trabajo de memoria.

2.4.3. NEAT

Desde la década de los 90's se han desarrollado varios sistemas que evolucionan tanto la topología como los pesos sinápticos de las redes neuronales. Estos métodos engloban una serie de ideas acerca de cómo debieran implementarse la *evolución de pesos y topologías en redes neuronales artificiales*, o TWEANNs[5].

Codificación Genética

Todos los métodos TWEANNs deben plantearse el cómo codificar las redes utilizando una representación genética eficiente. Estos pueden clasificarse entre aquellos que utilizan una codificación directa e indirecta.

Los esquemas de codificación directa, empleados por casi todos los TWEANNs, explicitan en el genoma cada nodo y conexión que debe aparecer en el genotipo. En contraste, la codificación indirecta solo especifica reglas para construir el fenotipo. Estas reglas pueden ser especificaciones por capas o reglas de creciemiento por división celular [34, 35]. La codificación indirecta permite una representación más compacta que la directa, pues no se especifica cada conexión ni nodo.

[4]Siglas para Neuro-Evolución de Topologías Crecientes (*Neuro Evolution of Augmenting Topologies*).
[5]Siglas para *Topology and Weight Evolving Artificial Neural Networks*

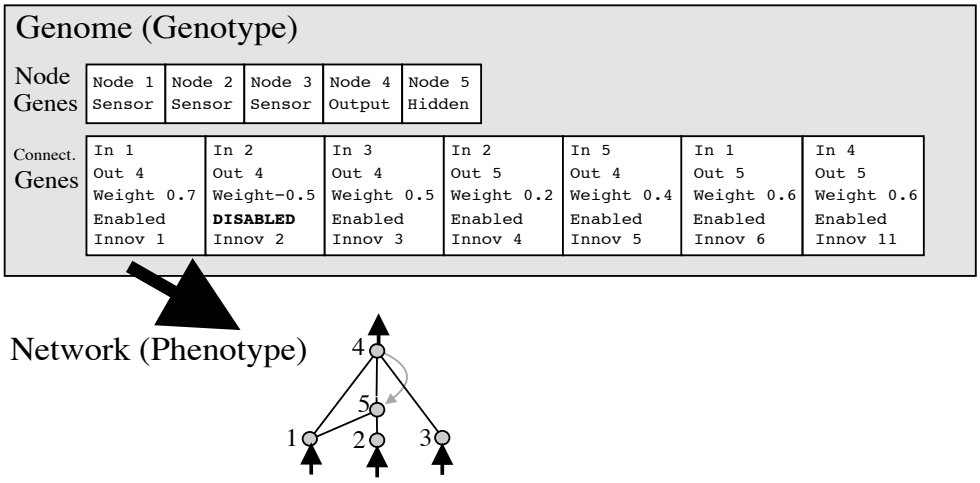

Figura 2.13: Ejemplo de relación entre genotipo y fenotipo en NEAT. Figura original de [22].

La representación genética en NEAT es directa, y está diseñada para permitir que los genes correspondientes puedan alinearse fácilmente para la recombinación. Los genomas son representaciones lineales de la conectividad de la red (figura 2.13). Esta representación consiste de dos listas, donde la primera describe los nodos disponibles y su tipo (puede ser de entrada, salida u oculto). La segunda lista describe las conexiones, especificando los nodos de entrada y salida, el peso sináptico de la conexión, si la conexión está o no habilitada y una marca indentificatoria de innovación, esto será explicado más adelante.

La mutación opera tanto sobre los pesos sinápticos como sobre las estructuras de la red. La mutación opera sobre los pesos como en cualquier otro método de neuro evolución, añadiendo o no números generados aleatoriamente a cada peso. Por otra parte, las mutaciones estructurales pueden ocurrir de dos formas, ya sea agregando conexiones o nodos (figura 2.14). Cada operación efectiva de mutación agrega genes a las listas. La mutación que agrega conexiones genera una nueva que une dos nodos previamente disconexos con un peso aleatorio. La mutación que agrega nodos divide una conexión preexistente entre dos nodos, colocando al nuevo entre estos con las nuevas conexiones respectivas y deshabilitando la preexistente. Para reducir el impacto de esta mutación, se conserva el peso sináptico preexistente entre el nodo nuevo y el de llegada, mientras que la conexión entre el nodo nuevo y el antiguo de salida recibe un peso sináptico de 1.

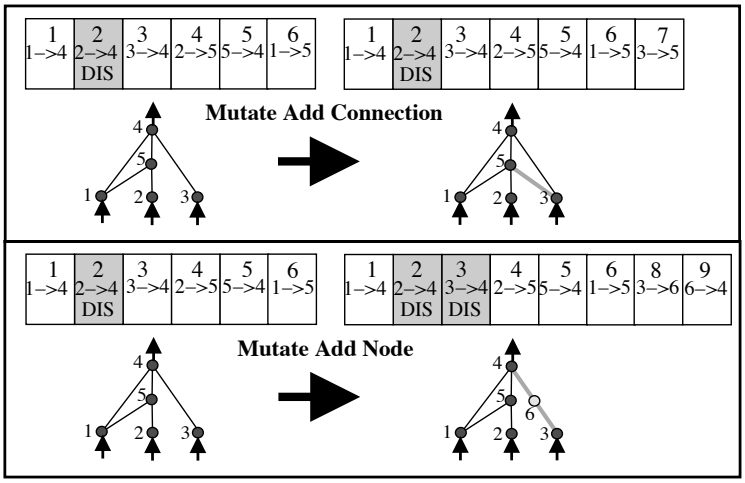

Figura 2.14: En la figura se muestran los dos tipos de mutación estructurales en NEAT. El primer tipo de mutación agrega una conexión entre dos nodos, mientras que el segundo tipo de mutación agrega un nodo nuevo, dividiendo una conexión preexistente entre dos nodos. Ilustración original de [22].

Marcas Históricas de los Genes

Uno de los principales problemas de la neuro evolución es el *Competing Conventions Problem*[6]. Este problema consiste en que pueden existir varias formas de representar un solución en el problema de optimizar los pesos en una red neuronal. Cuando dos representaciones corresponden a la misma solución, es muy posible que su recombinación sea catastrófica.

En la figura 2.15 se muestra el problema para una red con 3 nodos escondidos. Los nodos A, B y C pueden representar la misma solucón de $3! = 6$ formas distintas. Al recombinar estas representaciones es muy posible que se pierda información esencial. Por ejemplo, al recombinar [A,B,C] con [C,B,A] se podría obtener [C,B,C], con lo que se pierde un tercio de la información.

Para solucionar el problema de las convenciones competitivas, en NEAT se emplean marcas históricas del origen de cada gen. Con esto se puede identificar exactamente qué genes se corresponden entre individuos. Dos genes con la misma marca histórica representan la misma estructura (posiblemente con distintos pesos sinápticos).

Cada vez que aparece un nuevo gen estructural, se incrementa un número global de innovación y se le asigna al nuevo gen. De esta forma, la marca histórica crea una linea cronológica de la aparición de cada gen. Por ejemplo, digamos que las mutaciones en la figura 2.14 ocurrieron

[6]Problema de las convenciones competitivas.

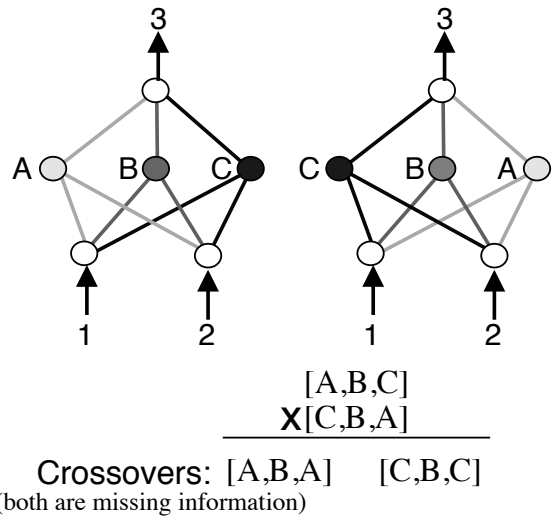

Figura 2.15: El problema de las convenciones competitivas. Dos redes neuronales con la misma funcionalidad pueden ser representadas de distinta forma. Esto hace que la recombinación sea potencialmente destructiva al perderse información. Figura original de [22].

secuencialmente. El nuevo gen de conexión creado en la primera mutación recibe el número 7, y las dos nuevas conexiones creadas producto de agregar un nodo reciben los número 8 y 9. En el futuro, cuando estos cromosomas se recombinen, su descendencia mantendrá las mismas marcas históricas en cada gen, conservándose a través de la evolución.

Un posible problema que puede surgir es que la misma innovación estructural reciba números de identificación distintos en la misma generación, si llegase a ocurrir en varias ocaciones. Esto se puede resolver manteniendo una lista de innovaciones que ocurren en cada generación, con lo que se puede forzar que una misma innovación generada más de una vez en una generación reciba una única marca histórica. Además, con esto se previene un aumento descontrolado de marcas históricas.

De esta forma, para el apareamiento de dos cromosomas se alinean los genes correspondientes según sus marcas históricas (figura 2.16). Los genes que no coinciden son o exceso (*excess*) o disjunturas (*disjoints*), dependiendo si las marcas históricas de estos genes están contenidas en el rango correspondiente al otro cromosoma en la recombinación. En la recombinación se eligen genes aleatoriamente entre aquellos compartidos entre ambos padres, mientras que todos los excesos y disjunturas son heredados.

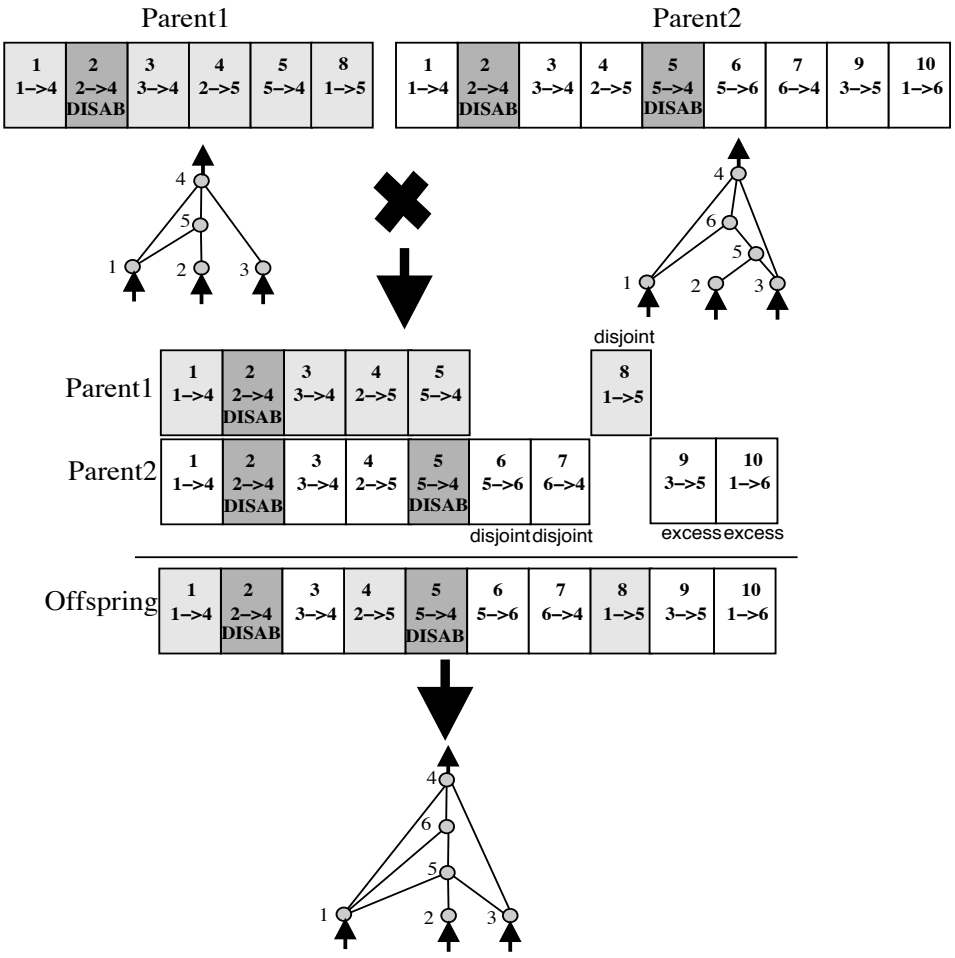

Figura 2.16: Ejemplo de recombinación entre dos cromosomas en NEAT. En el proceso se alinean los cromosomas según sus marcas históricas. Los genes a heredar, contenidos en el rango del cromosoma más corto, son seleccionados aleatoriamente entre los padres, mientras que los genes de exceso son siempre heredados. Figura original de [22].

Protección de la Innovación a través de la Especiación

En los métodos TWEANNs, las innovaciones toman forma al incorporar nuevas estructuras a las redes a través de la mutación. Estas modificaciones suelen generar el que desempeño de los individuos disminuya inicialmente, pues estas nuevas estructuras aún no han tenido la oportunidad de ser optimizadas. Producto de esta baja de fitness, es posible que se pierda al individuo en el proceso de selección, pues compite con otros individuos ya optimizados. Es por esto que hay que desarrollar mecanismos que protejan las innovaciones estructurales.

La especiación de la población es un mecanismo que permite que los individuos compitan entre

sus pares, en vez de hacerlo contra toda la población. De esta forma, las innovaciones estructurales se mantienen protegidas en nuevos grupos, donde tienen el tiempo de ser optimizadas, mientras compiten dentro de este nuevo grupo. La idea es agrupar a los individuos según su topología, generando de esta forma las especies. En esta tarea es realizable al explotar el concepto de las marcas históricas.

El número de excesos y disjunturas son medidas naturales de las semejanzas topológicas entre individuos. Mientras más disjunturas o excesos, menos semejantes son los individuos. De esta forma, NEAT emplea una relación que combina estos conceptos, añadiendo además un término que mide la diferencia promedio de pesos sinápticos entre los genes correspondientes, en el que se incluye a los genes desactivados (ecuación 2.3).

$$\delta = \frac{c_1 E}{N} + \frac{c_2 D}{N} + c_3 \bar{W} \tag{2.3}$$

Los coeficientes c_1, c_2 y c_3 ajustan la importancia de cada término. N es la cantidad de genes en el cromosoma más largo, $\bar{W}$ es el término de semejanza de pesos sinápticas y δ mide la distancia entre los individuos, donde un valor umbral determina si estos corresponden o no a una misma especie.

Además, NEAT utiliza el *reparto explítico del fitness* como mecanismo de reproducción [36], donde cada organismo en la misma especie debe compartir el fitness. De esta forma se previene que una especie crezca demasiado y termine ocupando toda la población, aún cuando varios individuos de la especie tengan fitness altos. El fitness ajustado f'_i del individuo i se calcula de la siguente forma:

$$f'_i = \frac{f_i}{\sum_{j=1}^{n} sh(\delta(i,j))} \tag{2.4}$$

Donde f_i es el fitness original del individuo, y la función $sh(\delta(i,j)$ se define como 1 si los individuos i y j son de la misma especie, y 0 en caso contrario.

Crecimiento Incremental desde Estructuras Minimales

En varios sistemas TWEANNs se genera una población inicial con topologias aleatorias, lo que asegura una diversidad inicial de la población. Pero este enfoque puede producir muchos problemas, por ejemplo que una red no tenga contacto con sus nodos de salida. Este tipo de problemas terminan por relentizar el proceso evolutivo, pues hay que esperar que los mecanismos de selección

descarten a los individuos disfuncionales.

Además hay que considerar que es deseable minimizar la topología de los individuos, puesto que de esta forma se reduce el consumo de recursos en la optimización de pesos que no necesariamente aportan al funcionamiento de la red.

En NEAT se privilegian las soluciones minimales al empezar con una población inicial uniforme con ningún nodo escondido (los nodos de entrada se conectan directamente a los de salida). La topología de los individuos crece a medida que ocurren las mutaciones estructurales, y su supervivencia solo depende del desempeño, por lo que todas las innovaciones topológicas deben ser justificadas. Además, con este enfoque se minimiza el espacio de búsqueda inicial, dándole a NEAT ventajas de rendimiento por sobre el resto de los sistemas TWEANNs [22].

Capítulo 3

Metodología

El trabajo de memoria consiste en el desarrollo de métodos de síntesis de algoritmos para control descentralizado de robots modulares genéricos empleando algoritmos evolutivos. Para esto, se seleccionaron dos estretegias evolutivas, Algoritmos Genéticos y NEAT.

Estas dos metodologías hacen referencia a dos enfoques sobre los algoritmos de control. El primer enfoque es el clásico, desarrollado en [3, 4, 20], donde los algoritmos consisten en un conjunto de reglas geométricas explícitas del tipo autómata celular. Estas reglas explícitas son aptas para ser codificadas en forma binaria, cuya representación puede ser evolucionada directamente con algoritmos genéticos.

El segundo enfoque es más novedoso, puesto que el algoritmo no es explicitado en una serie de reglas locales, sino que la acción correspondiente a una configuración geométrica dada, está dada por un cálculo realizado por una red neuronal, la cual es evolucionada con NEAT. Esta red neuronal inspecciona la vecindad inmediata del módulo activo, procesando estos datos para generar una acción a realizar.

Tanto la representación binaria como el procesamiento con redes neuronales se ven favorecidas con un simulador que trabaje valores numéricos, por lo que se elige esta forma.

Con estos antecedentes, se establecen las etapas del desarrollo del trabajo de memoria.

3.1. Etapas de Trabajo

Las etapas del trabajo se pueden agrupar en las siguientes:

1. Desarrollo del simulador.

2. Definición de función objetivo.
3. Implementación algoritmos evolutivos.
4. Pruebas preliminares y ajuste de parámetros de algoritmos evolutivos, como probabilidades y criterios de convergencia.
5. Obtención de resultados parciales y selección de parámetros de simulación.
6. Resultados finales y caracterización de los mejores individuos.

Estas etapas aplican tanto para algoritmos genéticos como para NEAT. Sin embargo, el simulador es el mismo en ambos casos, salvo adaptaciones para la evaluación del los métodos.

3.1.1. Desarrollo del Simulador

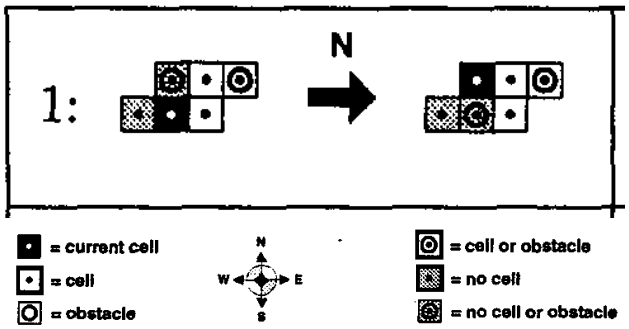

Figura 3.1: Ejemplo de regla para control descentralizado de robots modulares reconfigurables. Esta regla consta de dos partes. La primera, arriba a la izquierda, nos da una configuración geométrica en torno al módulo activo (en negro). Esta configuración consta de disposiciones relativas de otros módulos u obstáculos del escenario. La segunda parte de la regla consiste en la acción, que en este caso es desplazar el módulo activo hacia arriba. Fuente [4].

Para el desarrollo del simulador se trabajó, en primera instancia, de forma tal que fuesen aplicables las reglas propuestas en [3, 4, 20]. Estas reglas se tradujeron a una representación matricial. Por ejemplo, la configuración de entrada en la figura 3.1 se puede representar por una matriz de 5x5 (figura 3.2).

0	0	0	0	0
0	0	1	3	2
0	1	3	3	0
0	0	0	0	0
0	0	0	0	0

Figura 3.2: Configuración geométrica requerida en torno a un módulo activo. El módulo activo es siempre el 3 al centro de la matriz.

0	1	2
3	–	4
5	6	7

Figura 3.3: Si la configuración en torno al módulo activo es satisfecha, el módulo procede a desplazarse a alguna posición adyacente indicada por la matriz de acción. Por ejemplo, si la acción fuese subir, esto se representa por un 1.

Donde cada número se interpreta como:

- 0: posición no evaluada.
- 1: espacio vacío requerido.
- 2: obstáculo.
- 3: módulo del robot.

La salida de la regla es desplazarse en solamente una posición, en cualquier dirección. Esto se puede representar con una matriz de 3x3 (figura 3.3). Donde cada número esta asociado a una única posición de la matriz.

Con esta representación de la regla cada posición de la matriz de entrada puede ser representada con 2 bits, y la acción puede ser representada con 3 bits.

En el caso del presente trabajo, se elige inspeccionar la vecindad inmediata del módulo activo, con lo que la matriz de entrada se reduce a una de 3x3, cuyo centro es el módulo activo. De esta forma, cada regla queda definida por 21 bits a manipular por el algoritmo genético.

Es importante recalcar dos puntos:

1. En la representación elegida se pierden 2 bits, puesto que en el centro de la matriz de entrada siempre se tiene un módulo del robot.
2. El algoritmo de control descentralizado consiste en un conjunto de reglas, por lo que cada algoritmo a evaluar queda definido por $\{21 \times$ número de reglas$\}$ bits.

Así, el simulador se ve como en la figura 3.4.

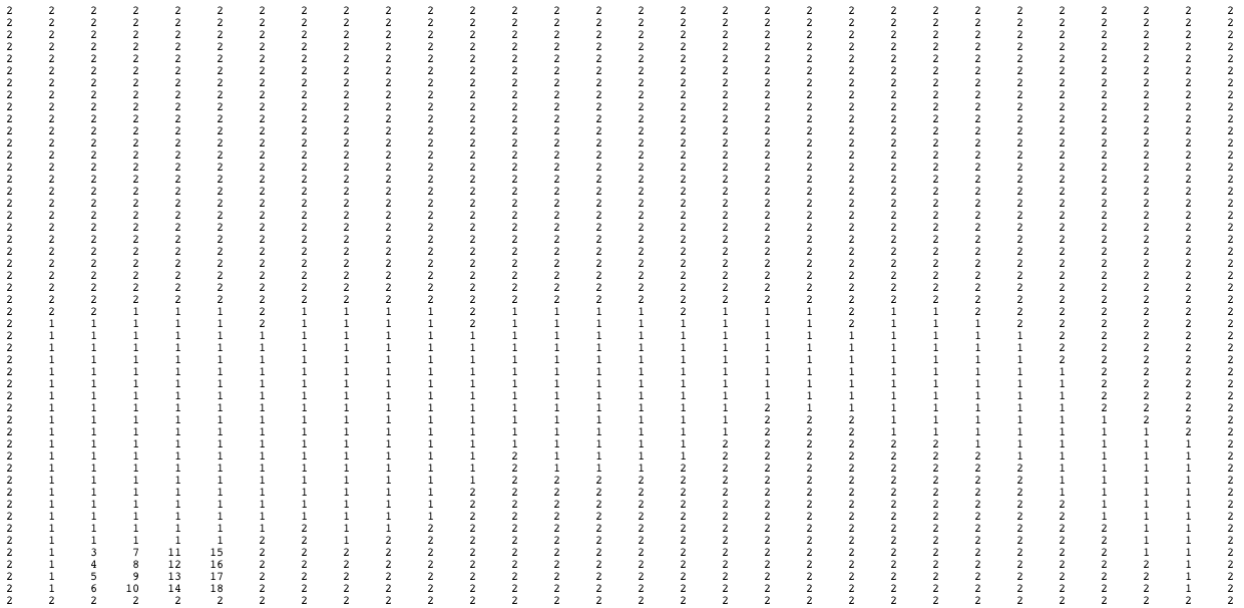

Figura 3.4: Representación numérica del escenario de aprendizaje sobre la que trabajan los algoritmos evolutivos. Los números "1" representan las celdas que corresponden al fondo del escenario y los números "2" representan los obstáculos del escenario. Los números mayores o iguales a "3" corresponden a los distintos módulos del robot en el caso de activación secuencial de los módulos. En el caso de activación aleatoria de los módulos, solo se emplean números "3". En ambos tipos de activación, la evaluación de las reglas considera a los módulos como "3", la identificación de los módulos es solo para efectos de orden de activación.

En el caso de las redes neuronales, estas aprovechan directamente la representación numérica del simulador para calcular una acción de salida. De esta forma, la red consta inicialmente de 8 nodos de entrada, que inspeccionan los valores de las celdas entorno al módulo activo (de valores 1,2 o 3), y 4 nodos de salida. Los nodos de salida entregan valores 0 o 1, dos asociados al movimiento vertical y dos al horizontal. El movimiento vertical queda determinado por la resta de 2 nodos de salida, donde un -1 representa bajar; un 0 no moverse; y un 1 es subir. Para determinar el movimiento horizontal se procede de forma equivalente.

La topología interna de las redes neuronales queda determinada por NEAT.

El simulador se genera con un algortimo de procesamiento de imágenes (anexos A.3 y B.3). Este algoritmo toma una figura bi-color y la traduce a la representación mencionada anteriormente, por ejemplo la figura 3.5. Por último, vale notar que el simulador sólo vela por que los movimientos

sean legítimos, es decir, que si un módulo se mueve, lo haga a una posición disponible.

