TRABAJO DE FIN DE GRADO

Grado en Matemáticas

Título: Simulación de algoritmos distribuidos para sistemas robóticos modulares en retículos hexagonales

Autor: Sergio Ordóñez Pérez

Directora: Vera Sacristán Adinolfi

Departamento: Matemática Aplicada II

Convocatoria: 2012-2013

Facultat de Matemàtiques i Estadística

UNIVERSITAT POLITÈCNICA DE CATALUNYA

Universitat Politècnica de Catalunya
Facultat de Matemàtiques i Estadística

Trabajo de fin de grado en Matemáticas

Simulación de algoritmos distribuidos para sistemas robóticos modulares en retículos hexagonales

Sergio Ordóñez Pérez

Directora: Vera Sacristán Adinolfi

Departament de Matemàtica Aplicada II

Resumen

Desde hace aproximadamente 15 años el estudio de los sistemas robóticos modulares distribuidos ha aumentado considerablemente. No es de extrañar, puesto que esto está intrínsecamente relacionado con la resolución de problemas de máxima actualidad como es el caso del problema de reconfiguración. Este problema consiste en dado un objeto formado por pequeños robots con una determinada forma, qué pasos deben ejecutar los robots para conseguir que el objeto tome otra forma totalmente diferente. El estudio de dicho algoritmo es el objetivo principal de este trabajo.

Algoritmos de este tipo han sido ampliamente estudiados suponiendo que los elementos que forman el robot son cuadrados para el caso de un sistema de dos dimensiones, o cúbicos para el caso de sistemas tridimensionales. Pero en este trabajo queríamos estudiar qué pasaría si nos hallásemos ante una teselación regular del plano un poco más compleja como lo es la teselación hexagonal. Por eso, este trabajo se centra en estudiar el algoritmo de reconfiguración sobre un retículo hexagonal en lugar de hacerlo sobre uno cuadrado.

Para ayudarnos en el estudio, en primer lugar se ha implementado un simulador que pueda mostrarnos los resultados del algoritmo, y así ha nacido el simulador llamado *HexaAgentSystem*. No se trata de una aplicación programada desde cero, sino de la adaptación de un simulador anterior llamado *AgentSystem* que era capaz de simular algoritmos distribuidos para sistemas robóticos modulares en retículos cuadrados y cúbicos. Así pues, la primera parte del trabajo ha consistido en adaptar este simulador al caso de retículos hexagonales. La segunda parte del trabajo se ha centrado en el algoritmo de reconfiguración en este retículo.

Como suele ocurrir en estos casos, la adaptación al caso hexagonal del algoritmo preexistente para el caso cuadrado no ha sido inmediata. El hecho de que cada hexágono del retículo tenga seis vecinos en vez de cuatro y que dos celdas hexagonales no puedan ser incidentes sólo en un vértice, como sí ocurre en el caso cuadrado, acarrea consecuencias imprevistas en cuanto a la adaptación del algoritmo. Así pues, en este trabajo se ha llevado a cabo un estudio sistemático de las diferencias y similitudes entre ambos casos antes de proceder a la implementación del algoritmo adaptado adecuadamente.

Los resultados del trabajo han sido satisfactorios, puesto que hemos conseguido implementar una parte importante del algoritmo de reconfiguración en retículos hexagonales con la intención de que este estudio sea de utilidad para investigadores en el futuro.

Abstract

For about 15 years the study of distributed modular robotic systems has increased significantly. This is not surprising since this is related to the resolution of highly topical issues such as the reconfiguration problem. This problem consists on given an object formed by small robots with a certain form, what steps are required for the robots to achieve a totally different form. The study of this algorithm is the primary objective of this thesis.

This algorithm has been widely studied assuming that the elements forming the robot are square in the case of a two-dimensional system, or cubic in three-dimensional systems. But in this thesis we wanted to study what would happen if we consider a regular tessellation of the plane a bit more complex as the hexagonal tessellation is. So this thesis focus on studying the reconfiguration algorithm on a hexagonal lattice instead of on a square one.