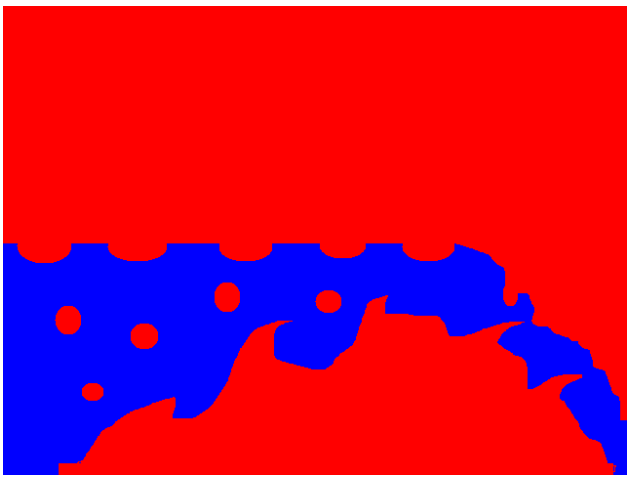

Figura 3.5: Escenario para el entrenamiento de algoritmos de locomoción. La parte roja de la figura representa los obstáculos del escenario. La parte azul representa el fondo.

3.1.2. Función Objetivo

Una parte fundamental del trabajo es establecer qué es lo que se quiere que el robot modular logre. En este caso, se definió que se quería es que el sistema se moviese hacia la derecha lo más posible, y que durante este movimiento el robot se mantenga conexo. Además, hay que resolver cómo medir esto, de forma tal que los algoritmos evolutivos manejen alguna medida de si se están logrando o no estos objetivos.

Esto se resolvió con una función objetivo que consta de dos partes. La primera mide el desplazamiento del robot, promediando la posición de todos los módulos al final de la simulación. La segunda parte mide la conectividad del robot, lo que se logra promediando el porcentaje de módulos conexos (en relación al total de módulos) através de toda la simulación.

De esta forma, la función de fitness para el individuo i queda como la multiplicación de estos dos factores:

$$f_i = \frac{\text{distancia promedio}}{\text{largo escenario}} \times \frac{\text{módulos conexos promedio}}{\text{total de módulos}}$$

Notar que la función de fitness, así definida, queda acotada por 0 y 1. Por otra parte, se definió que el escenario de aprendizaje sería el de la figura 3.5. Esto pues que, para maximizar el fitness, cada algoritmo en competencia debe aprender a moverse en todas las direcciones.

De esta forma cada individuo, evolucionado por los algoritmos evolutivos, es entregado al simulador como argumento. Luego se corre la simulación, desplazando al robot según el algoritmo. Finalmente, el simulador retorna un valor de fitness para cada individuo, según la función ya mencionada.

3.1.3. Algoritmos Evolutivos y Parámetros

Los algoritmos evolutivos empleados son el Algoritmo Genético Simple (anexo A.2), propuesto por D. Goldberg en [21], y Neuro Evolution of Augmenting Topologies, desarrollado por K. Stanley en [22].

El algoritmo genético empleado tiene como operadores la selección por ruleta y ranking, el crossover de uno o dos puntos, la mutación y el escalamiento del fitness (explicados extensamente en Antecedentes).

La probabilidad de crossover se fijó en 0.8, mientras que la probabilidad de mutación quedó variable en función del número de bits, de tal forma que la probabilidad fuese de modificar un solo bit por individuo (siguiendo recomendaciones en [21]). El crossover quedó establecido como de dos puntos. El método de selección se definió como de ruleta. La población se fijó de 30 individuos, definido según experiencias preliminares con problemas de calibración. El algoritmo evoluciona como máximo por 10000 generaciones, deteniéndose antes solo si no hay mejoras por más de 1000 generaciones (estancamiento).

El algoritmo NEAT es una adaptación del experimento XOR para Matlab disponible en la página web del proyecto[1]. Este código fue modificado en la parte en que se asignan los fitness a cada individuo (anexo B.2).

Se mantuvieron las probabilidades de los operadores genéticos del código original, así como también el tamaño de la población (150 individuos). Los parámetros que se modificaron son el número de generaciones, el estancamiento y el umbral de especiación. El número de generaciones

[1] http://www.cs.ucf.edu/~kstanley/neat.html

fue aumentado de 100 a 1000, sin ningún criterio de estancamiento. Esto fue establacido pues, en experimentos tempranos, se vió que para 100 generaciones aún se estaba en una etapa temprana de aprendizaje (gradiente elevado del fitness).

El estancamiento corresponde al número de generaciones en las que una especie no muestra mejoras en su fitness, tras lo que se hace su fitness igual a cero, de forma tal que la especie sea descartada por la selección. Este parámetro se igualó al número de generaciones de la evolución total, de forma tal que nunca se activara.

El umbral de especiación corresponde al valor para el cual se determina que dos individuos pertenecen o no a una misma especie (ecuación 2.3). Este valor se subió de 3 a 10, puesto que el operador de elitismo solo se activa para especies de más de cinco individuos. Con el umbral 3 original se generan muchas especies de menos de 5 individuos, por lo que nunca se preservaban a los individuos con buen desempeño. Con un umbral 10, se generan menos especies de más individuos, preservando aquellos de buen fitness.

3.1.4. Parámetros de Simulación

En las primeras etapas del trabajo de prefirió respetar los principios de aleatoriedad en la activación de los módulos propuestos en [3, 4, 20]. Sin embargo, esta aleatoriedad introduce un problema grave en el proceso de aprendizaje, puesto que, en simulaciones independientes, cada individuo obtiene fitness distintos. Esto se traduce en que un mismo individuo puede tener fitness bueno o malo, dependiendo de una simulación en particular.

Por esto, se decidió implemetar un modelo de activación secuencial, donde los módulos se activan siempre en el mismo orden, pero sin alterar la evaluación de estos. Es decir, al momento de calcular o aplicar reglas, el simulador ve siempre módulos anónimos.

En NEAT es obligatoria esta modificación, mientras que en algoritmos genéticos es alternativo. Esto pues en esta implementación de algoritmos genéticos se define al mejor individuo como aquel que en alguna generación obtuvo un alto puntaje (se considera que si obtuvo un alto puntaje alguna vez, debe tener material genético rescatable). Por otro lado, en la implementación de NEAT solo ven los fitness de la generación actual.

De lo anterior se definen 3 casos para el análisis final:

- Evolución de reglas implícitas (redes neuronales) con NEAT y activación secuencial.
- Evolución de reglas explícitas con algoritmos genéticos y activación aleatoria.
- Evolución de reglas explícitas con algoritmos genéticos y activación secuencial.

En los casos de reglas explícitas hay que definir, además, el número de reglas. Para esto se tomó como referencia al mayor número de reglas mencionado en [3, 4, 20], que es 22 reglas. Con esto, se definieron experiencias para optar por 5, 15, 30 o 45 reglas. Cada número de regla es evolucionado 3 veces para definir cual es la configuración que obtiene mejores resultados para algoritmos genéticos con activación aleatoria y secuencial.

En el caso con NEAT esto no es necesario, puesto que no hay parámetros de este tipo que definir en reglas implícitas.

3.1.5. Caracterización

Una vez elegidos los parámetros mencionados en las secciones anteriores, se procede a la evolución final de individuos. Para esto, en cada caso se realizan 5 rondas de aprendizaje independientes, lo que se garantiza acoplando el generador de números aleatorios con el reloj del computador. Esto con el fin de caracterizar el proceso de aprendizaje y analizar su variabilidad.

Se elige al mejor individuo de cada escenario y se procede a su caracterización. Esto es:

- Forma Inicial: Analizar el comportamiento del individuo para distintas configuraciones iniciales (forma inicial del robot).
- Escalamiento: Ver el comportamiento de los algoritmos para distintas escalas de simulación (escalabilidad del algoritmo).

La evolución de cada caso se realiza en el escenario de entrenamiento (figura 3.5), cuyas dimensiones son 30 celdas de alto y 50 celdas de largo. Las dimensiones del robot son 4x4 módulos.

La caracterización se realiza tanto en el entorno de entrenamiento como en un escenario de prueba (figura 3.6). Para el análisis de forma inicial se consideran robots con configuraciones iniciales de 4x4, 3x5, 5x3, 2x8 y 8x2 módulos, sobre un escenario de 30x50.

Para la experiencia de escalabilidad se considera una configuración inicial del robot de 4x4 módulos sobre un escenario de 30x50. Luego, estos valores se duplican, quintuplican y decuplican (las dimensiones se presentan en la tabla 4.2).

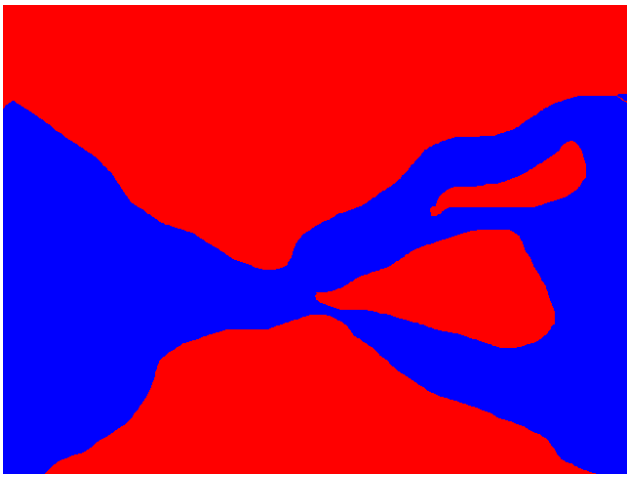

Figura 3.6: Escenario alternativo para la caracterización de los algoritmos. La parte roja de la figura representa los obstáculos del escenario. La parte azul representa el fondo.

3.2. Software y Equipo

El software fue desarrollado íntegramente en el programa Matlab 7.10.0.499 (R2010a) 64-bit.

El equipo usado para desarrollar las rondas de aprendizaje consta de los siguentes componentes:

- OS: Linux 2.6.32-35-generic Ubuntu 10.04 Lucid x86_64 GNU/Linux.
- CPU: Intel(R) Core(TM) i7-2600K CPU 3.40GHz.
- Memoria: 16GB RAM.
- Video: Nvidia GeForce GTX 480, 1536 MB.
- Placa Madre: ASUSTeK Computer Inc., modelo P8H67-M PRO.

Capítulo 4

Resultados

En la presente sección se presentan los resultados de las experiencias de obtención y caracterización de algoritmos para locomoción de robots modulares reconfigurables. Esto para los algoritmos generados con NEAT, Algoritmos Genéticos con activación aleatoria y secuencial de los módulos.

La caracterización de los algoritmos consta de dos partes, análisis de invarianza a la forma inicial del robot y análisis de invarianza a la escala de simulación. Ambos estudios se realizan tanto en el escenario de entrenamiento como en el de prueba (figuras 3.5 y 3.6). Con esto, además se puede estudiar la dependencia de las experiencias con el escenario de ensayo.

El estudio de forma inicial se lleva a cabo modificando la estructura inicial de los módulos del robot según la tabla 4.1. Por otra parte, las dimensiones para el estudio de escalamiento se muestran en la tabla 4.2, donde cada configuración corresponde al doble, quíntuple y décuple de las dimensiones iniciales.

Tabla 4.1: Dimensiones del robot para el estudio de invarianza a la forma inicial. El tamaño del escenario se mantiene en 30x50 pixeles.

Shape Index	**Robot Height**	**Robot Width**	**Symbol**
i	4	4	■
ii	3	5	▬
iii	5	3	▮
iv	2	8	▬
v	8	2	▮

Tabla 4.2: Dimensiones para el estudio de invarianza a la escala. Cada caso corresponde a 1×, 2×, 5× y 10× del tamaño original.

Scale Index	Scenario Height	Scenario Width	Robot Height	Robot Width
i	30	50	4	4
ii	60	100	8	8
iii	150	250	20	20
iv	300	500	40	40

Las dimensiones de los escenario de ensayo y módulos del robot corresponden a pixeles, que a su vez corresponden a celdas con valores numéricos interpretados por el simulador (figura 3.4).

4.1. NEAT

A continuación se presentan los resultados obtenidos con NEAT, donde la activación de los módulos del robot es secuencial.

4.1.1. Curva de Aprendizaje

En la figura 4.1 se muestra la curva de aprendizaje para 5 rondas independientes de entrenamiento con NEAT en 1000 generaciones de evolución. En la última generación se tiene un fitness medio de 0.7080 y una desviación estándar de 0.0147. La desviación estándar promedio através de las generaciones es de 0.0193. La curva de aprendizaje llega al fitness medio máximo en la 772^{va} generación.

Los fitness finales de cada ronda de aprendizaje se presentan en la tabla 4.3. De aquí se selecciona al mejor individuo de la tercera ronda de aprendizaje para ser caracterizado.

Tabla 4.3: Fitness finales de las 5 rondas de aprendizaje con NEAT.

Learning Round	Final Fitness
1^{st}	0.7110
2^{nd}	0.7106
3^{rd}	0.7297
4^{th}	0.6943
5^{th}	0.6941

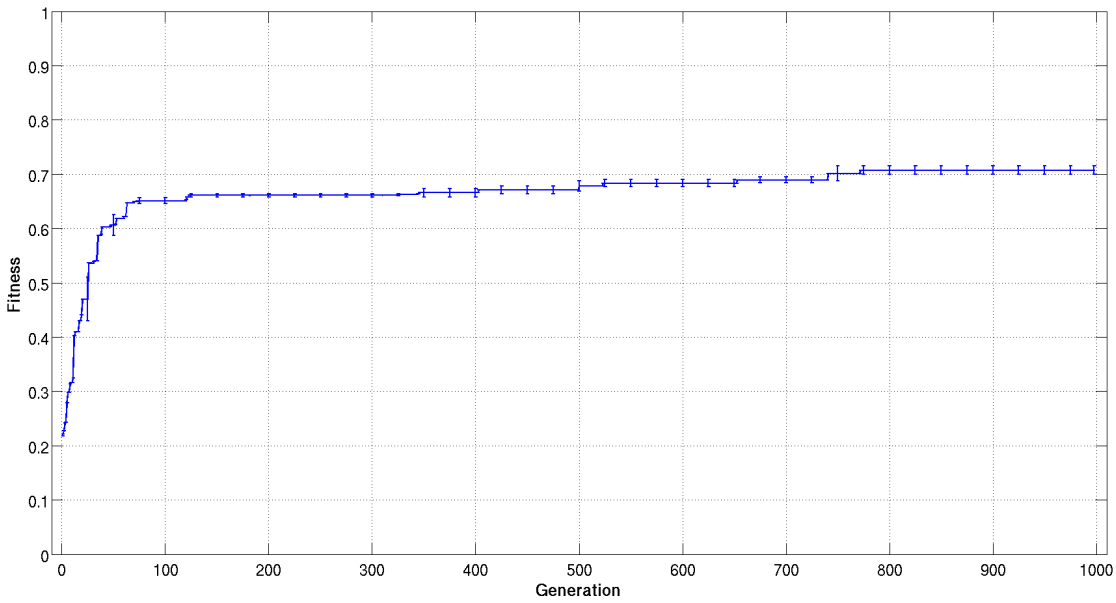

Figura 4.1: Curva de aprendizaje para 5 rondas de entrenamiento con NEAT. El fitness medio final es de 0.7080.

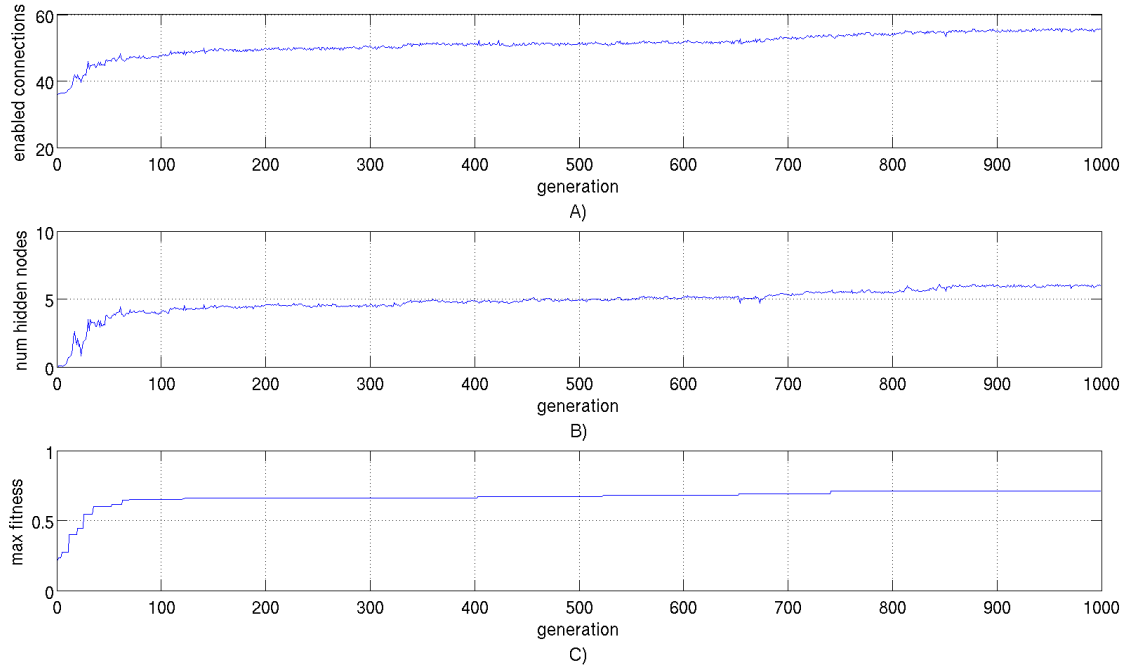

Figura 4.2: Detalle de la evolución de la topología en la tercera ronda de aprendizaje con NEAT. A) Número promedio de conexiones activas en la población. B) Número promedio de nodos escondidos en la población. C) Evolución del fitness máximo a través de las generaciones.

Cualitativamente se ve que la curva deja la zona de aprendizaje rápido antes de la 100^{va} generación. Antes de esta generación, la desviación estándar media, entre las curvas de aprendizaje, es de 0.0257. Luego, la desviación estándar media se reduce a 0.0185.

En la figura 4.2 se muestra, para la tercera ronda de aprendizaje con NEAT, la evolución del número medio de conexiones sinápticas activas en la población de redes neuronales. También se muestra el número promedio de nodos de las capas escodidas de las redes neuronales y la evolución del fitness máximo de la población.

4.1.2. Caracterización

A continuación se presentan los resultados de la caracterización del mejor individuo evolucionado en la tercera ronda de entrenamiento con NEAT.

Invarianza Forma Inicial

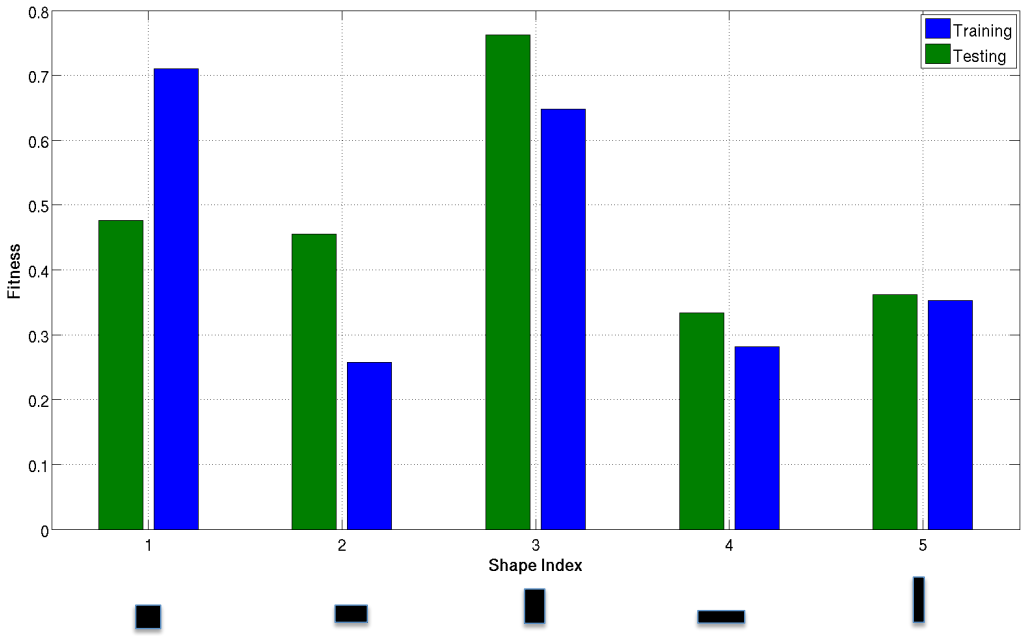

Figura 4.3: Fitness obtenidos en los escenarios de entrenamiento y prueba para la experiencia de forma inicial del robot. Los fitness para cada configuración inicial, en el escenario de entrenamiento, son 0.7106, 0.2572, 0.6477, 0.2812 y 0.3526. En el escenario de prueba, los fitness finales son 0.4758, 0.4551, 0.7620, 0.3335 y 0.3616.

En la figura 4.3 se muestran los fitness obtenidos por el individuo para las configuraciones indicadas en la tabla 4.1. La primera configuración corresponde a la misma en la que se realizó el

entrenamiento, por lo que el fitness coincide.

Los fitness obtenidos en el escenario de prueba, para las distintas configuraciones de forma inicial, se muestran en la figura 4.3. Sin considerar la primera configuración, se ve que en ambos escenarios se obtiene el mayor fitness con la tercera configuración (robot con configuración inicial de 5x3 módulos). Esta configuración consigue un fitness incluso más alto en el escenario de prueba que en el de entrenamiento.

Para las distintas configuraciones, el fitness medio en el escenario de entrenamiento es de 0.4499 y la desviación estándar es de 0.2134. El fitness medio en el escenario de prueba es de 0.4776 y la desviación estándar es de 0.1700.

En el escenario de entrenamiento, la razón entre la desviación estándar y fitness medio es de 0.4743. En el escenario de prueba, esta razón es de 0.3560.

Invarianza Escala

En la figura 4.4 se muestran los fitness obtenidos en el escenario de entrenamiento y prueba, respectivamente, para las dimensiones indicadas en la tabla 4.2.

Para el escenario de entrenamiento el fitness medio y desviación estándar son, respectivamente 0.5499 y 0.1073. En el escenario de prueba se tiene un fitness medio de 0.5707 y una desviación estándar de 0.0969.

La razón entre la desviación estándar y el fitness medio es de 0.1951 para el escenario de entrenamiento. Para el escenario de pruebas, esta razón es de 0.1698.

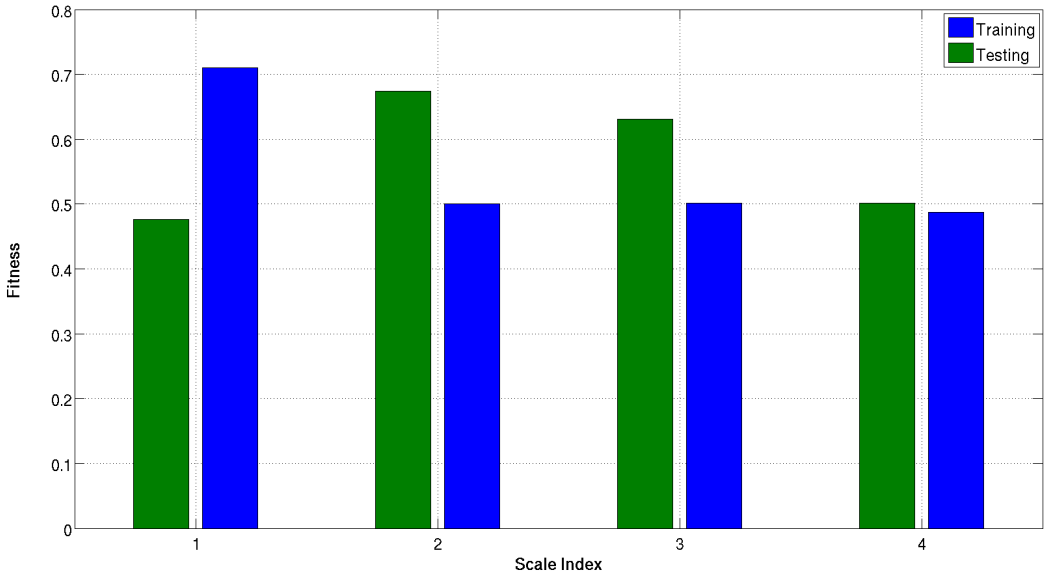

Figura 4.4: Fitness finales simulaciones en distintas escalas con NEAT en el escenario de entrenamiento y prueba. Para cada configuración, en el escenario de entrenamiento, los fitness son 0.7106, 0.5003, 0.5015 y 0.4872. En el escenario de prueba, los fitness finales son 0.4758, 0.6744, 0.6311 y 0.5016.

4.1.3. Simulación

A continuación se muestra una secuencia de instantáneas de la simulación en la primera configuración de la tabla 4.1, en los escenarios de entrenamiento (figura 4.4) y prueba (figura 4.5).

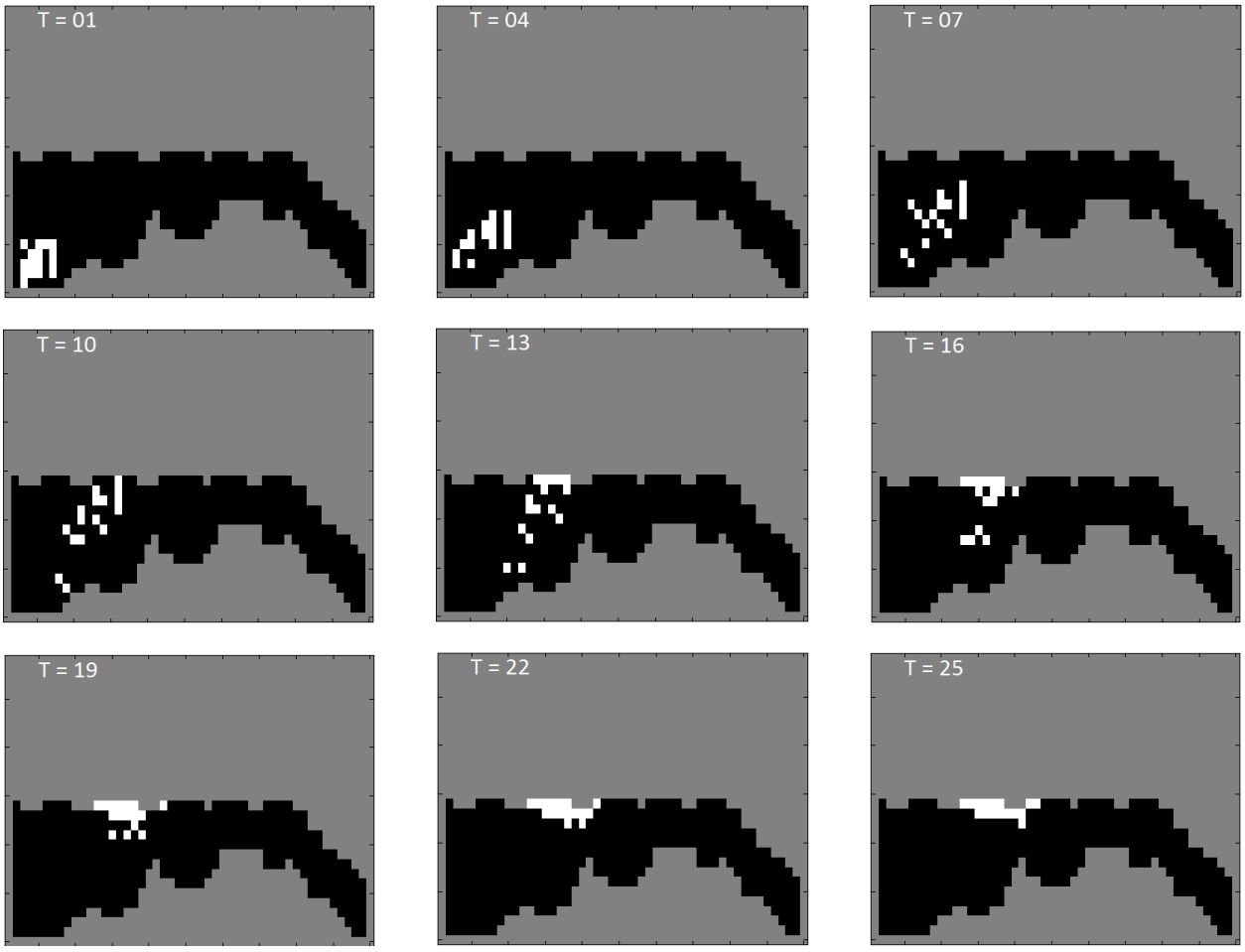

Figura 4.4: Instantáneas de la simulación en el escenario de entrenamiento con el mejor individuo evolucionado con NEAT.

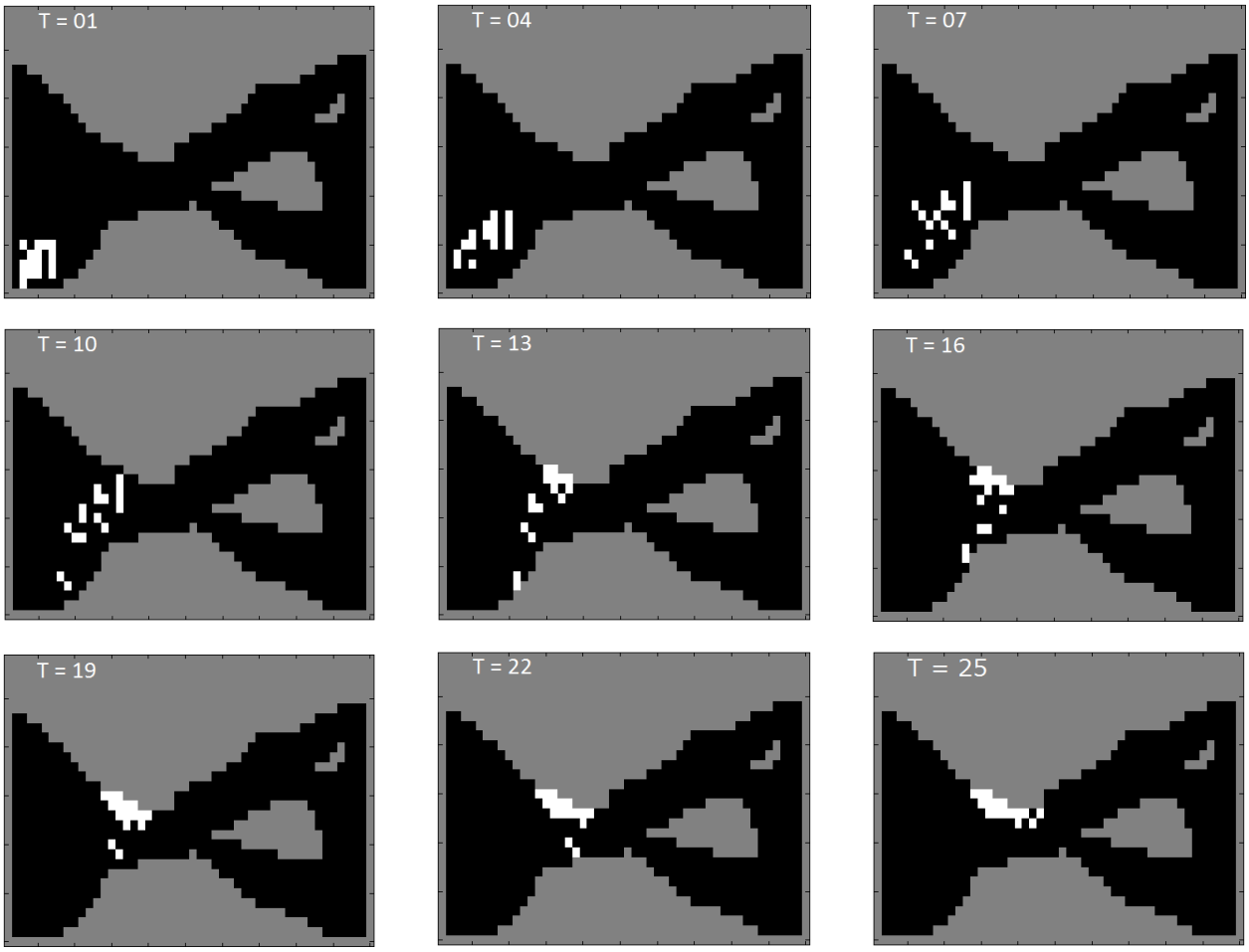

Figura 4.5: Instantáneas de la simulación en el escenario de prueba con el mejor individuo evolucionado con NEAT.

4.2. Algoritmos Genéticos con Activación Aleatoria

A continuación se muestran los resultados obtenidos con algoritmos genéticos y activación aleatoria de los módulos.

4.2.1. Número de Reglas y Curva de Aprendizaje

En la figura 4.6 se muestran los fitness finales para 3 rondas de aprendizaje para 5, 15, 30 y 45 reglas. De aquí se elige trabajar con 30 reglas, puesto que este número de reglas tienen el mayor fitness y la menor desviación estándar en sus tres rondas de aprendizaje.

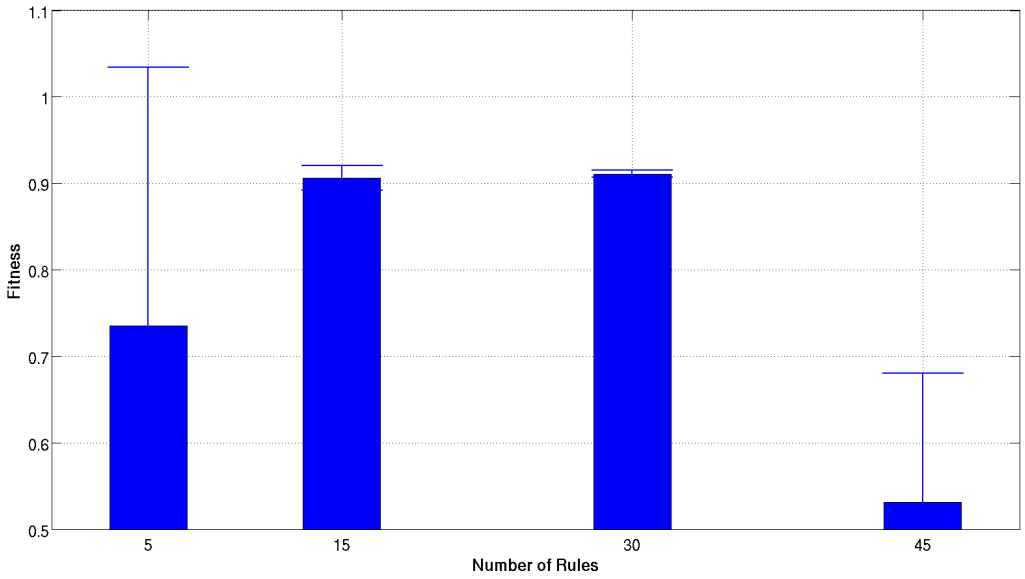

Figura 4.6: Fitness promedio y desviación estándar para distinto número de reglas. Para cada caso se realizaron 3 rondas de entrenamiento independientes. El fitness medio, para cada caso, es de 0.736, 0.906, 0.910 y 0.531.

En la figura 4.7 se muestra la curva de aprendizaje para 5 rondas independientes con 30 reglas. En la última generación se tiene un fitness medio de 0.8603 y una desviación estándar de 0.1138. La desviación estándar media através de las generaciones es de 0.1176. La curva de aprendizaje alcanza su fitness medio máximo en la 2803^{va} generación.

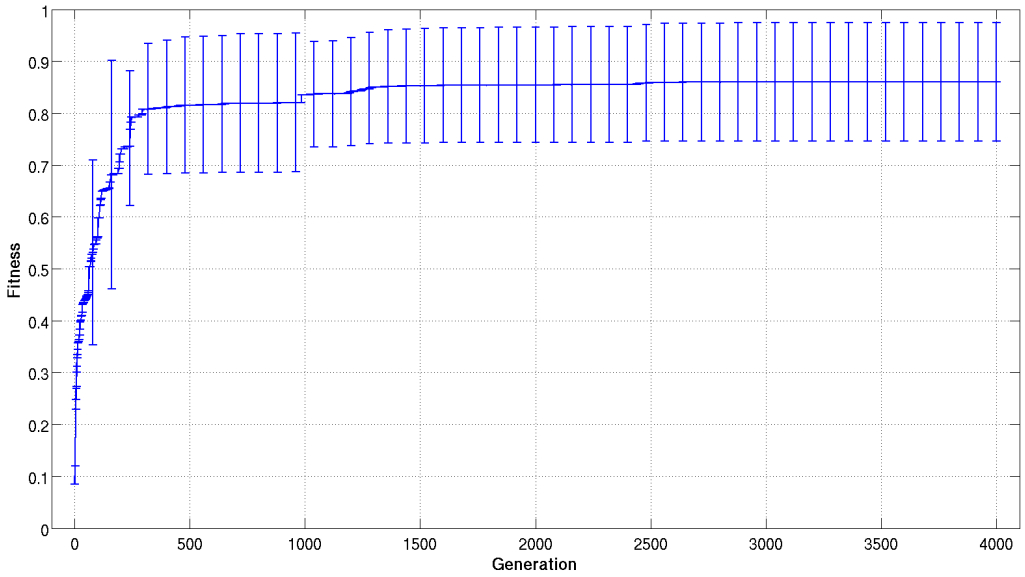

Figura 4.7: Curva de aprendizaje para 5 evoluciones independientes con algoritmos genéticos. El fitness medio final es de 0.8603.

Los fitness finales de las 5 rondas de aprendizaje con 30 reglas se prensentan en la tabla 4.4. De aquí se elige al mejor individuo evolucionado en la tercera ronda para ser caracterizado.

Tabla 4.4: Fitness finales de las 5 rondas de aprendizaje con Algoritmos Genéticos y activación aleatoria.

Learning Round	Final Fitness
1^{st}	0.9079
2^{nd}	0.9089
3^{rd}	0.9158
4^{th}	0.6568
5^{th}	0.9123

Cualitativamente, la curva de aprendizaje deja la zona de aprendizaje rápido en la 500^{va} generación. Antes de este punto, la desviación estándar media es de 0.1397. Luego, la desviación estándar baja a 0.1145.

4.2.2. Caracterización

A continuación se presentan los resultados de la caracterización del individuo evolucionado en la tercera ronda de aprendizaje con Algoritmos Genéticos y activación aleatoria.

Invarianza Forma Inicial

En la figura 4.8 se muestran los resultados obtenidos en el escenario de entrenamiento y prueba para las configuraciones indicadas en la tabla 4.1.

Para el escenario de entrenamiento, el fitness medio obtenido en las distintas configuraciones es de 0.5436, y la desviación estándar de los fitness es de 0.0456. El promedio de los errores de las 5 configuraciones es de 0.0481.

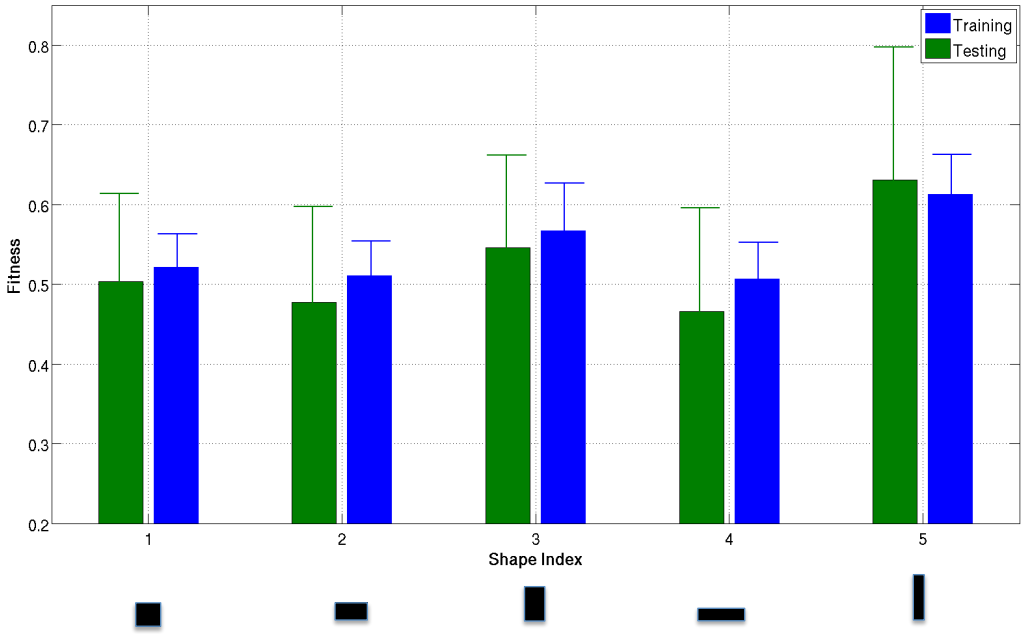

Figura 4.8: Resultados del ensayo de variación de la forma inicial del robot en el escenario de entrenamiento. Los fitness medios para cada configuración son 0.5215, 0.5105, 0.5668, 0.5062 y 0.6129. En el escenario de prueba los fitness medios para cada configuración son 0.5031, 0.4775, 0.5458, 0.4657 y 0.6310. Cada configuración fue probada 100 veces.

En el escenario de prueba, el fitness medio para las distintas configuraciones es de 0.5246. La desviación estándar de los fitness es de 0.0669. El promedio de los errores de las 5 configuraciones

es de 0.1287.

La razón entre la desviación estándar promedio y el fitness promedio en el escenario de entrenamiento, entre las distintas configuraciones, es de 0.0885. En el escenario de prueba, esta razón es de 0.2454.

En ambos escenarios se tiene el mismo orden de resultados. Es decir, tanto en el escenario de entrenamiento como en el de prueba el mejor fitness se obtiene en la 5^{ta} configuración, seguidos de la 3^{ra}, la 1^{ra}, la 2^{da} y la 4^{ta}. Esto permite concluir el desempeño, para distintas formas iniciales del robot, no depende del escenario.

Invarianza Escala

En las figuras 4.9 se muestran los resultados obtenidos para los escenarios de entrenamiento y prueba. Esto según las dimensiones señaladas en la tabla 4.2.

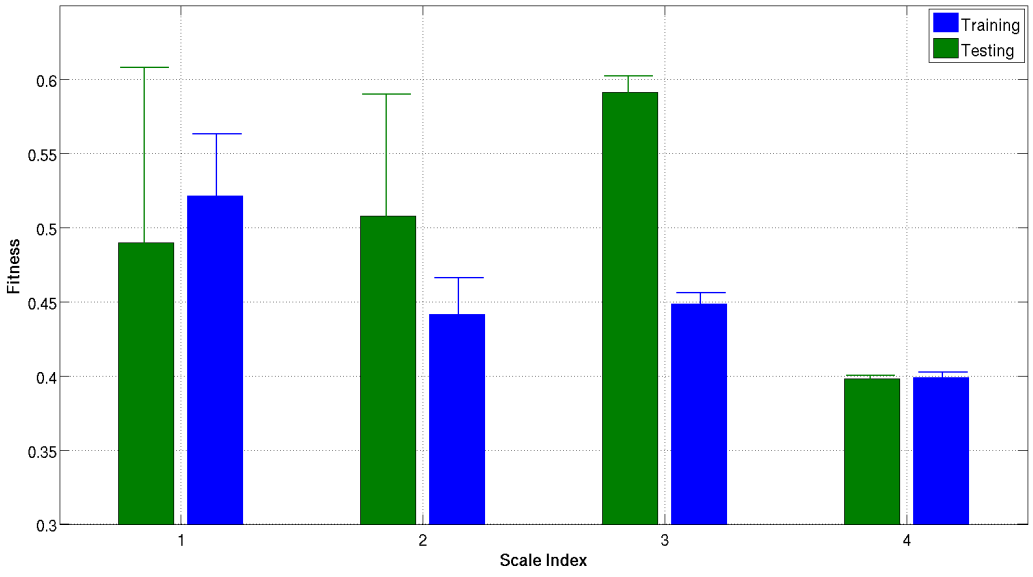

Figura 4.9: Fitness finales obtenidos en el escenario entrenamiento y prueba para la experiencia de variación de la escala de simulación. Los fitness medios para cada configuración, en el escenario de entrenamiento, son 0.5217, 0.4418, 0.4486 y 0.3990. En el escenario de prueba, los fitness finales son 0.4901, 0.5079, 0.5915 y 0.3983. Cada configuración fue probada 100 veces.

Para las distintas configuraciones, el fitness medio obtenido en el escenario de entrenamiento es

de 0.4528, con una desviación estándar de 0.0509. La desviación estándar promedio es de 0.0194.

En el escenario de prueba, el fitness promedio para las distintas configuraciones es de 0.4970, con una desviación estándar de 0.0792. La desviación estándar promedio es de 0.0532.

La razón entre el error medio y el fitness medio en el escenario de entrenamiento es de 0.0428. En el escenario de prueba, esta razón es de 0.1071.

4.2.3. Simulación

A continuación se muestran secuencias de instantáneas de las simulaciones en la primera configuración de la tabla 4.1. Esto para los escenarios de entrenamiento (figura 4.10) y prueba (figura 4.11).

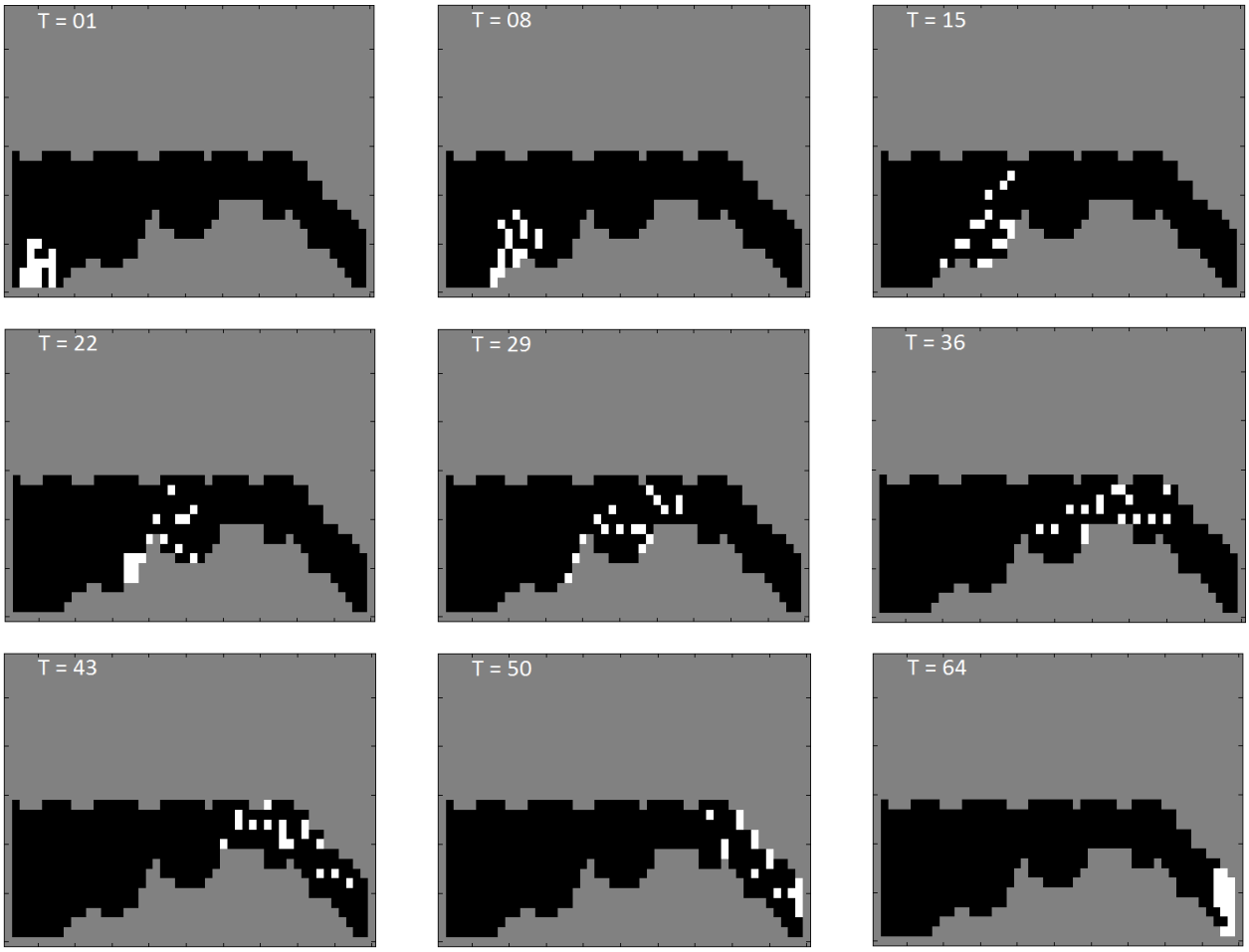

Figura 4.10: Instantáneas de la simulación en el escenario de entrenamiento con el mejor individuo obtenido con Algoritmos Genéticos y activación aleatoria.