To help in this study, we have first implemented a simulator that can show the results of the algorithm, and thus was born the simulator called *HexaAgentSystem*. It is not at absolutely new application, but an adaptaton of an already existing simulator called *AgentSystem* that is able to simulate distributed algorithms for modular robotic systems in square and cubic lattices. So the first part of the thesis has been adapting the simulator to the case of hexagonal lattices. The second part of this thesis is focused on studying the reconfiguration algorithm in this lattice.

As often happens in these cases, the adaptation of the existing algorithm from the square case to the hexagonal case has not been immediate. The fact that each hexagon has six neighbours instead of four and that two hexagonal cells can not be incident only at one vertex, as happens in the square lattice, entails unintended consequences regarding the adaptation of the algorithm. For these reasons in this thesis we have done a systematic study of the differences and similarities between the two cases before proceeding to the implementation of the properly adapted algorithm.

The study results have been satisfactory, since we have managed to implement a big part of the reconfiguration algorithm in hexagonal lattices with the hope that this study will be useful for future research.

Índice general

Capítulo 1. Introducción 1
1. Objetivo 1
2. El simulador 1
3. Algoritmo de reconfiguración 2

Capítulo 2. Descripción del modelo 3
1. Retículo hexagonal 3
2. Clasificación de los vecinos 3
3. Sistema de coordenadas 4
4. Universo 5
5. Módulos 6
6. Movimientos 6
7. Cambios de agarre 7
8. Reglas de actuación 8

Capítulo 3. Algoritmo de reconfiguración 11
1. Objetivo 11
2. Preparación de los módulos 12
3. Reconfiguración a la forma canónica 13
4. Reconfiguración a la forma final 14

Capítulo 4. Reconfiguración a la forma canónica 15
1. Objetivo 15
2. Fases del algoritmo 16
3. Activación 16
4. Movimiento: Rotaciones 20
5. Movimiento: Cambiar de agarre, siguiente 27
6. Movimiento: Cambiar de agarre, intermedio 29
7. Movimiento: Cambiar de agarre, opuesto 41
8. Formación de la tira 49

Capítulo 5. Simulador hexagonal 53
1. Implementación 53
2. Interficie 56
3. Acerca del simulador 59

Capítulo 6. Conclusiones 61

Capítulo 7. Compact User Guide of the hexagonal Agent System 63
1. Introduction 63
2. Program flow 63
3. Program inputs 64
4. Program window 75
5. Limitations 83
6. Program outputs 84
7. Requirements 86
8. Installation 86

Referencias 87

Capítulo 1
Introducción

1. Objetivo

Este proyecto tiene dos objetivos principales. Por un lado, el de crear un simulador capaz de reproducir la actuación distribuida de un sistema de robots modulares localizados en un retículo o escenario hexagonal. Mientras que esta parte del proyecto es más programativa que de investigación, la segunda mitad es todo lo contrario. El segundo objetivo consiste en adaptar un algoritmo de reconfiguración a este escenario y usar el simulador para implementarlo.

2. El simulador

Para entender mejor la primera parte del trabajo se debe profundizar en lo que quiere decir cada una de las expresiones utilizadas. Cuando hablamos de sistemas robóticos modulares nos referimos a un conjunto de módulos o entidades conviviendo en un espacio o universo e independientes unas de otras, pero capaces de interactuar con sus vecinos más próximos [**1**][**2**]. Al hablar de algoritmos distribuidos nos referimos a un conjunto de reglas o instrucciones que son ejecutadas independientemente por cada elemento de entre los de un conjunto, es decir, que cada elemento intentará ejecutar cada una de las reglas independientemente de lo que estén haciendo el resto, aunque atendiendo a la situación de su entorno más inmediato. Así pues, el simulador de algoritmos distribuidos para sistemas robóticos modulares antes mencionado no es más que una herramienta capaz de definir por un lado un conjunto de módulos y su estado, y por otro el conjunto de reglas que estos deberán aplicar de forma totalmente distribuida.

El simulador implementado no solo es capaz de definir el conjunto de módulos y de reglas, sino también de ejecutar las reglas definidas sobre el conjunto de módulos de forma distribuida y de mostrar el resultado obtenido al aplicarlas. Además, el simulador consta de una parte gráfica que muestra el proceso de aplicación de las reglas al conjunto de módulos para nuestro caso concreto aplicado a un universo de retículo hexagonal.

Obsérvese que ya existían simuladores similares de algoritmos distribuidos para sistemas robóticos modulares pero que trabajaban sobre otro tipo de retículos, como pueden ser el retículo cuadrado en un espacio de dos dimensiones, o el retículo cúbico en un espacio tridimensional [**3**][**4**]. A pesar de que existen algunos simuladores muy genéricos de sistemas robóticos modulares que se pueden utilizar para simular el caso de un retículo hexagonal, nos ha interesado hacer un simulador mucho más específico para estudiar este problema concreto.

No tiene sentido intentar expandir este tipo de retículos al espacio tridimensional a diferencia de en el caso cuadrado en el que sí que tenía sentido, así pues, durante el resto del trabajo se trabajará exclusivamente sobre el retículo hexagonal del plano.

3. Algoritmo de reconfiguración

La segunda parte del trabajo supone un ejercicio de investigación e implementación del conjunto de reglas que deben ser aplicadas al simulador para ser capaces de reconfigurar un conjunto de módulos cualquiera a otro totalmente diferente. Esta parte de investigación e implementación se realiza a partir de un precedente ya existente que reproduce el algoritmo de reconfiguración sobre un retículo cuadrado [**5**][**6**]. El objetivo del algoritmo es a partir de una configuración inicial de módulos conectados de una determinada manera, y mediante movimientos sencillos de estos, obtener otra configuración distinta, del mismo número de módulos pero con otra forma diferente. Para esto hay que entender plenamente las capacidades de los módulos de los que disponemos y los problemas que nos podemos encontrar al intentar realizar esta tarea en nuestro retículo hexagonal. De todo ello hablaremos más adelante.

Por último, es necesario añadir que el código del simulador está totalmente abierto a cualquier persona que quiera estudiarlo o incluso ampliarlo, y que tanto este como el conjunto de reglas obtenidas para la aplicación del algoritmo de reconfiguración en retículos hexagonales pueden encontrarse en [**7**].

Capítulo 2
Descripción del modelo

En la descripción del módelo que se hace a continuación se establece un primer contacto con el universo de retículo hexagonal en el que se sitúan los módulos, y se habla de las capacidades y limitaciones que estos tienen.

1. Retículo hexagonal

Como ya se ha comentado, en este proyecto se va a trabajar sobre un retículo hexagonal del plano. Esto quiere decir que se va a dividir el plano en hexagonos, cada uno identificado por dos coordenadas enteras.

En la división del plano mediante un retícul hexagonal, la primera peculiaridad que llama la atención es que cada posición dispone exactamente de 6 vecinos, entendiendo por vecinos a todas las posiciones adyacentes a ella en el plano. Si se compara este retículo con otros retículos del plano, como por ejemplo el triangular (en el que cada posición consta de 3 vecinos) o el cuadrado (4 vecinos) rápidamente se comprueba que el retículo hexagonal es la teselación regular del plano que dispone de un mayor número de adyacencias. Obsérvese que las únicas tres teselaciones regulares del plano son las ya mencionadas (triángulos equiláteros, cuadrados y hexágonos equiláteros) y que de las tres, la figura con más lados es el hexágono.