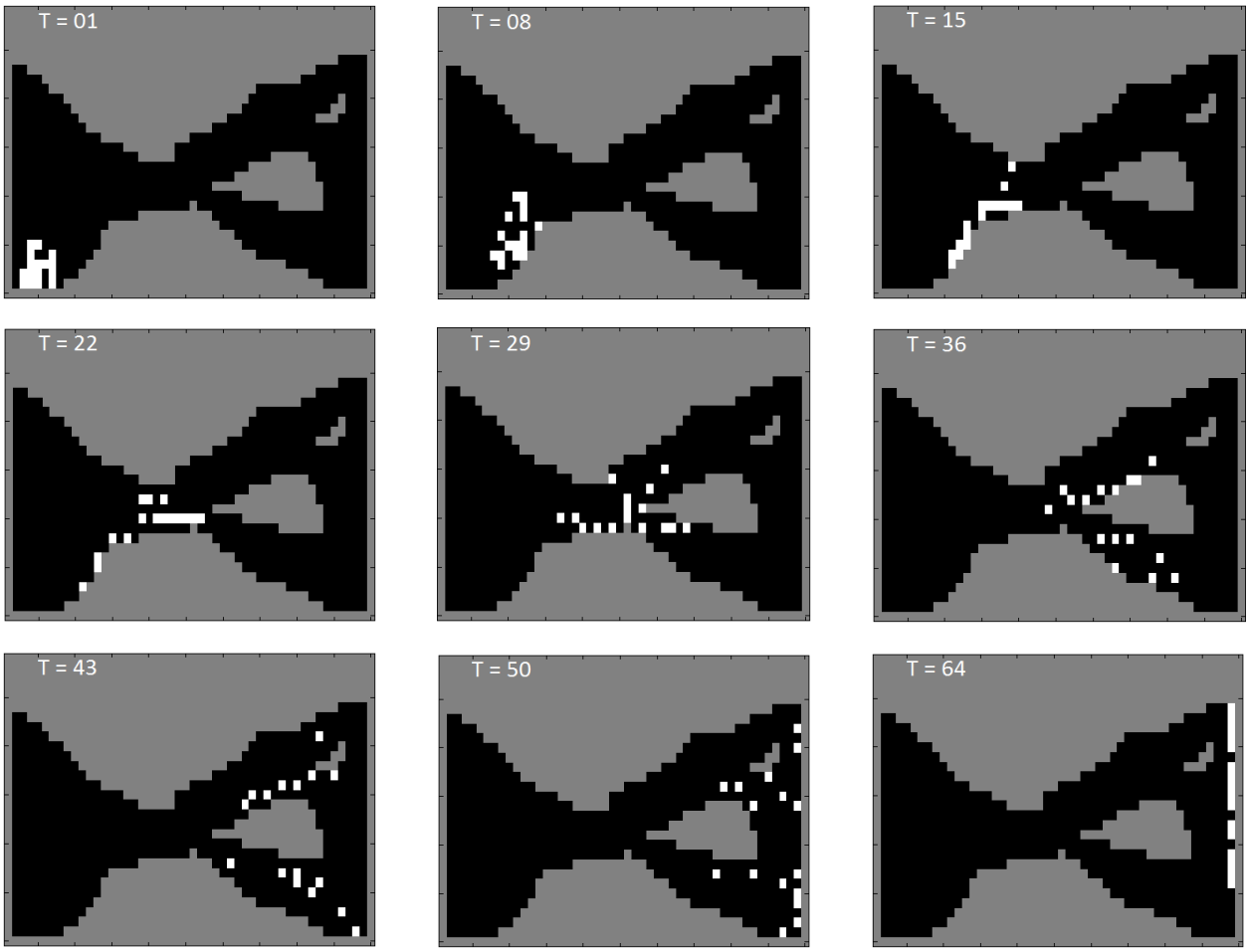

Figura 4.11: Instantáneas de la simulación en el escenario de prueba con el mejor individuo obtenido con Algoritmos Genéticos y activación aleatoria.

4.3. Algoritmos Genéticos con Activación Secuencial

A continuación se muestran los resultados obtenidos con algoritmos genéticos y activación secuencial de los módulos del robot.

4.3.1. Número de Reglas y Curva de Aprendizaje

En la figura 4.12 se muestran los fitness finales para 3 rondas de aprendizaje con 5, 15, 30 y 45 reglas. De aquí se elige trabajar con 30 reglas, puesto que es el parámetro que obtuvo el mayor fitness final. Además, esta selección facilita la comparación con el caso de activación aleatoria.

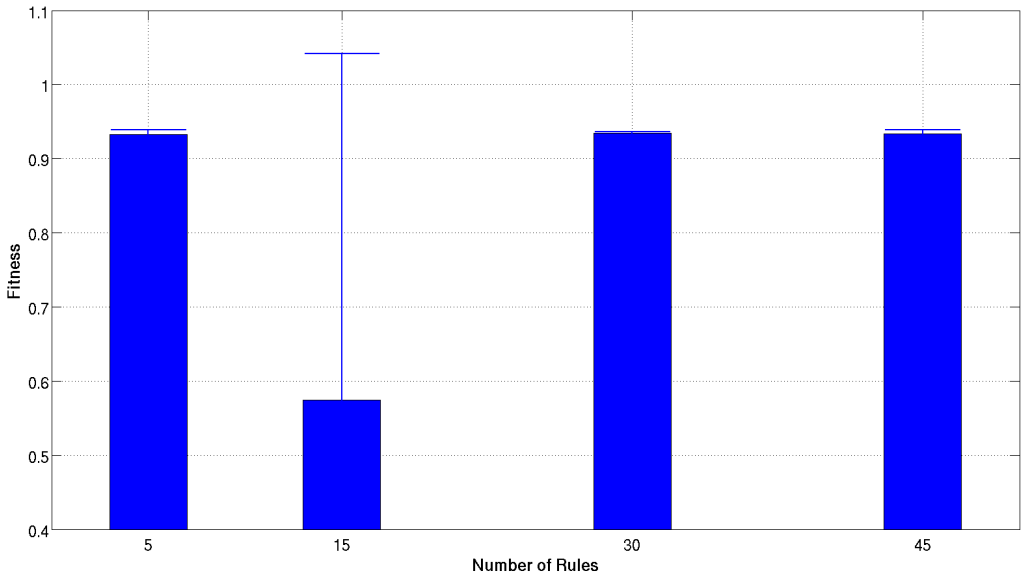

Figura 4.12: Fitness promedio y desviación estándar para distintos números de reglas. Para cada número de reglas se realizaron 3 rondas de entrenamiento independientes. El fitness medio, para cada caso, es de 0.9330, 0.5741, 0.9342 y 0.9336.

En la figura 4.13 se muestra la curva de aprendizaje de 5 rondas de entrenamiento independientes con 30 reglas. En la última generación se tiene un fitness medio de 0.8622 y una desviación estándar de 0.1540. La desviación estándar promedio, através de las generaciones, es de 0.1787. La curva de aprendizaje alcanza su máximo fitness medio en la 4053^{va} generación.

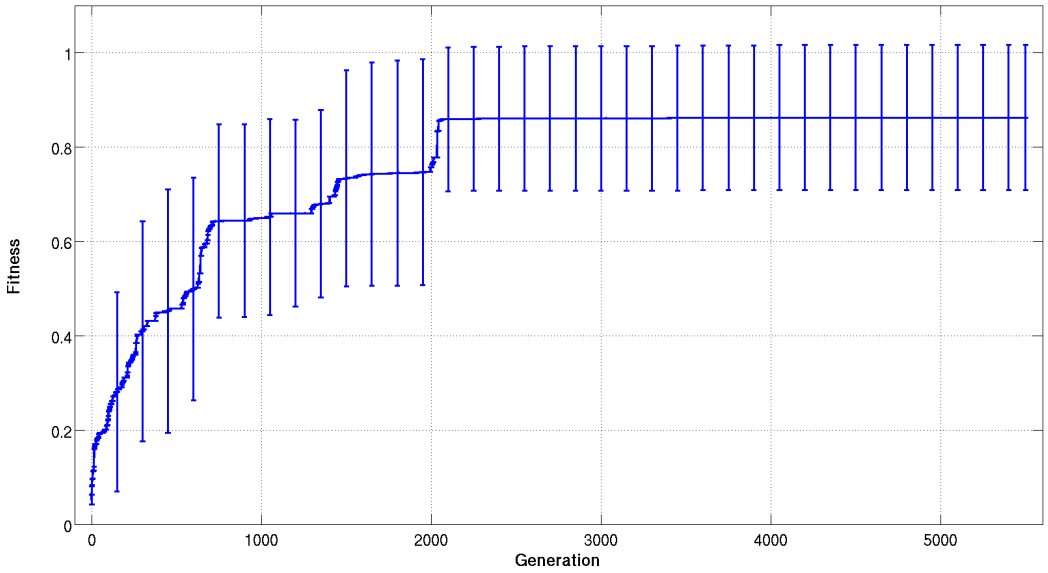

Figura 4.13: Curva de aprendizaje para 5 evoluciones independientes con algoritmos genéticos. El fitness medio final 0.8622.

Los fitness finales de las 5 rondas de aprendizaje se muestran en la tabla 4.5. De aquí se selecciona al mejor individuo evolucionado en la tercera ronda de aprendizaje para ser caracterizado.

Tabla 4.5: Fitness finales de las 5 rondas de aprendizaje con Algoritmos Genéticos y activación secuencial.

Learning Round	Final Fitness
1^{st}	0.9361
2^{nd}	0.9317
3^{rd}	0.9348
4^{th}	0.5869
5^{th}	0.9215

Cualitativamente se ve que la curva entra a la meseta de la curva de aprendizaje entorno a la 2000^{va} generación. Antes de este punto, la desviación estándar promedio es de 0.2221. Después de este punto, la desviación promedio baja a 0.1540.

4.3.2. Caracterización

A continuación se presentan los resultados de la caracterización del individuo evolucionado en la tercera ronda de aprendizaje con algoritmos genéticos y activación secuencial.

Forma Inicial

En la figura 4.14 se muestran los fitness finales obtenidos en el escenario de entrenamiento y prueba, con configuraciones según la tabla 4.1. En el escenario de entrenamiento, el fitness medio obtenido es de 0.9121, con una desviación estándar de 0.0122.

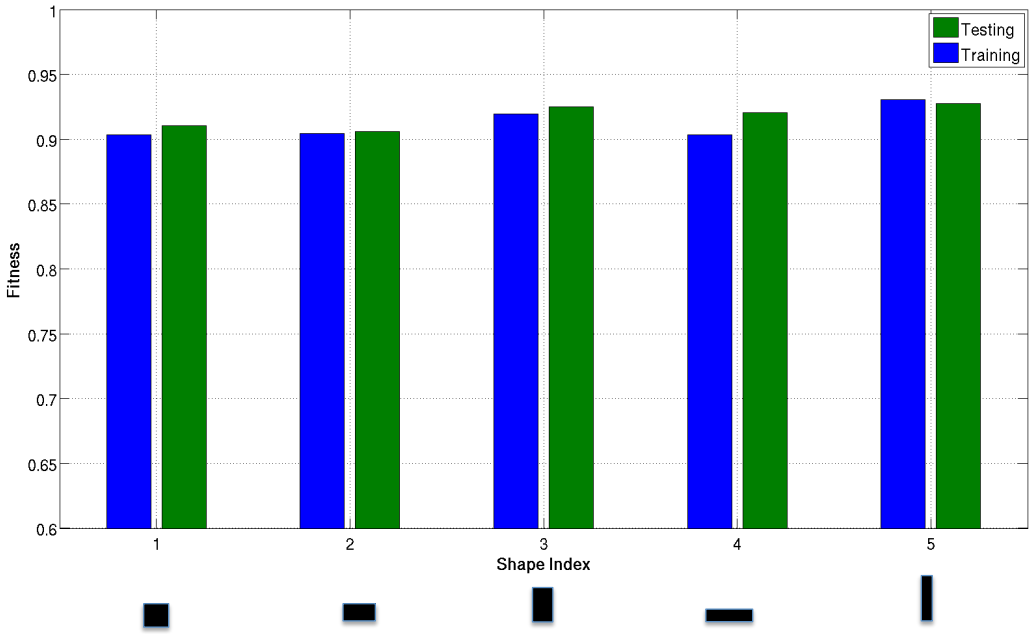

Figura 4.14: Resultados de modificar la forma inicial de robot en el escenario de entrenamiento y prueba. En el escenario de entrenamiento, los fitness son 0.9032, 0.9045, 0.9193, 0.9033 y 0.9303. Por otro lado, en el escenario de prueba, los fitness son 0.9106, 0.9057, 0.9250, 0.9206 y 0.9275.

Para el ensayo de forma inicial en el escenario de prueba, el fitness medio es de 0.9179, con una desviación estándar de 0.0094.

En el escenario de entrenamiento, la razón entre la desviación estándar promedio y el fitness promedio es de 0.0134. En el escenario de prueba esta razón es 0.0102.

En ambos escenarios se tienen los máximos valores para las configuraciones 5 y 3. En general, todos los fitness son cercanos al fitness obtenido en la evolución del individuo, por lo que se puede concluir que hay invarianza tanto al escenario como a la forma inicial del robot.

Invarianza Escala

En las figura 4.15 se muestran los resultados del ensayo de escala para los escenarios de entrenamiento y prueba, según la tabla 4.2.

El fitness medio obtenido en el escenario de entrenamiento es de 0.6241, con una desviación estándar de 0.2303. Por otra parte, en el escenario de prueba se obtuvo un fitness medio de 0.7655, con una desviación estándar de 0.1669.

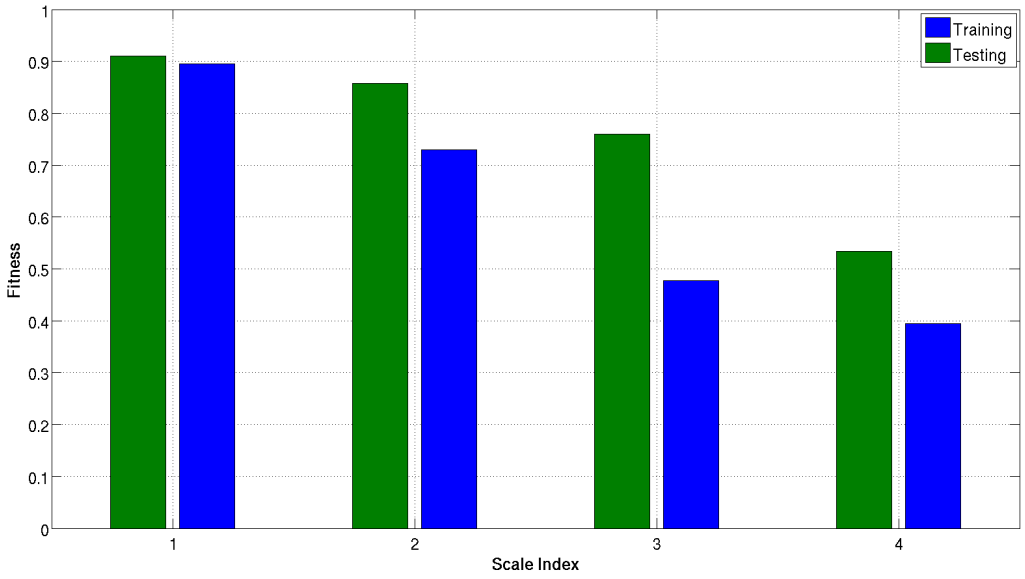

Figura 4.15: Resultados de modificar la escala de simulación en el escenario de entrenamiento y prueba. Para el escenario de entrenamiento, los fitness son 0.8957, 0.7292, 0.4768 y 0.3947. Por otro lado, para el escenario de prueba, los fitness son 0.9106, 0.8579, 0.7602 y 0.5333.

Se nota una relación inversa entre la escala y el fitness alcanzado en cada configuración. Como esta tendencia se da en ambos escenarios, se puede concluir que este comportamiento no depende el escenario donde se simule.

4.3.3. Simulación

A continuación se muestran instantáneas de las simulaciones en la primera configuración de la tabla 4.1. Esto para los escenarios de entrenamiento (figura 4.10) y prueba (figura 4.11).

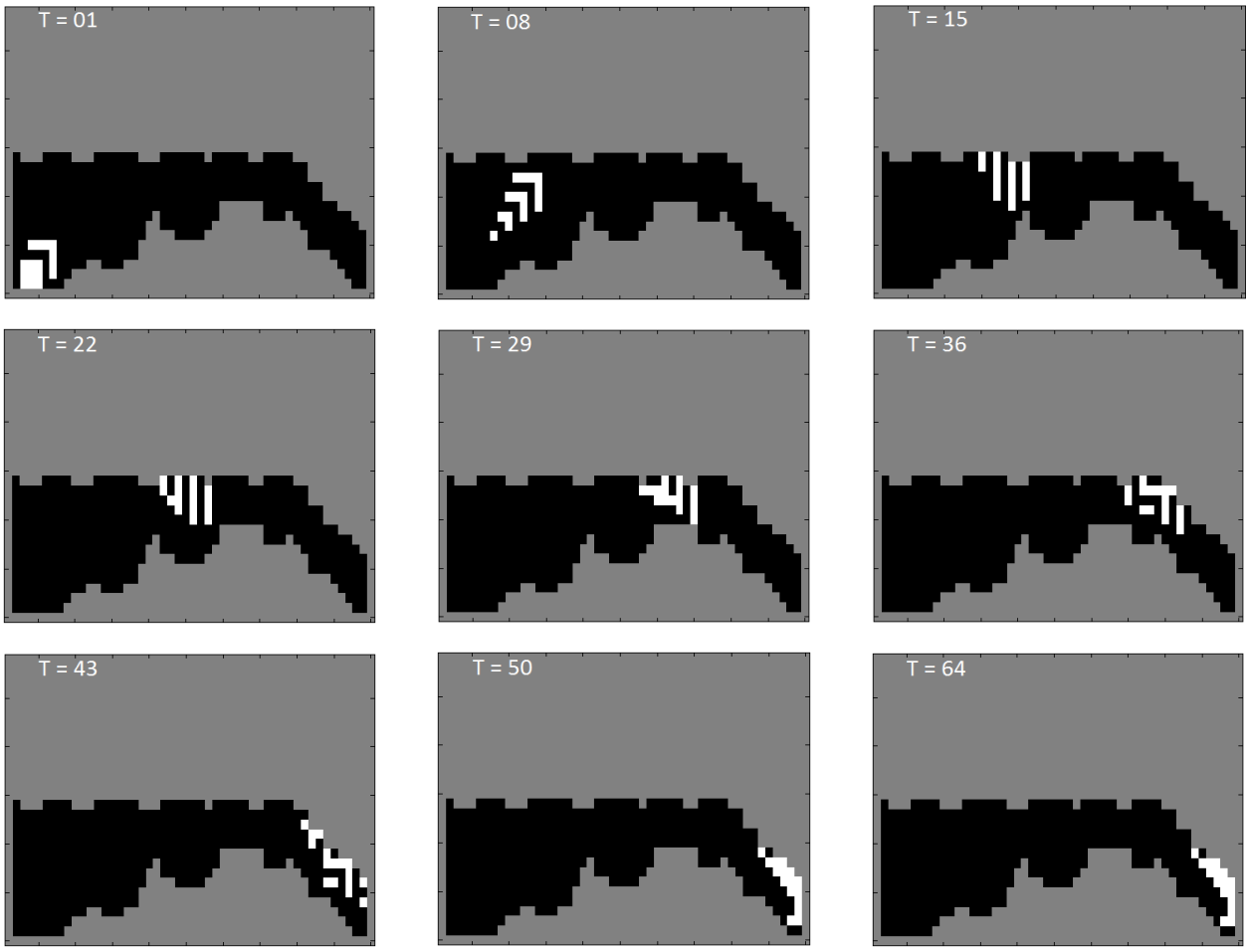

Figura 4.16: Instantáneas de la simulación en el escenario de entrenamiento con el mejor individuo obtenido con Algoritmos Genéticos y activación secuencial.

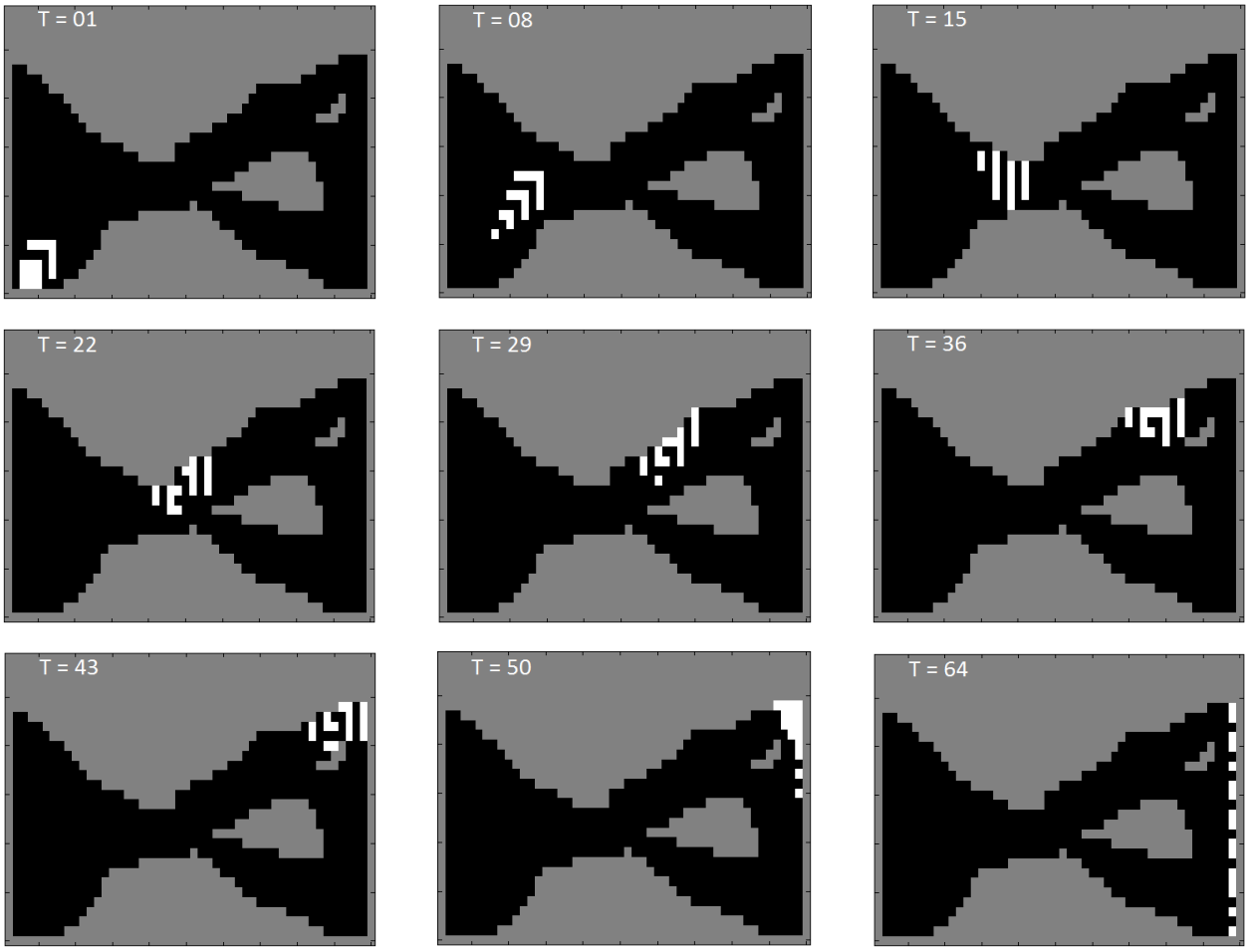

Figura 4.17: Instantáneas de la simulación en el escenario de prueba con el mejor individuo obtenido con Algoritmos Genéticos y activación secuencial.

Capítulo 5

Análisis y Discusión de Resultados

En la presente sección se muestra el análisis de los resultados obtenidos con las distintas estrategias de aprendizaje evolutivo. Se realiza un estudio sobre los resultados del proceso de obtención de las soluciones y de las experiencias de caracterización. Luego, se presentará una discusión global sobre los resultados obtenidos en relación al proceso de trabajo.

5.1. Análisis de Curvas de Aprendizaje

En las figuras 5.1 y 5.2 se presentan las curvas de aprendizaje obtenidas con los distintos algoritmos evolutivos en función de las generaciones e individuos evaluados. En estas figuras se puede apreciar que el proceso de aprendizaje con NEAT y algoritmos genéticos con activación aleatoria tiene una tasa de aprendizaje inicial alta, llegando rápidamente a la zona de meseta de aprendizaje (aprendizaje más lento). Por otro lado, la curva correspondiente a algoritmos genéticos con activación secuencial tiene un desarrollo mucho más extendido, lo que da cuenta de una mejor exploración del espacio de búsqueda.

En la tabla 5.1 se muestran distintas medidas de error de las evoluciones, junto con el número de la generación en la que se alcanzó el máximo fitness medio obtenido por cada método.

De los valores prensentados en la tabla 5.1 se puede ver que NEAT tiene el menor error medio asociado al proceso de aprendizaje, por lo que se puede concluir que es el método más repetible. Sin embargo, también es el método con el menor fitness final y el que alcanza el máximo fitness medio en el menor número de generaciones, por lo que se podrían atribuir estos valores a un problema de convergencia prematura. Esto último queda descartado al tratarse de 5 rondas de aprendizaje independientes.