Es necesario clasificar correctamente cada vecino y disponer de un sistema de coordenadas capaz de identificar unívocamente cada posición del plano. Dado que la teselación hexagonal es la menos simple de las 3 teselaciones regulares del plano, cabe esperar que sea más complicado trabajar con ella de lo que lo sería la teselación cuadrada a la que estamos más acostumbrados.

2. Clasificación de los vecinos

El primer paso que se debe dar para definir el retículo hexagonal sobre el que queremos trabajar es el de diferenciar entre los distintos vecinos que tiene una posición cualquiera del plano.

En nuestro caso, se utilizan los puntos cardinales para identificar los distintos vecinos. Aunque los puntos cardinales son 4 y el número de vecinos en nuestro retículo es 6 se han utilizado los puntos cardinales intermedios para diferenciar unívocamente todas las posiciones adyacentes.

La clasificación de los vecinos en el retículo hexagonal queda determinada de la manera que se ilustra en la Figura 1.

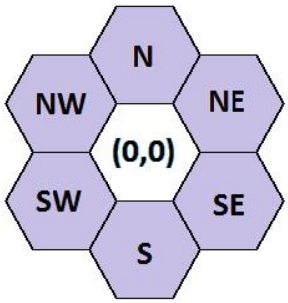

FIGURA 1. Clasificación de los vecinos en un retículo hexagonal.

3. Sistema de coordenadas

El segundo punto a tener en cuenta a la hora de definir nuestro retículo hexagonal es el sistema de coordenadas que se va a utilizar. Son varios los sistemas de coordenadas válidos que se pueden usar, y no hay uno que sea más popular que el resto en el uso de este tipo de retículos. La Figura 2 muestra cuál es el sistema que se utiliza en este proyecto.

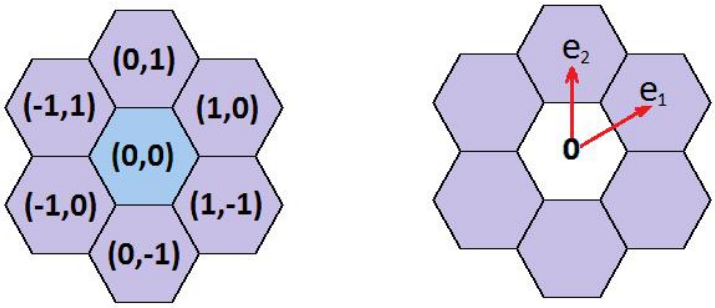

FIGURA 2. Sistema de coordenadas en un retículo hexagonal.

Se ha decidido utilizar este sistema de coordenadas en el que el eje X va en la dirección noreste y el eje Y va en la dirección norte por varias razones, pero la principal es que de esta manera todas las coordenadas están compuestas por números enteros, y cualquier pareja de números enteros se corresponde con una posición en el retículo.

Existen otros sistemas de coordenadas que tienen otras ventajas, como por ejemplo ejes ortogonales, pero en los que o bien hay que definir ciertas corrdenadas con fracciones, o bien disponen de huecos correspondientes a coordenadas que no se corresponden con ningún hexágono del retículo.

4. Universo

Cuando hablamos de universo, nos referimos a las posiciones del retículo que los módulos pueden ocupar durante la ejecución del algoritmo. Así pues, el universo impone unas limitaciones en las posiciones que los módulos pueden ocupar.

Definimos el universo mediante 4 números enteros que se corresponden con la menor y la mayor posicióndel eje X que un módulo puede ocupar, y lo mismo con el eje Y. Mientras que este tipo de definición determinaría un rectángulo en el caso de situarnos sobre un retículo cuadrado, en el caso de un retículo hexagonal el resultado es algo diferente.

La Figura 3 muestra cómo es el universo dado un retículo hexagonal si las coordenadas en el eje X varian entre x_1 y x_2 y al mismo tiempo las coordenadas del eje Y van desde y_1 hasta y_2.

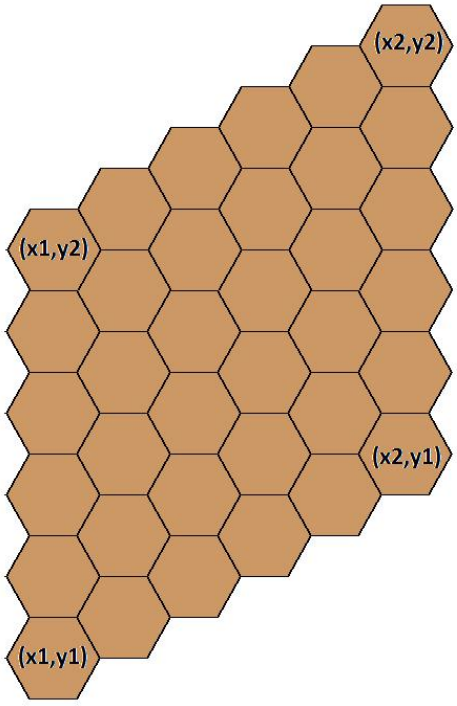

FIGURA 3. Universo en nuestro retículo hexagonal.

Se observa claramente como el universo no tiene forma rectangular, sino en todo caso, romboidal.

5. Módulos

Los módulos que vamos a utilizar deben ser vistos como pequeños robots capaces de realizar una serie de tareas básicas. Estas tareas las ejecutaran a partir de reglas definidas en nuestro simulador cuando satisfagan las condiciones. Debemos tener en cuenta que todos los módulos son iguales, y que por lo tanto tienen las mismas capacidades y aplican las mismas reglas. Los módulos con los que trabajaremos tienen una capacidad de procesamiento simple y una memoria muy limitada (de tamaño constante).

Cada módulo dispone de ciertas capacidades básicas como son: detectar si está o no conectado a un vecino, enviar y recibir mensajes simples de sus vecinos, realizar operaciones elementales con números y contadores, y cambiar de estado o de posición. Las operaciones elementales que puede realizar un módulo son las de comparar y combinar mediante operaciones aritméticas datos de contadores, mensajes y otros números enteros.

Además, cada módulo tiene información propia que puede almacenar y modificar a través de las reglas. La información de la que dispone es: las conexiones existentes con sus vecinos, el valor de sus contadores en los que se almacenan valores enteros, y los mensajes de entrada y de salida.

Detalles más precisos sobre las características de los módulos se encuentran en el manual de usuario.

6. Movimientos

Para hablar de los movimientos que tienen permitidos los módulos sobre el universo de retículo hexagonal debemos pensar en ellos como objetos físicos que se encuentran sobre un papel que ha sido teselado en hexágonos y que representa el plano. Si tenemos claras estas suposiciones, entenderemos mejor las limitaciones básicas del movimiento de nuestros módulos.

La primera limitación es que dos módulos no pueden estar simultáneamente sobre una misma posición del retículo. Además, como el espacio es plano, un módulo no puede pasar por encima de otro o atravesarlo. Esta limitación se deduce a partir de la suposición de que el módulo es un ente físico que ocupa un espacio determinado.

La segunda limitación tiene que ver con la fuerza necesaria para mover un módulo y consiste en que un módulo no puede moverse por sí solo, sino que necesita la ayuda de otro que le esté agarrando. Esto se debe a que un módulo no puede accionar su propio movimiento, pero sí puede, estando él fijo en una posición, empujar a otro módulo conectado a él en una dirección determinada. En otras palabras, el movimiento de un módulo siempre es relativo a otro.

Así pues, las dos acciones que un módulo puede realizar para dotar al sistema de movimiento son agarrarse o soltarse de un módulo adyacente, o hacer que un módulo agarrado a otro realice una rotación sobre el que está fijo. Este segundo tipo de desplazamiento consiste en dejar que el módulo en movimiento gire entorno

al módulo fijo desde una de sus posiciones adyacentes a la siguiente, tal como se muestra en la Figura 4.