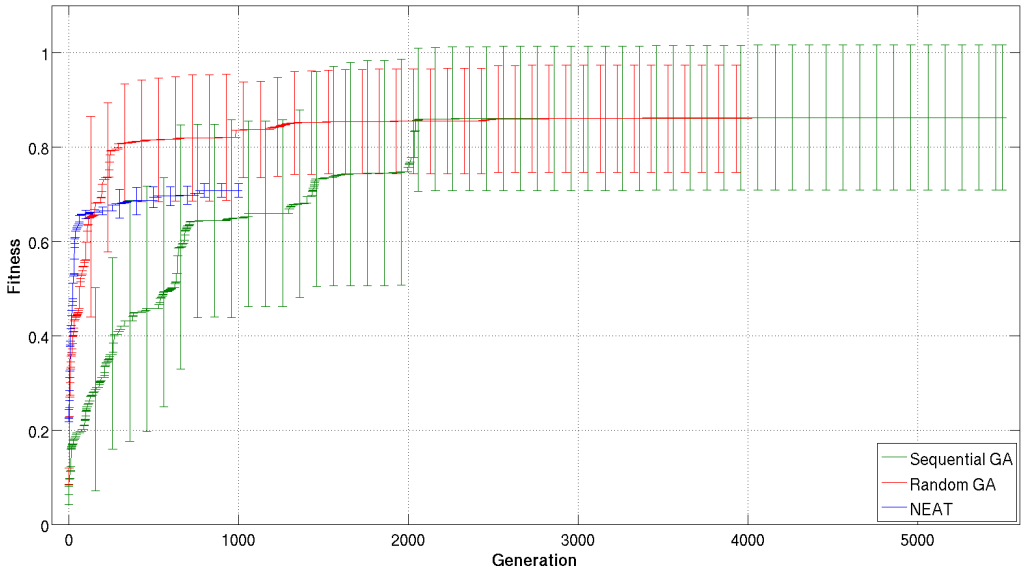

Figura 5.1: Curvas de aprendizaje obtenidas con 5 rondas independientes de entrenamiento con NEAT, Algoritmos Genéticos con activación aleatoria y secuencial de los módulos. Los fitness finales de cada método son 0.7080, 0.8603 y 0.8622, respectivamente.

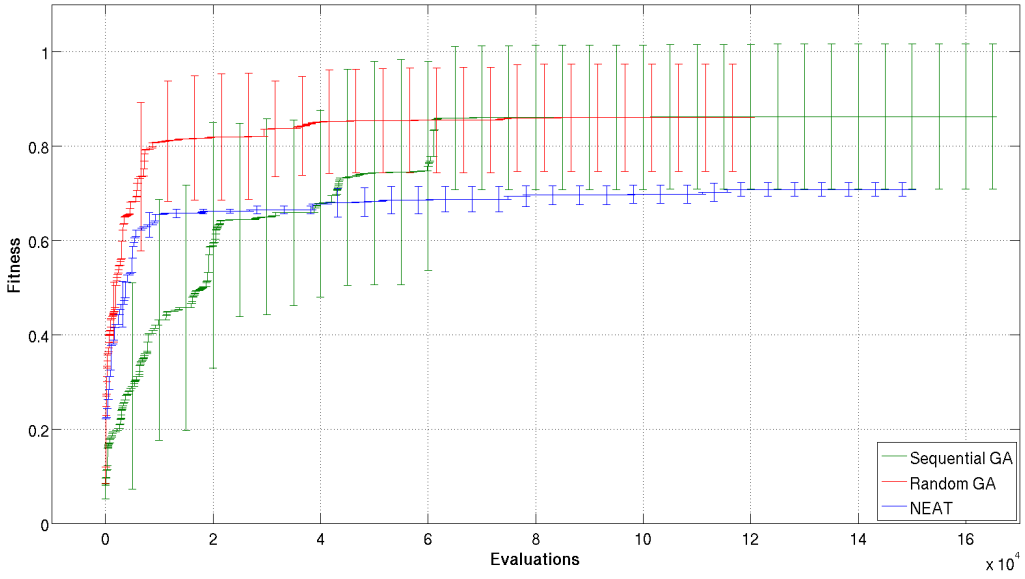

Figura 5.2: Curva de aprendizaje en función del número de evaluaciones. Esto es equivalente al número de individuos evaluados a través de las generaciones. En el caso de los métodos con algoritmos genéticos se emplearon 30 individuos, mientras que con NEAT se empleo 150.

Tabla 5.1: Errores promedio através de las generaciones, error final y generación en la que se alcanza el máximo fitness para las curvas de aprendizaje de NEAT, Algoritmos Genéticos con activación aleatoria y secuencial de los módulos.

Method	**Mean Error**	**Final Error**	**Max Fitness Generation**
NEAT	0.0193	0.0147	772
Random GA	0.1176	0.1138	2803
Sequential GA	0.1787	0.1540	4053

Es importante notar que el error asociado a las curvas de aprendizaje con algoritmos genéticos tiene una alta desviación estándar en relación a la de la curva con NEAT. Sin embargo, estas altas desviaciones se deben básicamente a que hubo, en ambos casos, solo una ronda de aprendizaje con fitness final considerablemente menor al resto (en ambos casos la 4^{ta} ronda de las tablas 4.4 y 4.5).

En todas las curvas de aprendizaje se pudo establecer cualitativamente un punto desde el cual el proceso de aprendizaje entra en una meseta, o zona de aprendizaje lento. Estos resultados se presentan en la tabla 5.2. De aquí se puede ver que todas las curvas tienen una tendencia convergente en relación a este punto de decrecimiento de la pendiente de aprendizaje.

Tabla 5.2: Punto de entrada en la meseta de aprendizaje y las desviaciones estándar del proceso de aprendizaje en relación a este punto, para NEAT y Algoritmos Genéticos con activación aleatoria y secuencial.

Method	**Decreasing Point**	**Pre-Deviation**	**Post-Deviation**
NEAT	100^{va}	0.0257	0.0185
Random GA	500^{va}	0.1397	0.1145
Sequential GA	2000^{va}	0.2221	0.1540

5.2. Análisis de Invarianza a la Forma Inicial del Robot

En las tablas 5.3 y 5.4 se presenta un resumen de los resultados de invarianza a la configuración inicial de los módulos del robot para las dimensiones de la tabla 4.1. Se puede notar que el individuo generado con algoritmos genéticos y activación secuencial obtuvo fitness cercanos al de entrenamiento en todas las configuraciones. Este comportamiento es independientemente del escenario en que se corra la simulación.

Tabla 5.3: Fitness medio y desviación estándar del ensayo de forma inicial en el escenario de entrenamiento en todas las configuraciones.

Method	**Mean Fitness**	**Standard Deviation**
NEAT	0.4499	0.2134
Random GA	0.5436	0.0456
Sequential GA	0.9121	0.0122

Tabla 5.4: Fitness medio y desviación estándar del ensayo de forma inicial en el escenario de prueba en todas las configuraciones.

Method	**Mean Fitness**	**Standard Deviation**
NEAT	0.4776	0.1700
Random GA	0.5246	0.0669
Sequential GA	0.9179	0.0094

En la tabla 5.5 se presentan los cuocientes entre la desviación estándar de los fitness alcanzados y el fitness medio, en los dos escenarios, ensayados para las distintas configuraciones de la tabla 4.1. Estos cuocientes dan una medida de que tan buena o mala fue la distribución de resultados para las distintas configuraciones. El valor de semejanza corresponde al cuociente entre estas medidas en los dos escenarios. Con esto se puede establecer que tan similares son las dispersiones de los resultados, con lo que se puede medir la dependencia de la dispersión de los resultados para con el escenario.

Tabla 5.5: Cuocientes entre la desviación estándar de los resultados, obtenidos en los dos escenarios, y el fitness medio obtenido en cada experiencia.

Method	**Training Scenario**	**Testing Scenario**	**Similarity**
NEAT	0.47	0.36	0.77
Random GA	0.08	0.13	0.61
Sequential GA	0.01	0.01	1.00

Según la medida de semejaza se puede establecer que la experiencia forma inicial es más invariante al escenario en el caso evolucionado con algoritmos genéticos y activación secuencial, seguido de los resultados con NEAT y algoritmos genéticos con activación aleatoria.

En las figuras 4.8 y 4.14, correspondientes a los resultados obtenidos con ambos algoritmos

genéticos, con se puede ver que las configuraciones con mejor fitness corresponden a las 5 y 3 de la tabla 4.1, que corresponden la las configuraciones iniciales del robot de 8x2 y 5x3 módulos, respectivamente. De esto se puede concluir que, al menos con algoritmos genéticos, se obtiene un mejor comportamiento de los algoritmos de locomoción con configuraciones más bien verticales.

5.3. Análisis de Invarianza a la Escala de Simulación

En las tablas 5.6 y 5.7 se presenta un resumen de los resultados promedios para la experiencia de invarianza a la escala de simulación para las configuraciones de la tabla 4.2

Tabla 5.6: Fitness medio y desviación estándar del ensayo de escala de simulación en el escenario de entrenamiento, en todas las configuraciones.

Method	**Mean Fitness**	**Standard Deviation**
NEAT	0.5499	0.1073
Random GA	0.4528	0.0509
Sequential GA	0.6241	0.2303

Tabla 5.7: Fitness medio y desviación estándar del ensayo de escala de simulación en el escenario de prueba, en todas las configuraciones.

Method	**Mean Fitness**	**Standard Deviation**
NEAT	0.5707	0.0969
Random GA	0.4970	0.0792
Sequential GA	0.7655	0.1669

En la tabla 5.8 se presenta un análisis similar al de la tabla 5.5, pero para la experiencia de escalamiento de simulación.

Tabla 5.8: Cuocientes entre la desviación estándar de los resultados, obtenidos en los dos escenarios, y el fitness medio obtenido en cada experiencia de escala de simulación.

Method	**Training Scenario**	**Testing Scenario**	**Similarity**
NEAT	0.20	0.17	0.85
Random GA	0.11	0.16	0.69
Sequential GA	0.37	0.22	0.59

Según la medida de semejanza se puede decir que el individuo evolucionado con NEAT es el más invariante al escenario en la experiencia de escalamiento de la simulación, seguido de los individuos evolucionados con algoritmos genéticos con activación aleatoria y activación secuencial.

En el caso de algoritmos genéticos con activación secuencial es evidente la relación inversa entre el fitness obtenido y la escala de simulación (figura 4.15).

5.4. Discusión de Resultados

Los resultados mostraron que, en general, no hay una relación clara entre los parámetros modificados y el comportamiento del robot simulado, salvo en el caso del individuo generado con algoritmos genéticos y activación secuencial de los módulos. Para este caso se vió que el robot no varía su comportamiento significativamente frente a distintas configuraciones iniciales del robot. También se vió que, con los algoritmos generados, el fitness del robot está inversamente relacionado con la escala de simulación. En ambas experiencias, los resultados no dependen del escenario donde se ensaye.

En el caso de los las reglas generadas con algoritmos genéticos (independientemente del tipo de activación), se vió que los mejores desempeños, ante la variación de la forma inicial del robot, se obtuvieron en las configuración iniciales más altas que anchas.

Sin embargo, hubo una variable no ponderada que tiende a inhabilitar la condición de conectividad. Esta variable es el número de iteraciones de simulación, que se fijó en 500 para todas las rutinas de aprendizaje. Esta variable implica que en cada iteración de estas 500 se evalúan todos los módulos.

El término de conectividad opera promediando el porcentaje de módulos conexos através de las iteraciones de simulación. Entonces, como se vió en gran parte de las soluciones, si el robot llega en pocas iteraciones a un estado de estancamiento, pero con unos pocos módulos aún movieńdose en torno a grueso del robot, se tiene que el promedio de conectividad deja de representar al desplazamiento inconexo. En otras palabras, el promedio de conectividad tiende a 1, a pesar del desplazamiento inconexo.

Aunque el simulador efectivamente reconoce el estancamiento temprano de la simulación, el aislamiento de estas situaciones de “estabilidad dinámica” se hace más complejo. Tampoco hay

una forma clara de acotar el número de iteraciones de simulación, dado que no se sabe cuantas le toma al robot en llegar a un estado estable.

Capítulo 6

Conclusiones

En el presente trabajo de memoria se generaron distintas metodologías para sintetizar algoritmos de control descentralizado para robots modulares reconfigurables genéricos. Para esto, se desarrolló un simulador e implementaron distintas metodologás de entrenamiento evolutivo. También se caracterizaron los resultados obtenidos en experiencias de variación de la forma inicial del robot y modificando la escala de la simulación.

Los resultados mostraron que, en general, no se puede hablar de invarianza a la forma inicial ni a la escala de la simulación, salvo en el caso del individuo generado con algoritmos genéticos y activación secuencial de los módulos. Para este caso se vió que el robot no varía su comportamiento significativamente frente a distintas configuraciones iniciales del robot. También se vió que, con los algoritmos generados, el desempeño del robot es inversamente proporcional con la escala de simulación, lo que tiene relación con el número de iteraciones del simulador. Además, el comportamiento, en las experiencias de forma y escala, es similar en ambos escenarios.

En el caso de los las reglas generadas con algoritmos genéticos (independientemente del tipo de activación), se vió que los mejores desempeños, ante la variación de la forma inicial del robot, se obtuvieron en las configuración iniciales más altas que anchas.

En general, se ha probado que se pueden generar algoritmos de control para locomoción de robots modulares reconfigurables con métodos evolutivos. Esto se ve en el hecho de que se logró que el robot se moviese en la dirección deseada con todos los métodos probados en este trabajo.

Sin embargo, hubo una variable no ponderada que tiende a inhabilitar la condición de conectividad. Esta variable es el número de iteraciones de simulación, que se fijó en 500 para todas las rutinas de aprendizaje. Esta variable implica que en cada iteración de estas 500 se evalúan todos

los módulos.

El término de conectividad opera promediando el porcentaje de módulos conexos através de las iteraciones de simulación. Entonces, como se vió en gran parte de las soluciones, si el robot llega en pocas iteraciones a un estado estacionario, donde unos pocos módulos quedan orbitando en torno a grueso del robot, se tiene que el promedio de conectividad deja de representar al desplazamiento inconexo. En otras palabras, el promedio de conectividad tiende a 1, a pesar del desplazamiento inconexo.

Otro problema con el que se tuvo que lidiar fue la activación aleatoria de los módulos. En primera instancia se prefirió respetar la aleatoriedad propuesta en la literatura, dado que este enfoque aumenta la robustez del robot en entornos no controlados. Sin embargo, para efectos del entrenamiento esto fue catastrófico y, de hecho, esto fue lo que dio pie a realizar el análisis de algoritmos genéticos con y sin activación secuencial de los módulos. Un claro ejemplo de esto se puede ver en la figura 4.8, donde lo esperable era que el fitness del individuo (generado con algoritmos genéticos y activación aleatoria), para la primera configuración de la tabla 4.1, fuese al menos similar al fitness alcanzado durante el aprendizaje. Esto considerando que la primera configuración de prueba corresponde a la misma del entrenamiento.

El simulador, al ser básicamente numérico, mostró una gran versatilidad de uso. Esta representación numérica permitió implementar directamente los algoritmos que involucran redes neuronales, lo que es una propuesta novedosa que no había sido probada en trabajos anteriores. Sin embargo, este simulador es simple, en el sentido de que es bidimensional y no considera factores tales como la gravedad, por lo que también queda propuesto extender el simulador con este tipo de variables.

Otro tema, que requirió un esfuerzo importante, fue la codificación de las reglas explícitas que debían manipular los algoritmos genéticos. Si bien se logró reducir la representación de cada regla a 21 bits, hay que tener en cuenta que el algoritmo de locomoción está constituido por una colección de reglas, por lo que el número de bits crece considerablemente.

A pesar de todos los puntos por corregir, es destacable el hecho de que los mejores resultados se obtuvieron con una versión simple de los algoritmos genéticos. En experiencias tempranas se probó el algoritmo genético que viene implementado por defecto en Matlab R2010a, con buenos resultados. Por un tema de manejo de datos y transparencia del método se optó por programar un algoritmo propio. Sin embargo, en los problemas de calibración se vió que el algoritmo de Matlab

obtenía resultados buenos en apenas 20 generaciones, mientras que el algoritmo genético simple, utilizado en la memoria, requería cerca de 100 generaciones en alcanzar resultados similares. Esto da cuenta del alto potencial de mejoramiento si se implementaran operadores más avanzados.

En fin, se demostró que el concepto de sintetizar reglas de control para locomoción de robots modulares genéricos es posible, algo que no había sido probado a la fecha.

Capítulo 7

Bibliografía

[1] S.C.Goldstein, J.D. Campbell & T.C. Mowry, "*Programmable matter*" IEEE Computer, vol. 28, no. 6, pp. 99101, May 2005.

[2] B.Salemi, M. Moll & W.-M. Shen,"*SUPERBOT: A deployable, multi-functional, and modular self-reconfigurable robotic system*", Proc. 2006 IEEE/RSJ Intl. Conf. Intelligent Robots Systems, Oct. 2006, pp.3636-3641.

[3] K.Kotay & D.Rus. "*Generic Distributed Assembly and Repair Algorithms for Self-Reconfiguring Robots*", Procs. of IEEE/RSJ Int. Conf. on Intellingent Robots and Systems, 2004, pp.2362-2369.

[4] Z.Butler, K.Kotay, D.Rus & K.Tomita, " *Generic Descentralized Control for a Class of Self-Reconfigurable Robots*", Procs. of IEEE Int. Conf. on Robotics and Automation, 2002, pp.809-816.

[5] A.Kamimura, S.Murata, E.Yoshida, H.Korosawa, K.Tomita & S.Kokaji, "*Self-Reconfigurable Modular Robot, Experiments on Reconfiguration and Locomotion*", Procs. of IEEE/RSJ Int. Conf. on Intelligent Robots and Systems, 2001.

[6] C.Chiang & G.Chirikjian, "*Modular Robot Motion Planning Using Similarity Metrics*", Autonomous Robots 10, 2001, pp.91-96.

[7] R.Ravichandran, G.Gordon & S.Copen Goldstein, "*A Scalable Distributed Algorithm for Shape Transformation in Multi-Robot Systems*", Procs. of IEEE/RSJ Int. Conf. on Intellingent Robots and Systems, 2007, pp.4188-4193.

[8] J.Kubica, A.Casal & T.Hogg, "*Complex Behaviors form Local Rules in Modular Self-Reconfigurable Robots*", Procs. of IEEE Int. Conf. on Robotics and Automation, 2001, pp.360-367.

[9] G.Honrby, "*The Age-Layered Population Structure for Reducing the Problem of Premature Convergence*", Procs. of the Genetic and Evolutionary Computation Conf., 2009.

[10] M.Kolantarov, M.Tolley, H.Lipson & D.Erickson, "*Hydrodinamically Driven Docking of Blocks for 3D Fluidic Assembly*", Microfluid Nanofluid 9, 2010, pp.551-558.

[11] M.Tolley, M.Krishnan, D.Erickson & H.Lipson, "*Dynamically Programable Fluidic Assembly*", Applied Physics Letters, 2008, pp.93-95.

[12] M.Tolley, M.Kalontarov, J.Neubert, D.Erickson & H.Lipson, "*Stochastic Modular Robotic Systems: A Study of Fluidic Assemble Strategies*", IEEE Transactions on Robotics 26, no. 3, 2010, pp. 518-530.

[13] M.Krishnan, M.Tolley, H.Lipson & D.Erickson, "*Increased Robustness for Fluidic Self-Assembly*", Physics of Fluidics 20, 2008, pp. 20-36.

[14] R.Fitch, Z.Butler & D.Rus, "*Reconfiguration Planning for Heterogeneous Self-Reconfigurable Robots*", Procs. of IEEE/RSJ Int. Conf. on Intelligent Robots and Systems, 2003, pp. 2460-2467.

[15] R.Garca, J.Hiller & H.Lipson, "*A Vacuum-Based Bonding Mechanism for Modular Robotics*", Procs. of IEEE Int. Conf. on Robotics and Automation, 2010, pp.57-62.

[16] H.Lipson, "*Evolutionary Robotics and Open-Ended Design Automation*", Biomimetics, B.Cohen, ed., CRC Press, 2005, pp.129-155.

[17] Z.Butler, S.Murata, & D.Rus, "*Distributed Replication Algorithms for Self-Reconfiguring Modular Robots*", 6th Int. Conf. Distributed Autonomous Robotic Systems , 2002

[18] D.Linden, "*Antenna Design Using Genetic Algorithms*", GECCO '02 Procs. of the Genetic and Evolutionary Computation Conference, ed. Morgan Kaufmann Publishers, pp. 1133-1140, 2002.

[19] D.Andre, F.Bennett & J.R. Koza, "*Evolution of Intricate Long-Distance Communication Signals in Cellular Automata Using Genetic Programming*", Artificial Life V (Christopher G. Langton and Katsunori Shimohara, eds.), pp. 513-520. Cambridge, Massachusetts: MIT Press, 1997.

[20] Z.Butler, K.Kotay, D.Rus & K.Tomita, "*Generic Decentralized Control for Lattice-Based Self-Reconfigurable Robots*", Int. Journal of Robotics Research, vol 23, pp.919-937,2004.

[21] D.Goldberg, "*Genetic Algorithms in Search, Optimization & Machine Learning*", Ed. Addison Wesley Longman Inc., 1989.

[22] K. Stanley & R.Miikkulainen, "*Evolving Neural Networks through Augmenting Topologies*", Evolutionary Computation 10(2), 2002, pp.99-127.

[23] M.Yim, J.Lamping, J.Mao & J.G.Chase, "*Rhombic Dodecahedron Shape for Self-Assembling Robots*", Xerox PARC, SPL TechReport P9710777, 1997.

[24] Alba.E & Cotta.C, "*Evolutionary Algorithms*", Dpto. Lenguajes y Ciencias de la Computación, ETSI Informática, Universidad de Málaga, 2004.

[25] Jones.T, "*Evolutionary Algorithms, Fitness Landscapes and Search*", Tesis de Doctorado, Universidad de México, 1995.

[26] J.Zagal, C.Morales & F.Torres, "*Robótica y Automatización del Diseño*", apuntes del curso ME-704, Dpto. Ing. Mecánica, 2010.

[27] S.Wolfram , "*Cellular automata as models of complexity*", Nature v.311, pp.419-424, 1984.

[28] L.J.Fogel, A.J. Owens & M.J. Walsh, "*Artificial Intelligence Through Simulated Evolution*", Wiley, New York, 1966.

[29] I.Rechenberg, "*Evolutionsstrategie: Optimierung technischer Systeme nach Prinzipien der biologischen Evolution*", Frommann-Holzboog Verlag, Stuttgart, 1973.

[30] H.P.Schwefel, "*Numerische Optimierung von ComputerModellen mittels der Evolutionsstrategie*", volume 26 of Interdisciplinary Systems Research. Birkh user, Basel, 1977.

[31] J.H.Holland, "*Adaptation in Natural and Artificial Systems*". University of Michigan Press, Ann Harbor, 1975.

[32] D.Moriarty, "*Symbiotic Evolution of Neural Networks in Sequential Decision Task*", Ph.D Thesis, Department of Computer Sciences, University of Texas at Austin, 1997.

[33] D.Moriarty & R.Miikkulainen, "*Efficient reinforcement learning through symbiotic evolution*", Machine Learning, 22:11-32, 1996.

[34] F.Gruau, "*Genetic Synthesis of Modular Neural Networks*", Procs. of the 5th Int. Conf. on Genetic Algorithms, pp.318-325, 1993.

[35] M.Mandischer, "*Representation and Evolution of Neural Networks*", Artificial Neural Nets and Genetic Algorithms, pp.643-649, 1993.

[36] D.Goldberg & J.Richardson, "*Genetic Algorithms with Sharing for Multimodal Function Optimization*", Procs. of the 2nd Int. Conf. on Genetic Algorithms, pp.148-154, 1987.

[37] N.Packard, "*Cellular Automaton Models for Dendritic Growth*", Institute for Advance Study, 1984.

[38] D.Young, "*A Local Activator-Inhibitor Model of Vertebrate Skin Patterns*", Lawrence Livermore National Laboratory, 1983.

[39] M.Gardner, "*Wheels, Life and other Mathematical Amusements*", 1983.

[40] R.Berlekamp, R.Guy & J.Conway, "*Winning Ways for your Mathematical Plays*", vol.2, 1982.

[41] J.Koza, "*Genetic Programming II: Automatic Discovery of Reusable Programs*", Cambridge, MA:MIT Press, 1994.

Apéndice A

Código del Análisis con Algoritmo Genético Simple

El algoritmo genético desarrollado está basado en el Algoritmo Genético Simple propuesto por David Goldberg en [21].

A.1. Código Principal

```
%% Add paths
path(path,'./Ga')
path(path,'./Sim')
dir = strcat('./Results/',date,'.mat');
dir=char(saving_files(i));

%% Open matlabpool
if matlabpool('size') == 0
    matlabpool open
end

%% Rules Setup
% center of input rule is always a robot (3x3 -> 8)
% binary for the 8 posible actions (3x3\(2,2))

number_of_rules = 15;
input_rules = 9*2; % bits needed per rule
output_rules = 3; % bits needed per rule

% total number of bits needed
num_var = number_of_rules*(input_rules + output_rules);

%% Sim setup
sim_opt = sim_options(  'sim_iterations', 500, ...
    'input_rules', input_rules, ...
```

```
    'output_rules', output_rules, ...
    'rules_num', number_of_rules, ...
    'random_rule', 'on', ... % on|off
    'num_var', num_var, ...
    'robot_height', 4, ...
    'robot_width', 4, ...
    'scenario_height', 30, ...
    'scenario_width', 50, ...
    'secuential_activation','true', ... % true | false
    'floor_depth', 1, ...
    'plot','false');
CM = build_scenario(sim_opt,'./Pics/pattern_1.png');
fitness_fcn = @(x) simulator(x, CM, sim_opt);

%% Ga Setup
ga_opt = ga_options('generations', 10000, ...
    'stall_limit', 1000, ...
    'elite', 2, ...
    'crossover_prob', 0.80 , ...
    'mutation_prob', 1/num_var, ...
    'popsize', 30, ...
    'num_var',num_var, ...
    'selection_type', 'roulette', ...% 'tournament'
    'crossover_type', 'double', ...% 'double','single'
    'scaling', 'true', ... % 'false'
    'scaling_multiple', 2, ... % pressure (1.1-2)
    'report','true', ...
    'report_mark', 25, ...
    'plot','false', ... % true|false
    'file_name', dir, ...
    'autosave', 50);

%% Main
[x,fval,max,avg,min] = ...
    ga_main(fitness_fcn, ga_opt);

if matlabpool('size') > 0
    matlabpool close
end
save(dir);
disp('Process completed succesfully ...')
```

A.2. Código Algoritmo Genético Simple

```
function [x,fval,max,avg,min,pic] = ...
    ga_main(fun, options)

%[X, FVAL, MAX, AVG, MIN, PLOT] = GA_MAIN(FUN, NUM_VAR, OPTIONS)
```

```
%
% Returns the X value of the fittest individual
% (represented by NUM_VAR binaries)
% and its function valur FVAL. It also returns
% three vectors containing the MAX, AVERAGE and
% MINIMUM fitness values of each generation.
%
% You can also save a handle to the learning curve PIC.



%% Set options
gen_limit = options.generations;
% stall_gen_limit = options.stall_generations;

pop_size = options.popsize;

num_var = options.num_var;


%% Statistics
report = strcmpi(options.report,'true');

% Preallocate statistics vectors
avg = zeros(gen_limit,1); max = avg; min = avg;

% Initialize population
gen = 0;
stall_count = 0;
current_max = 0;

[old_pop old_fit] = initialize_pop(options,fun);


%% Start
if report
    fprintf('\nMax\t\t Average\t Min\t\t Stall%%\t Generation\n\n');
end


 %% Main Loop
while (gen < gen_limit) && (stall_count < options.stall_limit)

    gen = gen + 1;

    [avg(gen), max(gen), min(gen)] = ...
        statistics(old_fit, gen, report, options, stall_count);

    if max(gen) == current_max % stall control
```

```
        stall_count = stall_count + 1;
    else
        stall_count = 0;
    end

    current_max = max(gen);

    [new_pop new_fit] = ...
        generation(old_pop, old_fit, fun, options);

    old_pop = new_pop; old_fit = new_fit;

    if mod(gen,options.autosave) == 0 % autosave
        last_gen = [old_pop old_fit];
        last_gen = sortrows(last_gen,num_var+1);

        x = last_gen(pop_size,1:num_var);
        fval = last_gen(pop_size,num_var+1);

        save(options.file_name);
        fprintf('\nAutosave completed ... file name %s\n\n', ...
            options.file_name);
    end

end


%% Last generation relevant values
last_gen = [old_pop old_fit];
last_gen = sortrows(last_gen,num_var+1);

x = last_gen(pop_size,1:num_var);
fval = last_gen(pop_size,num_var+1);


%% Plotting
if strcmpi(options.plot, 'true')
    pic = plot(max); hold; plot(avg,'k--'); plot(min,'r.-');
    xlabel('Generation'); ylabel('Fitness');
    legend('Max','Avg','Min','Location','SouthEast');
end

if gen ~= gen_limit
    disp('Stall limit reached ...');
end

save(options.file_name);

disp('GA finished ...');
```

```
%function [pop,fitness] = initialize_pop(popsize, num_var, fun)
function [pop,fitness] = initialize_pop(opt, fun)
%
% [POP,FITNESS] = INITIALIZE_POP(OPTIONS, FUN)
%
% Generates a random binary population of POPSIZE individuals in a POP
% matrix. It also generates a FITNESS vector containing this value for each
% individual

% generates a random binary population
pop = round(rand(opt.popsize, opt.num_var));

seed = opt.seed;

if (size(seed,2) == opt.num_var) && (size(seed,1) <= opt.popsize)
    pop(1:size(seed,1),:) = seed;
end

fitness = zeros(opt.popsize,1);

% evaluates each individual's fitness
for i=1:opt.popsize
    fitness(i) = fun(pop(i,:));
end


function opt = ga_options(varargin)

% Create/alter MY_GA options structure
%
% Default values:
%
%     'generations', 100, ...
%     'elite', 1, ...
%     'crossover_prob', 0.6 , ...
%     'mutation_prob', 0.05, ...
%     'popsize', 30, ...
%     'selection_type', 'roulette', ...% 'tournament'
%     'crossover_type', 'single', ...% 'double'
%     'scaling', 'true', ... % 'false'
%     'scaling_multiple', 2, ...% [1.2, 2] recomended
%     'report','true', ... % 'false'
%     'report_mark',25, ...% report titles
%     'plot','true');


%% Default Values
opt = struct(   'generations', 100, ...
```

```
                'stall_limit', 100, ...
                'elite', 1, ...
                'crossover_prob', 0.6 , ...
                'mutation_prob', 0.05, ...
                'popsize', 25, ...
                'num_var', 1, ...
                'seed', [], ...
                'selection_type', 'roulette', ...% 'tournament'
                'crossover_type', 'single', ...% 'double'
                'scaling', 'true', ... % 'false'
                'scaling_multiple', 2, ...% [1.2, 2] recomended
                'report','true', ... % 'false'
                'report_mark',25, ...% report titles
                'plot','true', ...
                'file_name',[], ... % path for saving solutions
                'autosave', 10 ... % generations between autosave
                );


%% Modified Values
for i=1:nargin-1
    if isfield(opt,varargin{i})
        opt.(varargin{i}) = varargin{i+1};
    end
end


function [new_pop,new_fit] = ...
    generation(old_pop, old_fit, fun, options)

% [NEW_POP new_fit] = GENERATION(OLD_POP, FUN, OPTIONS)
%
% Generates a new population using the crossover and mutation
% operators, each one with a CROSSOVER_PROB and a MUTATION_PROB
% probability of ocurrance.
%
% ELITE count preserves the best individuals from each generation
% to the next.

pop_size = size(old_pop,1);

str_len = size(old_pop,2);

elite = options.elite;

new_pop = zeros(size(old_pop));
new_fit = zeros(pop_size,1);

%% Elite count
best_ind = sortrows([old_pop old_fit],str_len+1);
```

```
for i=1:elite
    new_pop(i,:) = best_ind(size(best_ind,1),1:str_len);

    new_fit(i) = ...
        best_ind(size(best_ind,1),size(best_ind,2));

    best_ind(size(best_ind,1),:) = [];
end

%% The rest of the operations (cross with embadded mutation)
 j = elite;

while j < pop_size

    mate1 = selection(old_fit, options);
    mate2 = selection(old_fit, options);

    if (pop_size - j) >= 2
        % cross over with two childs
        [new_pop(j+1,:) new_pop(j+2,:)] = ...
            crossover(old_pop(mate1,:),old_pop(mate2,:),options);

        j = j + 2;

    elseif (pop_size - j) == 1
        % crossover with one child
        new_pop(j+1,:) = crossover(old_pop(mate1,:),...
            old_pop(mate2,:),options);

        % increment population count
        j = j + 1;
    end

end

%% Calculate new_pop fitness
parfor i = elite+1:pop_size
    new_fit(i) = fun(new_pop(i,:));
end


function [child1, child2] = crossover(parent1, parent2, options)
%
% [CHILD1,CHILD2] = CROSSOVER(PARENT1, PARENT2, OPTIONS)
%
% Takes 2 individuals (PARENT1 and PARENT2), and produces two new
% individuals (CHILD1 and CHILD2) by crossing them with a probability
% CROSSOVER_PROB.
%
```

```
% Else, it returns the parents with no modification.
%
% The TYPE variable can be 'SINGLE' or 'DOUBLE' for a single point
% or two-point crossover. Default type is 'SINGLE'

pc = options.crossover_prob;
type = options.crossover_type;

len = size(parent1,2);

switch type
    case {'single','Single','SINGLE'}
        if rand() < pc
            cpoint = 1 + round(rand()*(len-2));

            child1 = [parent1(:,1:cpoint) parent2(cpoint+1:len)];
            child2 = [parent2(:,1:cpoint) parent1(cpoint+1:len)];
        else
            child1 = parent1;
            child2 = parent2;
        end
    case {'double','Double','DOUBLE'}
        if rand() < pc
            cpoint1 = 1+round(rand()*(len-2));
            cpoint2 = 1+round(rand()*(len-2));

            % cpoint1 always less than cpoint2
            while cpoint1 >= cpoint2
                cpoint1 = 1+round(rand()*(len-2));
                cpoint2 = 1+round(rand()*(len-2));
            end
            child1 = [parent1(:,1:cpoint1) parent2(:,cpoint1+1:cpoint2) ...
                parent1(:,cpoint2+1:len)];

            child2 = [parent2(:,1:cpoint1) parent1(:,cpoint1+1:cpoint2) ...
                parent2(:,cpoint2+1:len)];
        else
            child1 = parent1;
            child2 = parent2;
        end
end


%% Embedded mutation
parfor i=1:len
    child1(i) = mutation(child1(i),options);
    child2(i) = mutation(child2(i), options);
end
```

```
function child = mutation(gen, options)

% CHILD = MUTATION( GEN, OPTIONS)
%
% Takes a GEN and mutates it with
% a probability MUTATION_PROB.

pm = options.mutation_prob;

if rand() < pm
    child = abs(gen-1);
else
    child = gen;
end


function fitness_out = scaling(fitness_in, options)
%
% FITNESS_OUT = SCALING(FITNESS_IN, OPTIONS)
%
% Returns a scaled fitness vector using the linear model. This model
% reduces the distance between individuals in a heterogeneous population
% and increase the distance in homogeneous populations.

c = options.scaling_multiple; % fitest desired multiple

avg = mean(fitness_in);

mx = max(fitness_in); mn = min(fitness_in);

if mn > (c*avg - mx) / (c-1) % non negative test
     delta = mx - mn;
     b = avg*(mx-c*avg) / delta;
     a = (c-1)*avg / delta;
else
    delta = avg - mn;
    a = avg / delta;
    b = -mn*avg / delta;
end

fitness_out = a*fitness_in + b;


function j = selection(fitness_vector, options)

% I = SELECTION(FITNESS_VECTOR, OPTIONS)
%
% Returns a selected individual index. The selection can be performed using
% a Roulette or a Tournament model. Default is Roulette.
%
```

```
% In the roulette model, each individual has a f(i)/sum(f(1..N)) probability
% to be selected. In the tournament model, a random individual sample is
% taken, from where the best individual is selected.

scaling_opt = strcmpi(options.scaling,'true');
type = options.selection_type;

switch type
    case {'roulette'}

        part_sum = 0; j = 0;

        if scaling_opt
            fitness_vector = scaling(fitness_vector,options);
        end

        fitness_vector = fitness_vector / sum(fitness_vector);

        val = rand();

        while part_sum < val
            j = j+1;
            part_sum = part_sum + fitness_vector(j);
        end

    case {'tournament'}

        pop_size = size(fitness_vector,1);

        % number of individuals in the tournament
        num = ceil(rand()*pop_size);

        % individuals indexes
        tournament = zeros(num,2);

        for i = 1:num
            tournament(i,1) = ceil(rand()*num);
            tournament(i,2) = fitness_vector(tournament(i,1));
        end

        tournament = sortrows(tournament,2);

        j = tournament(num,1);

end


function [avg_fit,max_fit,min_fit] = ...
    statistics(fit, gen, report, options, stall_count)
```

```
% [AVG, MAX, MIN] = STATISTICS(POPULATION, REPORT)
%
% Generates the statistics from a fitness vector.

%% Main statistics
avg_fit = mean(fit);

max_fit = max(fit);

min_fit = min(fit);


%% Report

if report

    report_int = options.report_mark;

    stall_limit = options.stall_limit;

    gen_limit = num2str(options.generations);

    fprintf('%E\t %E\t %E\t %G%%\t %G/%s\n', ...
        max_fit, avg_fit, ...
        min_fit,100*stall_count/stall_limit, gen, gen_limit);

        if mod(gen,report_int) == 0
            fprintf('\nMax\t\t Average\t Min\t\t Stall%%\t Generation\n\n');
        end
end
```

A.3. Código Simulador AG

```
function y = simulator(x, CM, options)

% Y = SIMULATOR(X,CM,OPTIONS)
%
% Simulates a configuration specified in OPTIONS, on a scenario CM
% with a set of rules X.
%
% It returns the average position in the horizontal axis for all
% modules.

do_plot = strcmpi(options.plot,'true');
secuential_activation=strcmpi(options.secuential_activation,'true');
colormap(gray);

%% Simulation settings
```

```
iterations = options.sim_iterations;
rules = cod2rules(x, options);
iter_count = 1;

list_rob = current(CM, options);
connected = zeros(iterations,2);
%% Main Loop

while iter_count < iterations
    CM_backup = CM;
    while size(list_rob,1)~=0 % D1 loop
        if secuential_activation
            rindx=1;
        else
            rindx=ceil(size(list_rob,1)*rand());
        end
        pos = [list_rob(rindx,1) list_rob(rindx,2)];
        CM = apply_rules(CM, pos, rules, options);
        if do_plot
            if secuential_activation
                CM_aux=CM;
                CM_aux(CM_aux >= 3) = 3;
                imagesc(CM); drawnow;
            else
                imagesc(CM);drawnow;
            end
        end
        list_rob(rindx,:) = [];
    end
    list_rob = current(CM, options);
    connected(iter_count,:) = check_connected(CM);
    iter_count = iter_count + 1;
    if CM == CM_backup % Break if no movement
        y = fitness_fcn(list_rob,connected,iter_count,options);
        return
    end
end

%% Fitness Function
y = fitness_fcn(list_rob,connected,iterations,options);


function CM = build_scenario(options,file)

% CM = BUILD_SCENARIO(OPTIONS, FILE)
%
% Creates an scenario with the dimensions specified in OPTIONS from
% an image contained in FILE. If no file is given, it creates an
% empty scenario.
%
```

```
% Types of cell
% 0 reserved for non evaluated cells
% 1 == background
% 2 == obstacle
% 3 == robot

num_arg = nargin;

M = options.scenario_height;
N = options.scenario_width;
rw = options.robot_width;
rh = options.robot_height;
fd = options.floor_depth;


%% Creates obstacles
switch num_arg
    case 1 % Build default empty scenario
        CM = ones(M,N);

    case 2 % Create scenario from image file
        CM = imread(file);
        CM = imresize(CM,[M N]);
        CM = rgb2gray(CM);
        CM = histeq(CM);
        CM = im2bw(CM) + 1;
end


%% Creates Frame
CM(:,1:fd) = 2;
CM(:,N+1-fd:N) = 2;

CM(1:fd,:) = 2;
CM(M+1-fd:M,:) = 2;


%% Creates square robot
if strcmpi(options.secuential_activation,'true')
    list = 1:rh*rw;
    list=list+2;
    CM(M-rh-fd+1:M-fd,2*fd+1:2*fd+rw) = reshape(list,rh,rw);
else % annonymous bots
    CM(M-rh-fd+1:M-fd,2*fd+1:2*fd+rw) = 3;
end


function sum = bin_2_dec(x)

% Takes a ROW of 1's and 0's and transform it to a decimal
```

```
% number
%
% example:
%   x = [1 0 0]
%   bin_2_dec(x) returns 4

if size(x,1)~=1
    error('Argument should be a single row');
end

vars = size(x,2);

sum = 0;

for i=1:vars
    sum = sum + 2^(i-1)*x(vars-i+1);
end


function CM = apply_rules(CM, pos, rules, opt)

i = pos(1); j = pos(2);
num_rules = size(rules,3);
ratio = 1;
SM_in = CM(i-ratio:i+ratio , j-ratio:j+ratio);
list = 1:num_rules;

while numel(list) > 0

    % Select rule to evaluate
    if strcmpi(opt.random_rule,'on') % random
        rule_pos = ceil(rand()*numel(list));
    else % secuencial
        rule_pos = 1;
    end

    % Apply rule
    rule_val = list(rule_pos);
    %keyboard
    if fits(SM_in, rules(1:3,:,rule_val), rules(4,2,rule_val))
        SM_out = swap(SM_in, rules(4,3,rule_val));
        CM(i-ratio:i+ratio, j-ratio:j+ratio) = SM_out;
        break
    end

    list(rule_pos) = [];
end


function y = check_connected(cm_original)
```

```
% Returns the size of the biggest robot connected group

% only robot cells visible
cm=cm_original;
cm(cm>=3)=3;
cm(cm ~= 3) = 0;
cc = bwconncomp(cm);
numPix_per_group = cellfun(@numel,cc.PixelIdxList);
y = max(numPix_per_group);


function rules = cod2rules(x, options)

% Return a (3+1)x3xnum matrix containing the rule set decodefied
% from a bit vector of lenght 'num', assuming a set of Moore's rules

% Single rule lenght = 21
% 3+18 = 3 output bits + 2 bits x 9 positions (Moore's)

% Pre allocate output rules

num = options.rules_num;

input_vector = zeros(3*3*num,1);

% output_vector = zeros(num,1);

for i=1:9*num
    var = 2*(i-1);
    input_vector(i) = bin_2_dec(x(1+var:2+var));
end

limit = 2*9*num+1; % where the output bits begins


% Reshape inputs to a 3x3xnum matrix

rules = reshape(input_vector,[3 3 num]);

% Force center to be always a robot ('3')

rules(2,2,:) = 3;

% Decode output vector and assing it to the rules position (4,3)

for i=1:num
    var = 3*(i-1);
    rules(4,3,i) = bin_2_dec(x(limit+var:(limit+2)+var));
    rules(4,2,i) = sum(sum(rules(1:3,:,i)~=0));
```

```
end


function list_out = current(CM, options)

% LIST_OUT = CURRENT(CM, OPTIONS)
%
% Returns a list containing the positions of
% all robot cells.

if strcmpi(options.secuential_activation,'true')
    robot_number=options.robot_width*options.robot_height;
    list_out=zeros(robot_number,2);
    for i=3:robot_number+2 %generates list in id order for secuential activation
        [row col]=find(CM==i);
        list_out(i-2,:)=[row col];
    end
else
    [row col]=find(CM==3);
    list_out=[row col];
end


function y = fitness_fcn(list_rob, connected_hist, count, opt)

% w = opt.robot_width;
% h = opt.robot_height;

w = opt.robot_width;
h = opt.robot_height;

N = opt.scenario_width;

% Displacement
dist = mean(list_rob(:,2));
%y = mean(list_rob(:,2))/N;

% Biggest group history
connected = mean(connected_hist(1:count));

y = (dist/N) * (connected/(w*h));
% connected == pctje of the total number
% of robot that remained connected throught simulation


function fits = fits(SM_in, rule_in, to_analize)

% States
% 0: not evaluated
% 1: background
```

```
% 2: obstacle
% 3: robot

SM_in(2,2)=3; % assume 3x3 rule, force center to be a 3(robot).

rule_len = size(rule_in,2);

count = 0;

for i=1:rule_len
    for j=1:rule_len
        if rule_in(i,j)~=0
            switch rule_in(i,j)

                case 1 % Evaluated BackGround
                    if SM_in(i,j) == 1
                    count = count + 1;
                    end

                case 2 % Evaluated Obstacle
                    if SM_in(i,j) == 2
                        count = count + 1;
                    end

                case 3 % Evaluated Robot
                    if SM_in(i,j) == 3
                        count = count + 1;
                    end
            end
        end
    end
end

fits = (count == to_analize);


function opt = sim_options(varargin)

% Create/alter scenario parameters


%% Default Values

opt = struct(   'sim_iterations', 40, ...
                'input_rules', 0, ...
                'output_rule', 0, ...
                'rules_num', 0, ...
                'random_rule', 'on', ...
                'num_var', 0, ...
                'robot_height', 3, ...
```

```
                'robot_width', 3, ...
                'scenario_height',15, ...
                'scenario_width', 25, ...
                'secuential_activation', 'true', ...
                'floor_depth', 1, ...
                'plot', 'false');


%% Modified Input Values
for i=1:nargin-1
    if isfield(opt,varargin{i})
        opt.(varargin{i}) = varargin{i+1};
    end
end


function SM_out = swap(SM_in, rule_out)

%  output:(
% 8 posible actions in the Moore neighborhood
% [0 1 2
%  3   4
%  5 6 7]

equivalence=[1 1;1 2;1 3;2 1;2 3;3 1;3 2;3 3];

SM_out=SM_in;

if SM_out(equivalence(rule_out+1,1),equivalence(rule_out+1,2)) == 1
    SM_out(equivalence(rule_out+1,1),equivalence(rule_out+1,2)) = SM_out(2,2);
    SM_out(2,2)=1;
    return
end
```

Apéndice B

Código del Análisis con NEAT

EL código de NEAT utilizado es una modificación del experimento XOR desarrollado por Christian Mayr[1].

B.1. Código Principal

```
clear;
tic;
path(path,'./Sim'); % Add simulator path
if matlabpool('size') == 0 % Open matlabpool
    matlabpool open
end
% Set simulator options
simulation.sim_iterations = 500;
simulation.robot_height = 4;
simulation.robot_width = 4;
simulation.scenario_height = 30;
simulation.scenario_width = 50;
simulation.floor_depth = 1;
simulation.plot = 'false'; % 'true' for visulalization
simulation.ID = 'on'; % 'on' to secuential modules activation.
simulation.runs = 1;% to avoid oscilating fitness, a mean of simulation.runs will be
    assigned as fitness
CM = build_scenario(simulation,'./Pics/pattern_1.png');% create scenario, if no image file
    is given, an empty square scenario is created
%CM = build_scenario(simulation); % empty scenario

%%%%%%%%%%%%%%list of NEAT parameters %%%%%%%%%%%%%%%%%%%%%%%%%%
maxgeneration=1000; %maximum number of generations for generational loop
load_flag=0; %if set to 1, will load population, generation, innovation_record and
    species_record from neatsave.mat at start of algorithm, if set to 0, algorithm will
    start with initial population, new species record and new innovation record, at
    generation=1 (default option)
```