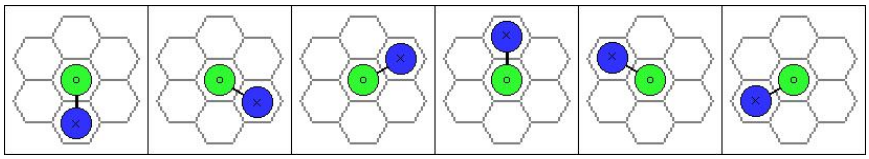

FIGURA 4. Ejemplo de movimiento de rotación.

Así pues para dotar al sistema de movimiento hacen falta un mínimo de dos módulos que se vayan turnando para empujarse el uno al otro, pero contra más módulos haya sobre el retículo, más complejos serán los movimiento que se pueden realizar. Por ejemplo, con más de dos módulos podemos programar un movimiento que consista en empujar a un módulo, que después se suelte de su agarre para agarrase a uno nuevo, que a su vez lo empuje hasta situarlo en una posición de adyacencia respecto a otro, y así sucesivamente. Asimismo, se podrán producir diversos movimiento en paralelo, debido a que como ya hemos dicho, se trata de un sistema distribuido.

7. Cambios de agarre

Ya hemos hablado de la capacidad que tiene un módulo para agarrase o soltarse de otro que sea adyacente a él. Precisamente esta capacidad dota a los módulos con la oportunidad de realizar un movimiento que llamamos *cambio de agarre*. Este tipo de movimiento no lleva asociado un cambio en la posición del módulo, sino que consiste en soltarse de un módulo adyacente a él al que estaba previamente agarrado, para posteriormente agarrarse a otro de sus módulos adyacentes al que no estaba agarrado en un principio.

Al soltarse de un módulo para agarrase a otro, diferenciamos entre distintos casos dependiendo de cuál es la posición del nuevo módulo al que se agarra, relativa al módulo al que estaba agarrado en un principio. Nos hallaremos en un caso o en otro dependiendo de cuántas posiciones adyacentes nos hemos saltado desde el módulo incial hasta el módulo final contadas antihorariamente.

Si al soltarse el módulo de un vecino, este se agarra al vecino siguiente considerando el sentido antihorario, estaremos hablando del movimiento *cambio de agarre, siguiente*. En la Figura 5 se observa este tipo de movimiento en el que un módulo se suelta de su agarre actual (negro) y se agarra a su vecino siguiente (rojo).

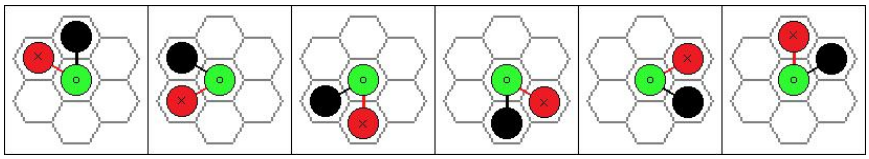

FIGURA 5. Movimiento de cambio de agarre, siguiente.

Si cuando se suelta el módulo de un vecino, este se agarra al segundo vecino posterior considerando el sentido antihorario, estaremos hablando del movimiento *cambio de agarre, intermedio*. En la Figura 6 se observa este tipo de movimiento en el que un módulo se suelta de su agarre actual (negro) y se agarra a su vecino intermedio (rojo).

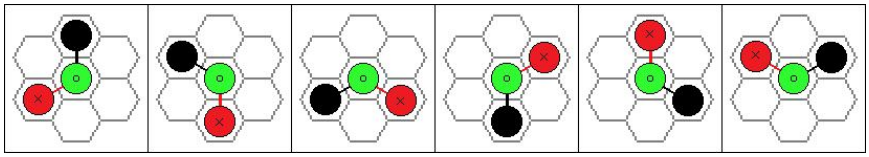

FIGURA 6. Movimiento de cambio de agarre, intermedio.