[1] http://www.cs.ucf.edu/~kstanley/neat.html

```
save_flag=1; %if set to 1, will save population, generation, innovation_record and
    species_record to neatsave.mat at every generation (default option)
% upshot of this is: the settings above will start with initial population (of your
    specification) and save all important structures at every generation, so if your
    workstation crashes or you have to interrupt evolution, you can, at next startup, simply
     set the load flag to 1 and continue where you have left off.
% Please note, however, that only changing structures are saved, the parameters in this
    section will not be saved, so you have to ensure that you use the same parameters when
    using a saved version of evolution as when you created this saved version!
% Also note that all variables are saved in binary matlab format, so file is only readable
    by matlab. If you want to look at some of the values stored in this file, load it in
    matlab, then you can access the saved values
average_number_non_disabled_connections=[];
average_number_hidden_nodes=[];
max_overall_fitness=[];

fitness_record=zeros(1,maxgeneration); % record of the max and mean population fitness at
    each generation.

%parameters initial population
population_size=150;
number_input_nodes=8; % correspond to the current module's 8 neighbors
number_output_nodes=4;

%vector_connected_input_nodes=[1 2]; %vector of initially connected input nodes out of
    complete number of input nodes
%(if you want to start with a subset and let evolution decide which ones are necessary)
%for a fully connected initial population, uncomment the following:
vector_connected_input_nodes=1:number_input_nodes;


%speciation parameters
% The following structure will contain various information on single species
% This data will be used for fitness sharing, reproduction, and for visualisation purposes
species_record(1).ID=0;%consecutive species ID's
species_record(1).number_individuals=0;%number of individuals in species
species_record(1).generation_record=[]; %matrix will be 4 rows by (number of generations
    existent) columns, will contain (from top to bottom) number of generation, mean raw
    fitness, max raw fitness, and index of individual in population which has produced max
    raw fitness

speciation.c1=1.0; %Speciation parameters as published by Ken Stanley. "Delta function"
speciation.c2=1.0;
speciation.c3=0.4;
speciation.threshold=10;% original threshold 3;

%reproduction parameters
%stagnation+refocuse
```

```
stagnation.threshold=1e-2; %threshold to judge if a species is in stagnation (max fitness of
     species varies below threshold) this threshold is of course dependent on your fitness
    function, if you have a fitness function which has a large spread, you might want to
    increase this threshold
stagnation.number_generation=maxgeneration; %15 original %if max fitness of species has
    stayed within stagnation.threshold in the last stagnation.number_generation generations,
     all its fitnesses will be reduced to 0, so it will die out
%Computation is done the following way: the absolute difference between the average max
    fitness of the last stagnation.number_generation generations and the max fitness of each
     of these generations is computed and compared to stagnation.threshold.
%if it stays within this threshold for the indicated number of generations, the species is
    eliminated
refocus.threshold=1e-2;
refocus.number_generation=20; %if maximum overall fitness of population doesn't change
    within threhold for this number of generations, only the top two species are allowed to
    reproduce


%initial setup
initial.kill_percentage=0.2; %the percentage of each species which will be eliminated (
    lowest performing individuals)
initial.number_for_kill=5; % the above percentage for eliminating individuals will only be
    used in species which have more individuals than min_number_for_kill
% Please note that whatever the above settings, the code always ensures that at least 2
    individuals are kept to be able to cross over, or at least the single individual in a
    species with only one individual
initial.number_copy=5; % species which have equal or greater than number_copy individuals
    will have their best individual copied unchanged into the next generation

%selection (ranking and stochastic universal sampling)
selection.pressure=2; % Number between 1.1 and 2.0, determines selective pressure towards
    most fit individual of species

%crossover
crossover.percentage=0.8; %percentage governs the way in which new population will be
    composed from old population.  exception: species with just one individual can only use
    mutation
crossover.probability_interspecies=0.0 ; %if crossover has been selected, this probability
    governs the intra/interspecies parent composition being used for the
crossover.probability_multipoint=0.6; %standard-crossover in which matching connection genes
     are inherited randomly from both parents. In the (1-crossover.probability_multipoint)
    cases, weights of the new connection genes are the mean of the corresponding parent
    genes



%mutation
mutation.probability_add_node=0.03;
mutation.probability_add_connection=0.05;
```

```
mutation.probability_recurrency=0.0; %if we are in add_connection_mutation, this governs if
    a recurrent connection is allowed. Note: this will only activate if the random
    connection is a recurrent one, otherwise the connection is simply accepted. If no
    possible non-recurrent connections exist for the current node genes, then for e.g. a
    probability of 0.1, 9 times out of 10 no connection is added.
mutation.probability_mutate_weight=0.9;
mutation.weight_cap=8; % weights will be restricted from -mutation.weight_cap to mutation.
    weight_cap
mutation.weight_range=5; % random distribution with width mutation.weight_range, centered on
     0. mutation range of 5 will give random distribution from -2.5 to 2.5
mutation.probability_gene_reenabled=0.25; % Probability of a connection gene being reenabled
     in offspring if it was inherited disabled


%%%%%%%%%%%%%%%%main algorithm %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
if load_flag==0

    %call function to create initial population
    %for information about the make-up of the population structure and the innovation_record
        , look at initial_population.m
    [population,innovation_record]=initial_population(population_size, number_input_nodes,
        number_output_nodes, vector_connected_input_nodes);

    %initial speciation
    number_connections=(length(vector_connected_input_nodes)+1)*number_output_nodes;
    %put first individual in species one and update species_record
    population(1).species=1;
    matrix_reference_individuals=population(1).connectiongenes(4,:); %species reference
        matrix (abbreviated, only weights, since there are no topology differences in
        initial population)
    species_record(1).ID=1;
    species_record(1).number_individuals=1;

    %Loop through rest of individuals and either assign to existing species or create new
        species and use first individual of new species as reference
    for index_individual=2:size(population,2);
        assigned_existing_species_flag=0;
        new_species_flag=0;
        index_species=1;
        while assigned_existing_species_flag==0 & new_species_flag==0 %loops through the
            existing species, terminates when either the individual is assigned to existing
            species or there are no more species to test it against, which means it is a new
             species
            distance=speciation.c3*sum(abs(population(index_individual).connectiongenes(4,:)
                -matrix_reference_individuals(index_species,:)))/number_connections; %
                computes compatibility distance, abbreviated, only average weight distance
                considered
            if distance<speciation.threshold %If within threshold, assign to the existing
                species
```

```
                population(index_individual).species=index_species;
                assigned_existing_species_flag=1;
                species_record(index_species).number_individuals=species_record(
                    index_species).number_individuals+1;
            end
            index_species=index_species+1;
            if index_species>size(matrix_reference_individuals,1) &
                assigned_existing_species_flag==0 %Outside of species references, must be
                new species
                new_species_flag=1;
            end
        end
        if new_species_flag==1 %check for new species, if it is, update the species_record
            and use individual as reference for new species
            population(index_individual).species=index_species;
            matrix_reference_individuals=[matrix_reference_individuals;population(
                index_individual).connectiongenes(4,:)];
            species_record(index_species).ID=index_species;
            species_record(index_species).number_individuals=1; %if number individuals in a
                species is zero, that species is extinct

        end
    end
    generation=1;
else % start with saved version of evolution
    load 'neatsave'
end


%%% Main loop
flag_solution=0;
% all_time_best_individual = population(1);
while generation<maxgeneration && flag_solution==0

    if save_flag==1 % Backup copies of current generation
        save 'neatsave' population generation innovation_record species_record CM simulation
             fitness_record
        %all_time_best_individual
    end

    % call evaluation function (in this case XOR), fitnesses of individuals will be stored
        in population(:).fitness
    % IMPORTANT reproduction assumes an (all positive!) evaluation function where a higher
        value means better fitness (in other words, the algorithm is geared towards
        maximizing a fitness function which can only assume values between 0 and +Inf)
    parfor individual=1:size(population,2)
        population(individual).fitness = simulator(CM,simulation,population(individual));
    end
```

```
fitness_record(1,generation)=max([population(:).fitness]);

generation

%compute mean and max raw fitnesses in each species and store in species_record.
    generation_record
max_fitnesses_current_generation=zeros(1,size(species_record,2));

for index_species=1:size(species_record,2)
    if species_record(index_species).number_individuals>0
        [max_fitness,index_individual_max]=max(([population(:).species]==index_species)
            .*[population(:).fitness]);
        mean_fitness=sum(([population(:).species]==index_species).*[population(:).
            fitness])/species_record(index_species).number_individuals;
        % Compute stagnation vector (last stagnation.number_generation-1 max fitnesses
            plus current fitness
        if size(species_record(index_species).generation_record,2)>stagnation.
            number_generation-2
            stagnation_vector=[species_record(index_species).generation_record(3,size(
                species_record(index_species).generation_record,2)-stagnation.
                number_generation+2:size(species_record(index_species).generation_record
                ,2)),max_fitness];
            if sum(abs(stagnation_vector-mean(stagnation_vector))<stagnation.threshold)
                ==stagnation.number_generation %Check for stagnation
                mean_fitness=0.01; %set mean fitness to small value to eliminate species
                     (cannot be set to 0, if only one species is present, we would have
                    divide by zero in fitness sharing. anyways, with only one species
                    present, we have to keep it)
            end
        end
        species_record(index_species).generation_record=[species_record(index_species).
            generation_record,[generation;mean_fitness;max_fitness;index_individual_max
            ]];
        max_fitnesses_current_generation(1,index_species)=max_fitness;
    end
end

%check for refocus
[top_fitness,index_top_species]=max(max_fitnesses_current_generation);
if size(species_record(index_top_species).generation_record,2)>refocus.number_generation
    index1=size(species_record(index_top_species).generation_record,2)-refocus.
        number_generation;
    index2=size(species_record(index_top_species).generation_record,2);
    if sum(abs(species_record(index_top_species).generation_record(3,index1:index2)-mean
        (species_record(index_top_species).generation_record(3,index1:index2)))<refocus.
        threshold)==refocus.number_generation
        [discard,vector_cull]=sort(-max_fitnesses_current_generation);
        vector_cull=vector_cull(1,3:sum(max_fitnesses_current_generation>0));
        for index_species=1:size(vector_cull,2)
```

```
            index_cull=vector_cull(1,index_species);
            species_record(index_cull).generation_record(2,size(species_record(
                index_cull).generation_record,2))=0.01;
        end
    end
end

%visualisation fitness & species
a=0;
b=0;
for index_individual=1:size(population,2)
    a=a+sum(population(index_individual).connectiongenes(5,:)==1);
    b=b+sum(population(index_individual).nodegenes(2,:)==3);
end
average_number_non_disabled_connections=[average_number_non_disabled_connections,[a/
    population_size;generation]];
average_number_hidden_nodes=[average_number_hidden_nodes,[b/population_size;generation
    ]];
c=[];
for index_species=1:size(species_record,2)
    c=[c,species_record(index_species).generation_record(1:3,size(species_record(
        index_species).generation_record,2))];
end
max_overall_fitness=[max_overall_fitness,[max(c(3,:).*(c(1,:)==generation));generation
    ]];
maximale_fitness=max(c(3,:).*(c(1,:)==generation))

subplot(2,2,1);
plot(average_number_non_disabled_connections(2,:),
    average_number_non_disabled_connections(1,:));
ylabel('non disabled connections');
subplot(2,2,2);
plot(average_number_hidden_nodes(2,:),average_number_hidden_nodes(1,:));
ylabel('num hidden nodes');
subplot(2,2,3);
plot(max_overall_fitness(2,:),max_overall_fitness(1,:));
ylabel('max fitness');
drawnow;

if flag_solution==0
    %call reproduction function with parameters, current population and species record,
        returns new population, new species record and new innovation record
    [population,species_record,innovation_record]=reproduce(population, species_record,
        innovation_record, initial, selection, crossover, mutation, speciation,
        generation, population_size);
    toc;
end
%increment generational counter
generation=generation+1;
```

```
end

saveas(gcf,'neatsave.fig','fig') % save figure

if matlabpool('size') > 0
    matlabpool close
end
```

B.2. Código NEAT

```
%%%%%%%%%%%%%%%%%%%%%%Generate minimal initial population

%% Neuro_Evolution_of_Augmenting_Topologies - NEAT
%% developed by Kenneth Stanley (kstanley@cs.utexas.edu) & Risto Miikkulainen (risto@cs.
    utexas.edu)
%% Coding by Christian Mayr (matlab_neat@web.de)

function [population,innovation_record]=initial_population(number_individuals,
    number_input_nodes,number_output_nodes,vector_connected_input_nodes);

% nodegenes is array 4rows * (number_input_nodes+number_output_nodes+hidden-nodes(not
    existent in initial population) +1 (bias-node))columns
% nodegenes contains consecutive node ID's (upper row), node type (lower row) 1=input 2=
    output 3=hidden 4=bias, node input state, and node output state (used for evaluation,
    all input states zero initially, except bias node, which is always 1)
% connectiongenes is array 5 rows * number_connections columns
% from top to bottom, those five rows contain: innovation number, connection from,
    connection to, weight, enable bit
% the rest of the elements in the structure for an individual should be self-explanatory

% innovation_record tracks innovations in a 5rows by (number of innovations) columns matrix,
     contains innovation number, connect_from_node as well as connect_to_node for this
    innovation)
% the new node (if it is a new node mutation, then this node will appear in the 4th row when
     it is first connected. There will always be two innovations with one node mutation,
    since there is a connection to and from the new node.
% In the initial population, this will be abbreviated to the Node with the highest number
    appearing in the last column of the record, since only this is needed as starting point
    for the rest of the algorithm),
% and 5th row is generation this innovation occured (generation is assumed to be zero for
    the innovations in the initial population)

%compute number and matrix of initial connections (all connections between output nodes and
    the nodes listed in vector_connected_input_nodes)
number_connections=(length(vector_connected_input_nodes)+1)*number_output_nodes;
vector_connection_from=rep([vector_connected_input_nodes,number_input_nodes+1],[1
    number_output_nodes]);
vector_connection_to=[];
```

```
RandStream.setDefaultStream(RandStream('mt19937ar','seed',sum(100*clock)));


for index_output_node=(number_input_nodes+2):(number_input_nodes+1+number_output_nodes)
   vector_connection_to=[vector_connection_to,index_output_node*ones(1,length(
       vector_connected_input_nodes)+1)];
end
connection_matrix=[vector_connection_from;
                   vector_connection_to];

for index_individual=1:number_individuals;
   population(index_individual).nodegenes=[1:(number_input_nodes+1+number_output_nodes); %
       +1 node correspond to the bias
                                           ones(1,number_input_nodes),4,2*ones(1,
                                               number_output_nodes);
                                           zeros(1,number_input_nodes),1,zeros(1,
                                               number_output_nodes);
                                           zeros(1,number_input_nodes+1+number_output_nodes)
                                               ];
   population(index_individual).connectiongenes=[1:number_connections;
                                                  connection_matrix;
                                                  rand(1,number_connections)*2-1;
                                                  ones(1,number_connections)]; %all weights
                                                      uniformly distributed in [-1 +1], all
                                                      connections enabled
   population(index_individual).fitness=0;
   population(index_individual).species=0;
end
innovation_record=[population(index_individual).connectiongenes(1:3,:);zeros(size(population
    (index_individual).connectiongenes(1:2,:)))];
innovation_record(4,size(innovation_record,2))=max(population(1).nodegenes(1,:)); %highest
    node ID for initial population


% REP.m         Replicate a matrix
%
% This function replicates a matrix in both dimensions.
%
% Syntax:       MatOut = rep(MatIn,REPN);
%
% Input parameters:
%   MatIn    - Input Matrix (before replicating)
%
%   REPN     - Vector of 2 numbers, how many replications in each dimension
%              REPN(1): replicate vertically
%              REPN(2): replicate horizontally
%
%              Example:
%
%              MatIn = [1 2 3]
```

```
%              REPN = [1 2]: MatOut = [1 2 3 1 2 3]
%              REPN = [2 1]: MatOut = [1 2 3;
%                                      1 2 3]
%              REPN = [3 2]: MatOut = [1 2 3 1 2 3;
%                                      1 2 3 1 2 3;
%                                      1 2 3 1 2 3]
%
% Output parameter:
%   MatOut   - Output Matrix (after replicating)
%

% Author:   Carlos Fonseca & Hartmut Pohlheim
% History:  14.02.94        file created


function MatOut = rep(MatIn,REPN)

% Get size of input matrix
   [N_D,N_L] = size(MatIn);

% Calculate
   Ind_D = rem(0:REPN(1)*N_D-1,N_D) + 1;
   Ind_L = rem(0:REPN(2)*N_L-1,N_L) + 1;

% Create output matrix
   MatOut = MatIn(Ind_D,Ind_L);


% End of function


%% Reproduction -Main Evolutionary algorithm (Mutation, crossover, speciation)

%% Neuro_Evolution_of_Augmenting_Topologies - NEAT
%% developed by Kenneth Stanley (kstanley@cs.utexas.edu) & Risto Miikkulainen (risto@cs.
    utexas.edu)
%% Coding by Christian Mayr (matlab_neat@web.de)

function [new_population,updated_species_record,updated_innovation_record]=reproduce(
    population, species_record, innovation_record, initial, selection, crossover, mutation,
    speciation, generation, population_size);

% the following 'for' loop has these three objectives:

% 1.compute matrix of existing and propagating species from species_record (first row),
    assign their alloted number of offspring from the shared fitness (second row), and set
    their actual number of offspring to zero (third row) (will be incremented as new
    individuals are created from this species)

% 2.copy most fit individual in every species with more than initial.number_copy individuals
```

```
     unchanged into new generation (elitism) (But only if species is not dying out, i.e. has
     at least one individual allotted to itself in the new generation)
%   utilizes data stored in species_record.generation_record (index of individual in
    population having the highest fitness)

% 3.erase lowest percentage (initial.kill_percentage) in species with more than initial.
    number_for_kill individuals to keep them from taking part in reproduction
% Matlab actually doesn't offer a facility for redirecting the pointers which link one
    element in a structure with the next, so essentially this individual entry in the
    population structure cannot be erased. Rather, it will have its species ID set to zero,
% which has the same effect of removing it from the rest of the reproduction cycle, since
    all reproduction functions access the population structure through the species ID and no
     species has an ID of zero

% Compute sum_average_fitnesses
sum_average_fitnesses=0;
for index_species=1:size(species_record,2)
   sum_average_fitnesses=sum_average_fitnesses+species_record(index_species).
       generation_record(2,size(species_record(index_species).generation_record,2))*(
       species_record(index_species).number_individuals>0);
end
% The following two lines only initialize the new_population structure. Since its species is
     set to 0, the rest of the algorithm will not bother with it. It gets overwritten as
    soon as the first really new individual is created
new_population(1)=population(1);
new_population(1).species=0;


overflow=0;
index_individual=0;
matrix_existing_and_propagating_species=[];
for index_species=1:size(species_record,2)
   if species_record(index_species).number_individuals>0 %test if species existed in old
       generation
      number_offspring=species_record(index_species).generation_record(2,size(species_record
          (index_species).generation_record,2))/sum_average_fitnesses*population_size; %
          compute number of offspring in new generation
      overflow=overflow+number_offspring-floor(number_offspring);
      if overflow>=1 %Since new species sizes are fractions, overflow sums up the difference
           between size and floor(size), and everytime this overflow is above 1, the species
           gets one additional individual alotted to it
         number_offspring=ceil(number_offspring);
         overflow=overflow-1;
      else
         number_offspring=floor(number_offspring);
      end
      if number_offspring>0 %Check to see if species is dying out, only add those species to
           matrix_existing_and_propagating_species which have offspring in the new
          generation
```

```
        matrix_existing_and_propagating_species=[matrix_existing_and_propagating_species,[
            species_record(index_species).ID;number_offspring;0]]; %matrix (objective 1)
        if species_record(index_species).number_individuals>=initial.number_copy %check for
             condition for objective 2
           index_individual=index_individual+1;
           new_population(index_individual)=population(species_record(index_species).
               generation_record(4,size(species_record(index_species).generation_record,2))
               ); % Objective 2
           matrix_existing_and_propagating_species(3,size(
               matrix_existing_and_propagating_species,2))=1; %Update
               matrix_existing_and_propagating_species
        end
     end

     if (species_record(index_species).number_individuals>initial.kill_percentage) & (ceil(
         species_record(index_species).number_individuals*(1-initial.kill_percentage))>2) %
         check condition for objective 3
        matrix_individuals_species=[find([population(:).species]==index_species);[
            population(find([population(:).species]==index_species)).fitness]];
        [sorted_fitnesses_in_species,sorting_vector]=sort(matrix_individuals_species(2,:));
        matrix_individuals_species=matrix_individuals_species(:,sorting_vector);
        sorting_vector=matrix_individuals_species(1,:);

        for index_kill=1:floor(species_record(index_species).number_individuals*initial.
            kill_percentage)
           population(sorting_vector(index_kill)).species=0; %objective 3
        end
     end

  end
end

% generate reference of random individuals from every species from old population
% cycle through species ID's, and add reference individuals from old population, new species
     of new population will be added during reproduction
index_ref=0;
for index_species_ref=1:size(species_record,2)
  if sum([population(:).species]==index_species_ref)>0 %Check if species exists in old
       population
     index_ref=index_ref+1;
     [discard,index_ref_old]=max(([population(:).species]==index_species_ref).*rand(1,size(
         population,2)));
     population_ref(index_ref)=population(index_ref_old);
  end
end
matrix_existing_and_propagating_species

%% Standard-reproduction (Main)
%% Cycle through all existing species
```

```
for index_species=1:size(matrix_existing_and_propagating_species,2)
   count_individuals_species=0;
   species_ID=matrix_existing_and_propagating_species(1,index_species); %this is ID of
       species which will be reproduced this cycle
   %IMPORTANT: index_species only has relevance to matrix_existing_and_propagating_species,
       all other mechanisms using species in some way must use species_ID

   % Linear Ranking and stochastic universal sampling
   % Ranking with selection.pressure
   fitnesses_species=[population(find([population(:).species]==species_ID)).fitness];
   index_fitnesses_species=find([population(:).species]==species_ID);
   [discard,sorted_fitnesses]=sort(fitnesses_species);
   ranking=zeros(1,size(fitnesses_species,2));
   ranking(sorted_fitnesses)=1:size(fitnesses_species,2);
   if size(fitnesses_species,2)>1
       FitnV=(2-selection.pressure+2*(selection.pressure-1)/(size(fitnesses_species,2)-1)*(
           ranking-1))';
   else
       FitnV=2;
   end
   % stochastic universal sampling
   % First compute number of individuals to be selected (two parents required for every
       offspring through crossover, one for mutation)
   number_overall=matrix_existing_and_propagating_species(2,index_species)-
       matrix_existing_and_propagating_species(3,index_species);
   number_crossover=round(crossover.percentage*number_overall);
   number_mutate=number_overall-number_crossover;
   Nind=size(fitnesses_species,2);
   Nsel=2*number_crossover+number_mutate;

   if Nsel==0 %rare case, in which a species with at least initial.number_copy individuals
       gets individual copied, but compares poorly to new generation, which results in this
       copied individual being
      %the only individual of this species in new generation, so no crossover or mutation
          takes place.
      %setting Nsel to 1 will prevent division by zero error, but will have no other effect
          since the propagation loop is governed by matrix_existing_and_propagating_species,
           not by Nsel
      Nsel=1;
   end

   % Perform stochastic universal sampling (Code Snippet from Genetic Algorithm toolbox 1.2
       by Chipperfield et al)
   cumfit = cumsum(FitnV);
   trials = cumfit(Nind) / Nsel * (rand + (0:Nsel-1)');
   Mf = cumfit(:, ones(1, Nsel));
   Mt = trials(:, ones(1, Nind))';
   [NewChrIx, ans] = find(Mt < Mf & [ zeros(1, Nsel); Mf(1:Nind-1, :) ] <= Mt);
   % Shuffle selected Individuals
```

```
[ans, shuf] = sort(rand(Nsel, 1));
NewChrIx = NewChrIx(shuf);
% relate to indexes of individuals in population
NewChrIx = index_fitnesses_species(NewChrIx);


while matrix_existing_and_propagating_species(3,index_species)<
    matrix_existing_and_propagating_species(2,index_species) %cycle until actual number
    of offspring has reached allotted number of offspring
   index_individual=index_individual+1;
   count_individuals_species=count_individuals_species+1;
   % Crossover
   if count_individuals_species<=number_crossover %O.k: we are doing crossover
      %Select Parents
      %select parent1
      parent1=population(NewChrIx(2*count_individuals_species-1));
      %select parent2
      found_parent2=0;
      %sum([species_record(:).number_individuals]>0)
      if (rand<crossover.probability_interspecies) & (size(
          matrix_existing_and_propagating_species,2)>1) %select parent2 from other
          species (can only be done if there is more than one species in old population)
         while found_parent2==0;
            [discard,index_parent2]=max(rand(1,size(population,2)));
            parent2=population(index_parent2);
            found_parent2=((parent2.species~=0) & (parent2.species~=parent1.species)); %
                check if parent2.species is not species 0 (deleted individual) or species
                 of parent1
         end
         parent2.fitness=parent1.fitness; %set fitnesses to same to ensure that disjoint
             and excess genes are inherited fully from both parents (tip from ken)
      else % O.K. we take parent2 from same species as parent1
          parent2=population(NewChrIx(2*count_individuals_species));
      end

      %Crossover
      %inherit nodes from both parents
      new_individual.nodegenes=[];
      matrix_node_lineup=[[parent1.nodegenes(1,:);1:size(parent1.nodegenes,2);zeros(1,
          size(parent1.nodegenes,2))],[parent2.nodegenes(1,:);zeros(1,size(parent2.
          nodegenes,2));1:size(parent2.nodegenes,2)]];
      [discard,sort_node_vec]=sort(matrix_node_lineup(1,:));
      matrix_node_lineup=matrix_node_lineup(:,sort_node_vec);
      node_number=0;
      for index_node_sort=1:size(matrix_node_lineup,2)
         if node_number~=matrix_node_lineup(1,index_node_sort)
            if matrix_node_lineup(2,index_node_sort)>0
               new_individual.nodegenes=[new_individual.nodegenes,parent1.nodegenes(:,
                   matrix_node_lineup(2,index_node_sort))];
```

```
      else
         new_individual.nodegenes=[new_individual.nodegenes,parent2.nodegenes(:,
             matrix_node_lineup(3,index_node_sort))];
      end
      node_number=matrix_node_lineup(1,index_node_sort);
   end
end
%Crossover connection genes
%first do lineup of connection genes
matrix_lineup=[[parent1.connectiongenes(1,:);1:size(parent1.connectiongenes,2);
    zeros(1,size(parent1.connectiongenes,2))],[parent2.connectiongenes(1,:);zeros
    (1,size(parent2.connectiongenes,2));1:size(parent2.connectiongenes,2)]];
[discard,sort_vec]=sort(matrix_lineup(1,:));
matrix_lineup=matrix_lineup(:,sort_vec);
final_matrix_lineup=[];
innovation_number=0;
for index_sort=1:size(matrix_lineup,2)
   if innovation_number~=matrix_lineup(1,index_sort)
      final_matrix_lineup=[final_matrix_lineup,matrix_lineup(:,index_sort)];
      innovation_number=matrix_lineup(1,index_sort);
   else
      final_matrix_lineup(2:3,size(final_matrix_lineup,2))=final_matrix_lineup(2:3,
          size(final_matrix_lineup,2))+matrix_lineup(2:3,index_sort);
   end
end
% O.K. Connection Genes are lined up, start with crossover
new_individual.connectiongenes=[];

for index_lineup=1:size(final_matrix_lineup,2)
   if (final_matrix_lineup(2,index_lineup)>0) & (final_matrix_lineup(3,index_lineup
       )>0) %check for matching genes, do crossover
      if rand<0.5 %random crossover for matching genes
         new_individual.connectiongenes=[new_individual.connectiongenes,parent1.
             connectiongenes(:,final_matrix_lineup(2,index_lineup))];
      else
         new_individual.connectiongenes=[new_individual.connectiongenes,parent2.
             connectiongenes(:,final_matrix_lineup(3,index_lineup))];
      end
      if rand>crossover.probability_multipoint %weight averaging for offspring,
          otherwise the randomly inherited weights are left undisturbed
         new_individual.connectiongenes(4,size(new_individual.connectiongenes,2))=(
             parent1.connectiongenes(4,final_matrix_lineup(2,index_lineup))+parent2
             .connectiongenes(4,final_matrix_lineup(3,index_lineup)))/2;
      end
   end
   parent1_flag=sum(final_matrix_lineup(2,index_lineup+1:size(final_matrix_lineup
       ,2)));  % test if there exist further connection genes from index_lineup+1
       to end of final_matrix_lineup for parent1 (to detect excess)
   parent2_flag=sum(final_matrix_lineup(3,index_lineup+1:size(final_matrix_lineup
```

```
          ,2)));  % test if there exist further connection genes from index_lineup+1
          to end of final_matrix_lineup for parent1 (to detect excess)
       % Two cases to check (excess is taken care of in the disjoint gene checks)
       if (final_matrix_lineup(2,index_lineup)>0) & (final_matrix_lineup(3,index_lineup
           )==0) %Disjoint parent1
          if parent1.fitness>=parent2.fitness
             new_individual.connectiongenes=[new_individual.connectiongenes,parent1.
                 connectiongenes(:,final_matrix_lineup(2,index_lineup))];
          end
       end
       if (final_matrix_lineup(2,index_lineup)==0) & (final_matrix_lineup(3,
           index_lineup)>0) %Disjoint parent2
          if parent2.fitness>=parent1.fitness
             new_individual.connectiongenes=[new_individual.connectiongenes,parent2.
                 connectiongenes(:,final_matrix_lineup(3,index_lineup))];
          end
       end
    end
    new_individual.fitness=0; %has no impact on algorithm, only required for assignment
        to new population
    new_individual.species=parent1.species; %will be species hint for speciation
else % no crossover, just copy a individual of the species and mutate in subsequent
    steps
    new_individual=population(NewChrIx(number_crossover+count_individuals_species));
end

% Hidden nodes culling (remove any hidden nodes where there is no corresponding
    connection gene in the new individual)
connected_nodes=[];
for index_node_culling=1:size(new_individual.nodegenes,2)
    node_connected_flag=sum(new_individual.connectiongenes(2,:)==new_individual.
        nodegenes(1,index_node_culling))+sum(new_individual.connectiongenes(3,:)==
        new_individual.nodegenes(1,index_node_culling));
    if (node_connected_flag>0) | (new_individual.nodegenes(2,index_node_culling)~=3);
        connected_nodes=[connected_nodes,new_individual.nodegenes(:,index_node_culling
            )];
    end
end
new_individual.nodegenes=connected_nodes;

% Disabled Genes Mutation
%run through all connection genes in a new_individual, find disabled connection genes,
     enable again with crossover.probability_gene_reenabled probability
for index_connection_gene=1:size(new_individual.connectiongenes,2)
   if (new_individual.connectiongenes(5,index_connection_gene)==0)& (rand<mutation.
       probability_gene_reenabled)
      new_individual.connectiongenes(5,index_connection_gene)=1;
   end
end
```

```
% Weight Mutation
%run through all connection genes in a new_individual, decide on mutating or not
for index_connection_gene=1:size(new_individual.connectiongenes,2)
   if rand<mutation.probability_mutate_weight %*index_connection_gene/size(
       new_individual.connectiongenes,2) %linearly biased towards higher probability
       of mutation at end of connection genes
      new_individual.connectiongenes(4,index_connection_gene)=new_individual.
          connectiongenes(4,index_connection_gene)+mutation.weight_range*(rand-0.5);
   end
   % weight capping
   new_individual.connectiongenes(4,index_connection_gene)=new_individual.
       connectiongenes(4,index_connection_gene)*(abs(new_individual.connectiongenes(4,
       index_connection_gene))<=mutation.weight_cap)+(sign(new_individual.
       connectiongenes(4,index_connection_gene))*mutation.weight_cap)*(abs(
       new_individual.connectiongenes(4,index_connection_gene))>mutation.weight_cap);
end

% IMPORTANT: The checks for duplicate innovations in the following two types of
    mutation can only check in the current generation
% Add Connection Mutation
flag_recurrency_enabled=rand<mutation.probability_recurrency;
vector_possible_connect_from_nodes=new_individual.nodegenes(1,:); %connections can run
     from every node
vector_possible_connect_to_nodes=new_individual.nodegenes(1,find((new_individual.
    nodegenes(2,:)==2)+(new_individual.nodegenes(2,:)==3))); %connections can only run
     into hidden and output nodes
number_possible_connection=length(vector_possible_connect_from_nodes)*length(
    vector_possible_connect_to_nodes)-size(new_individual.connectiongenes,2);

flag1=(rand<mutation.probability_add_node);

if (rand<mutation.probability_add_connection) & (number_possible_connection>0) & (
    flag1==0) %check if new connections can be added to genes (if there are any
    possible connections which are not already existing in genes of new individual)
   % First build matrix containing all possible new connection for nodegene of new
       individual

   new_connection_matrix=[];
   for index_connect_from=1:length(vector_possible_connect_from_nodes)
      for index_connect_to=1:length(vector_possible_connect_to_nodes)
         possible_connection=[vector_possible_connect_from_nodes(index_connect_from);
             vector_possible_connect_to_nodes(index_connect_to)];
         if sum((new_individual.connectiongenes(2,:)==possible_connection(1)).*(
             new_individual.connectiongenes(3,:)==possible_connection(2)))==0 % Check
             if proposed connection is not already contained in gene
            new_connection_matrix=[new_connection_matrix,possible_connection];
         end
      end
```

```
end
% Shuffle possible new connections randomly
[discard,shuffle]=sort(rand(1,size(new_connection_matrix,2)));
new_connection_matrix=new_connection_matrix(:,shuffle);

index_new_connection=0;
flag_connection_ok=0;
% check if connection is o.k. (meaning either non-recurrent or recurrent and
    flag_recurrency_enabled set to 1) if not connection is found which is o.k.,no
    connection will be added to connection genes of new individual
while (flag_connection_ok==0) & (index_new_connection<size(new_connection_matrix,2)
    )
   index_new_connection=index_new_connection+1;
   new_connection=new_connection_matrix(:,index_new_connection);

   % test new connection if it is recurrent (i.e. at least one of the possibles
       path starting from connect_to node in the network leads back to the
       connect_from node
   flag_recurrent=0;
   if new_connection(1)==new_connection(2) %trivial recurrency
      flag_recurrent=1;
   end
   nodes_current_level=new_connection(2);
   depth=0;
   while flag_recurrent==0 & depth<size(new_individual.connectiongenes,2) & ~
       isempty(nodes_current_level)
      depth=depth+1;
      nodes_next_level=[];
      for index_check=1:size(nodes_current_level);
         nodes_next_level=[nodes_next_level,new_individual.connectiongenes(3,find(
             new_individual.connectiongenes(2,:)==nodes_current_level(index_check))
             )];
      end
      if sum(nodes_next_level(:)==new_connection(1))>0
         flag_recurrent=1;
      end
      nodes_current_level=nodes_next_level;
   end
   if flag_recurrent==0
      flag_connection_ok=1;
   elseif flag_recurrency_enabled
      flag_connection_ok=1;
   end
end

% Now we test if it is a true innovation (i.e. hasn't already happened in this
    generation) we can only do this if a valid new connection has been found
if flag_connection_ok
   index_already_happened=find((innovation_record(5,:)==generation).*(
```

```
            innovation_record(2,:)==new_connection(1)).*(innovation_record(3,:)==
            new_connection(2))); %set flag signifying new innovation (connection not
            contained in innovation_record of this generation)
         new_innovation=not(sum(index_already_happened));
         if new_innovation==1 % O.K. is new innovation
            new_connection=[max(innovation_record(1,:))+1;new_connection]; %Update the
                new connection with its innovation number
            % Update connection_genes
            new_individual.connectiongenes=[new_individual.connectiongenes,[
                new_connection;rand*2-1;1]];
            % Update innovation_record
            innovation_record=[innovation_record,[new_connection;0;generation]];
         else % connection gene already exists in innovation_record of this generation
            % Update connection_genes
            new_individual.connectiongenes=[new_individual.connectiongenes,[
                innovation_record(1:3,index_already_happened);rand*2-1;1]];
         end
      end
   end

   % Add (Insert) Node Mutation
   new_innovation=0;
   if flag1==1
      max_old_innovation_number=max((innovation_record(5,:)<generation).*
          innovation_record(1,:)); %highest innovation number from last generation (to
          ensure that only connections from from last generation or older are chosen for
          add node mutation, otherwise a new connection added in the last mutation might
          instantly be disabled)
      vector_possible_connections=[new_individual.connectiongenes(2:3,find((
          new_individual.connectiongenes(5,:)==1) & (new_individual.connectiongenes(1,:)
          <=max_old_innovation_number)));find((new_individual.connectiongenes(5,:)==1) &
          (new_individual.connectiongenes(1,:)<=max_old_innovation_number))];  %compute
          vector of connections into which a new node could be inserted and their
          positions in the connection_gene matrix. This vector is composed of all
          nondisabled connections which stem at least from the last generation or older
      insert_node_connection=vector_possible_connections(:,round(rand*size(
          vector_possible_connections,2)+0.5));
      new_innovation=1; %set provisionally to 1, will be checked
      exist_innovation=find((innovation_record(5,:)==generation).*(innovation_record(4,:)
          >0).*(innovation_record(2,:)==insert_node_connection(1))); %Beginning of check
          innovation record to test for real innovation. exist_innovation contains vector
           of index of elements in innovation record which fulfil three things: current
          generation, add node mutation and same connect from as current innovation
      if sum(exist_innovation)>0 %if these are fulfilled, we have to test for connect_to
          node to see if innovation really is the same
         for index_check=1:length(exist_innovation)
            if innovation_record(3,exist_innovation(index_check)+1)==
                insert_node_connection(2)
               new_innovation=0;
```

```
            index_already_existent_this_generation=exist_innovation(index_check);
         end
      end
   end
   if new_innovation==1 %O.K. is true innovation for current generation
      % Update node_genes
      new_node_number=max(innovation_record(4,:))+1;
      new_individual.nodegenes=[new_individual.nodegenes,[new_node_number;3;0;0]];
      % Update connection_genes
      new_individual.connectiongenes(5,insert_node_connection(3))=0; %disable old
          connection gene
      new_connections=[[max(innovation_record(1,:))+1;insert_node_connection(1);
          new_node_number;1;1],[max(innovation_record(1,:))+2;new_node_number;
          insert_node_connection(2);new_individual.connectiongenes(4,
          insert_node_connection(3));1]];
      new_individual.connectiongenes=[new_individual.connectiongenes,new_connections];
           %extend connection_genes by the two new connections
      % Update innovation_record
      innovation_record=[innovation_record,[new_connections(1:3,:);new_node_number,0;
          generation,generation]];
   else %no new innovation, has already happened at least once in this generation
      % Update node_genes
      node_number=innovation_record(4,index_already_existent_this_generation);
      new_individual.nodegenes=[new_individual.nodegenes,[node_number;3;0;0]];
      % Update connection_genes
      new_individual.connectiongenes(5,insert_node_connection(3))=0; %disable old
          connection gene
      new_connections=[innovation_record(1:3,index_already_existent_this_generation:
          index_already_existent_this_generation+1);1,new_individual.connectiongenes
          (4,insert_node_connection(3));1,1];
      length_con_gen=size(new_individual.connectiongenes,2); %length of the connection
           genes of current new_individual
      if new_individual.connectiongenes(1,length_con_gen)>new_connections(1,2) % check
           if there was an add_connection_mutation to current new_individual which has
           a higher innovation number than current add_node_mutation
          new_individual.connectiongenes=[new_individual.connectiongenes(:,1:
              length_con_gen-1),new_connections,new_individual.connectiongenes(:,
              length_con_gen)];


      else
         new_individual.connectiongenes=[new_individual.connectiongenes,
             new_connections];
      end
   end
end

%% Speciation
% Loop through comparison vector
```

```
species_assigned=0;
index_population_ref=0;
while species_assigned==0 & index_population_ref<size(population_ref,2)
   %extract reference_individual from reference population
   index_population_ref=index_population_ref+1;
   reference_individual=population_ref(index_population_ref);
   %run through both connection genes, compute disjoint, excess, and average weight
       difference
   max_num_genes=max([size(new_individual.connectiongenes,2),size(reference_individual
       .connectiongenes,2)]);
   max_num_innovation=max([new_individual.connectiongenes(1,:),reference_individual.
       connectiongenes(1,:)]);
   vector_innovation_new=[zeros(1,max(new_individual.connectiongenes(1,:))),ones(1,
       max_num_innovation-max(new_individual.connectiongenes(1,:)))];
   vector_innovation_new(new_individual.connectiongenes(1,:))=2;
   vector_weight_new=zeros(1,max_num_innovation);
   vector_weight_new(new_individual.connectiongenes(1,:))=new_individual.
       connectiongenes(4,:);
   vector_innovation_ref=[4*ones(1,max(reference_individual.connectiongenes(1,:))),8*
       ones(1,max_num_innovation-max(reference_individual.connectiongenes(1,:)))];
   vector_innovation_ref(reference_individual.connectiongenes(1,:))=16;
   vector_weight_ref=zeros(1,max_num_innovation);
   vector_weight_ref(reference_individual.connectiongenes(1,:))=reference_individual.
       connectiongenes(4,:);
   vector_lineup=vector_innovation_new+vector_innovation_ref;
   excess=sum(vector_lineup==10)+sum(vector_lineup==17);
   disjoint=sum(vector_lineup==6)+sum(vector_lineup==16);
   vector_matching=find(vector_lineup==18);
   average_weight_difference=sum(abs(vector_weight_new(vector_matching)-
       vector_weight_ref(vector_matching)))/length(vector_matching);
   max_num_genes=1;
   distance=speciation.c1*excess/max_num_genes+speciation.c2*disjoint/max_num_genes+
       speciation.c3*average_weight_difference;
   if distance<speciation.threshold
      % assign individual to same species as current reference individual

      new_individual.species=reference_individual.species;
      species_assigned=1; %set flag indicating new_individual has been assigned to
          species
   end
end
% not compatible with any? well, then create new species
if species_assigned==0
   new_species_ID=size(species_record,2)+1;
             % assign individual to new species
   new_individual.species=new_species_ID;
   % update species_record
   species_record(new_species_ID).ID=new_species_ID;
   species_record(new_species_ID).number_individuals=1;
```

```
         species_record(new_species_ID).generation_record=[];
         % update population reference
         population_ref(size(population_ref,2)+1)=new_individual;
      end

      % add new_individual to new_population
      new_population(index_individual)=new_individual;

      %Increment species
      matrix_existing_and_propagating_species(3,index_species)=
          matrix_existing_and_propagating_species(3,index_species)+1;
   end
end

% final update of species_record (can only be done now since old population sizes were
    needed during reproduction cycle)
for index_species=1:size(species_record,2)
   species_record(index_species).number_individuals=sum([new_population(:).species]==
       index_species);
end
%assign updated species_record to output
updated_species_record=species_record;
%assign updated innovation_record to output
updated_innovation_record=innovation_record;


%clear;

path(path,'./Sim')

load neatsave

best_ind = population(find([population(:).fitness] == ...
    max([population(:).fitness])))

simulation.plot = 'true'

simulator(CM,simulation,best_ind)
```

B.3. Código Simulador NEAT

```
function individual_fitness = simulator(CM, options, individual)

do_plot = strcmpi(options.plot,'true');
secuential_activation = strcmpi(options.ID,'on');
colormap(gray);
list_rob = current(CM,options);
iterations = options.sim_iterations;
iter_count = 1;
```

```
connected = zeros(iterations,2);
if (options.runs > 1) && (do_plot == 0) && (strcmpi(options.ID,'off')) % multiple runs only
    if no plotting
    run_limit = options.runs;
else
    run_limit = 1;
end
run_fitness = zeros(1,run_limit);

%% Main loop
for i=1:run_limit
    while iter_count < iterations
        CM_backup = CM;
        while size(list_rob,1)~=0 % D1 loop
            if secuential_activation
                rindx = 1;
            else
                rindx = ceil(size(list_rob,1)*rand());
            end

            pos = [list_rob(rindx,1) list_rob(rindx,2)];
            CM = apply_rules(CM, pos, individual);

            if do_plot
                CM_aux = CM;
                CM_aux(CM_aux >= 3) = 3;
                imagesc(CM_aux); drawnow;
            end

            list_rob(rindx,:) = [];
        end

        list_rob = current(CM,options);
        connected(iter_count,:) = check_connected(CM,options);
        iter_count = iter_count + 1;

        if CM == CM_backup % Break if no movement
            run_fitness(i) = fitness_fcn(list_rob,connected,iter_count,options);
            break
        end
    end
    %% Fitness Function
    run_fitness(i) = fitness_fcn(list_rob,connected,iter_count,options);
end

individual_fitness = mean(run_fitness);


function CM = apply_rules(CM, pos, individual)
```

```
i = pos(1); j = pos(2);

sigmoid = -5; % -5 to a binary behavior and -1 to a smoother behavior

%% Sub-image
ratio = 1;
SM_in = CM(i-ratio:i+ratio, j-ratio:j+ratio);

number_of_nodes = size(individual.nodegenes,2);
number_of_connections = size(individual.connectiongenes,2);
no_change_treshold = 1e-3; % treshold to judge if state of a node has changed significantly
    since last iteration


%% Asign values

%%%%%%%%%%%%%%%%Input values %%%%%%%%%%%%%%%%%%%%

individual.nodegenes(3,9) = 1; % bias node (9) input state set to 1

SM_in_backup = SM_in;
SM_in(SM_in >= 3) = 3;

% Set input nodes
individual.nodegenes(3,1) = SM_in(1,1);
individual.nodegenes(3,2) = SM_in(1,2);
individual.nodegenes(3,3) = SM_in(1,3);
individual.nodegenes(3,4) = SM_in(2,1);
individual.nodegenes(3,5) = SM_in(2,3);
individual.nodegenes(3,6) = SM_in(3,1);
individual.nodegenes(3,7) = SM_in(3,2);
individual.nodegenes(3,8) = SM_in(3,3);

individual.nodegenes(3,10:number_of_nodes) = 0; % set all node inputs states to zero

%%%%%%%%%%%%%%%Output values %%%%%%%%%%%%

individual.nodegenes(4,1:8) = individual.nodegenes(3,1:8)/3; % normalize inputs of the first
     layer
individual.nodegenes(4,9) = individual.nodegenes(4,9); % keep bias as its input (1)
individual.nodegenes(4,10:number_of_nodes) = 1./(1+exp(sigmoid*individual.nodegenes(3,10:
    number_of_nodes)));

%%%%%%%%%%%%%%%%Evaluate %%%%%%%%%%%%%%%%%%%%

no_change_count = 0;
index_loop = 0;

while (no_change_count < number_of_nodes) && (index_loop < 3*number_of_connections)
```

```
    index_loop = index_loop + 1;
    vector_node_state = individual.nodegenes(4,:);

    for index_connections=1:number_of_connections
        % Read relevant contents of connections gene (ID of node where
        % connection starts, ID node where connection ends and connection
        % weight
        ID_connection_from_node = individual.connectiongenes(2,index_connections);
        ID_connection_to_node = individual.connectiongenes(3,index_connections);
        connection_weight = individual.connectiongenes(4,index_connections);

        % Map node ID's (as extracted from single connection genes above)
        % to index of corresponding node in node genes matrix
        index_connection_from_node = find(individual.nodegenes(1,:) ==
            ID_connection_from_node);
        index_connection_to_node = find(individual.nodegenes(1,:) == ID_connection_to_node);

        if individual.connectiongenes(5,index_connections) == 1 % Check if connection
            enabled
            individual.nodegenes(3,index_connection_to_node) = individual.nodegenes(3,
                index_connection_to_node) + ...
                individual.nodegenes(4,index_connection_from_node)*connection_weight;
        end
    end

    % Pass on node input states to outputs for next time step
    individual.nodegenes(4,10:number_of_nodes) = 1./(1+exp(sigmoid*individual.nodegenes
        (3,10:number_of_nodes)));

    % Re-initialize node input states for next time step
    individual.nodegenes(3,10:number_of_nodes) = 0; % set all output and hidden node input
        states to zero
    no_change_count = sum(abs(individual.nodegenes(4,:) - vector_node_state) <
        no_change_treshold);
end

c1 = round(individual.nodegenes(4,10));
c2 = round(individual.nodegenes(4,11));
c3 = round(individual.nodegenes(4,12));
c4 = round(individual.nodegenes(4,13));

% -1,0,1  displacement in the horizontal(vertical) direction
vertical_variation = c2-c1;
horizontal_variation = c4-c3;

SM_out = swap(SM_in_backup,vertical_variation,horizontal_variation);

CM(i-ratio:i+ratio, j-ratio:j+ratio) = SM_out;
```

```
function CM = build_scenario(options,file)

% CM = BUILD_SCENARIO(OPTIONS, FILE)
%
% Creates an scenario with the dimensions specified in OPTIONS from
% an image contained in FILE. If no file is given, it creates an
% empty scenario.
%
% Types of cell
% 0 reserved for non evaluated cells
% 1 == background
% 2 == obstacle
% 3 == robot

num_arg = nargin;

M = options.scenario_height;
N = options.scenario_width;
rw = options.robot_width;
rh = options.robot_height;
fd = options.floor_depth;


%% Creates obstacles
switch num_arg
    case 1 % Build default empty scenario
        CM = ones(M,N);

    case 2 % Create scenario from image file
        CM = imread(file);
        CM = imresize(CM,[M N]);
        CM = rgb2gray(CM);
        CM = histeq(CM);
        CM = im2bw(CM) + 1;
end


%% Creates Frame
CM(:,1:fd) = 2;
CM(:,N+1-fd:N) = 2;

CM(1:fd,:) = 2;
CM(M+1-fd:M,:) = 2;


%% Creates square robot
if strcmpi(options.ID,'on') % secuential activation enabled
    list = 1:rh*rw;
    list = list + 2;
```

```
    CM(M-rh-fd+1:M-fd,2*fd+1:2*fd+rw) = reshape(list,rh,rw);
else % annonymous modules enabled
    CM(M-rh-fd+1:M-fd,2*fd+1:2*fd+rw) = 3;
end


function y = check_connected(cm, options)

% Returns the size of the biggest robot connected group

% standard representation (homogeneous)
if strcmpi(options.ID,'on')
    cm(cm > 3) = 3;
end

% only robot cells visible
cm(cm ~= 3) = 0;
cc = bwconncomp(cm,4);
numPix_per_group = cellfun(@numel,cc.PixelIdxList);
y = max(numPix_per_group);


function list_out = current(CM,options)

if strcmpi(options.ID,'off')
    [row, col] = find(CM == 3);
    list_out = [row col];
else
    robot_number = options.robot_width*options.robot_height;
    list_out = zeros(robot_number,2);
    for i=3:robot_number+2 % generates list in ID order for secuential activation
        [row col] = find(CM == i);
        list_out(i-2,:) = [row col];
    end
end


function y = fitness_fcn(list_rob, connected_hist, count, opt)

% w = opt.robot_width;
% h = opt.robot_height;

w = opt.robot_width;
h = opt.robot_height;

N = opt.scenario_width;

% Displacement
dist = mean(list_rob(:,2));
%y = mean(list_rob(:,2))/N;
```

```
% Biggest group history
connected = mean(connected_hist(1:count));

y = (dist/N) * (connected/(w*h));
% connected == pctje of the total number
% of robot that remained connected throught simulation


function fits = fits(SM_in, rule_in, to_analize)

% States
% 0: not evaluated
% 1: background
% 2: obstacle
% 3: robot

rule_len = size(rule_in,2);

count = 0;

for i=1:rule_len
    for j=1:rule_len
        if rule_in(i,j)~=0
            switch rule_in(i,j)

                case 1 % Evaluated BackGround
                    if SM_in(i,j) == 1
                    count = count + 1;
                    end

                case 2 % Evaluated Obstacle
                    if SM_in(i,j) == 2
                        count = count + 1;
                    end

                case 3 % Evaluated Robot
                    if SM_in(i,j) == 3
                        count = count + 1;
                    end
            end
        end
    end
end

fits = (count == to_analize);


function SM_out = swap(SM_in, vertical_var, horizontal_var)
```

```
%  output:(
% 8 posible actions in the Moore neighborhood
% [0 1 2
%  3   4
%  5 6 7]

if SM_in(2+vertical_var,2+horizontal_var) == 1
    SM_in(2+vertical_var,2+horizontal_var) = SM_in(2,2);
    SM_in(2,2) = 1;
end

SM_out = SM_in;
```