Por último, si al soltarse el módulo de un vecino, este se agarra al vecino opuesto, que coincide con el tercer vecino contado en sentido antihorario desde el inicial, estaremos hablando del movimiento *cambio de agarre, opuesto*. En la Figura 7 observamos este tipo de movimiento en el que un módulo se suelta de su agarre actual (negro) y se agarra a su vecino opuesto (rojo).

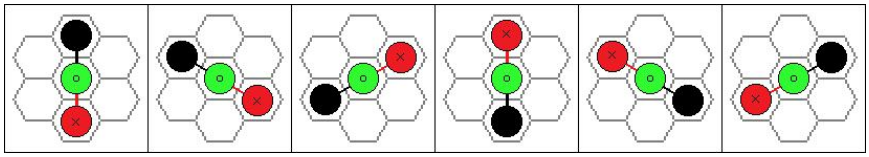

FIGURA 7. Movimiento de cambio de agarre, opuesto.

No consideramos más tipos de cambios de agarre a pesar de que existen todavía dos posiciones adyacentes no exploradas. Esto se debe a que el resto de posibles cambios de agarre no nos resultarán de utilidad en el problema de reconfiguración que queremos resolver y por ello no los mencionaremos a lo largo del trabajo. El motivo por el que no utilizamos el resto de posibles cambios de agarre para el algoritmo de reconfiguración es que este tipo de movimientos supondrían un recorrido por la frontera siguiendo la regla de la mano izquierda, y esto es algo que queremos evitar.

8. Reglas de actuación

Ya hemos mencionado que los módulos ejecutan de forma totalmente distribuida una o más reglas en cada iteración de entre las definidas en un conjunto. Para entender mejor cómo y cuándo se ejecutan estas reglas, debemos definirlas apropiadamente.

Las reglas que se utilizan en nuestro simulador constan de un nombre identificador de la regla, de un número que se corresponde con su prioridad, de una precondición, y de una acción.

El nombre de la regla es una cadena de texto no vacía y única, es decir, no puede haber dos reglas con el mismo nombre. La prioridad es un número entero positivo que decide en caso de que varias reglas puedan ser ejecutadas, cuál de ellas tiene prioridad sobre el resto, donde un mayor valor de la prioridad gana sobre los valores menores. La precondición de una regla determina si esta puede ser aplicada o no. Por último, la acción define qué ocurre cuando la regla se ejecuta.

Para más información acerca de las reglas y de cada una de sus partes debe consultarse el manual de usuario del simulador.

Capítulo 3
Algoritmo de reconfiguración

1. Objetivo

El objetivo de esta parte del trabajo es implementar un algoritmo que sea capaz de simular cómo debe comportarse el conjunto de los módulos para pasar de una forma a otra. Para ello estudiamos cómo conseguir que una cantidad indeterminada de módulos ya conectados puedan interactuar entre ellos de tal manera que modifiquen su forma hasta tranformarse primero en una forma canónica, y finalmente en la forma deseada.

Nuestra solución pasa por construir un árbol generador del conjunto de módulos, garantizar que las hojas del árbol se encuentren en la frontera de la configuración, e imponer que sean los únicos módulos capaces de moverse. El movimiento de los módulos se realizará a través de la frontera del conjunto, y siguiendo primero la regla de la mano derecha durante su conversión en la forma canónica, y posteriormente la regla de la mano izquierda durante su tranformación hasta conseguir la forma deseada.

Como ya hemos explicado en el capítulo anterior, no se permite que haya dos módulos sobre una misma posición del universo, ni tampoco que un módulo se mueva a través de otro.

Cabe destacar que el algoritmo implementado se trata de un algoritmo distribuido. Esto significa que cada módulo actúa por su cuenta, sin necesidad de que haya un controlador central que les diga a cada uno lo que deben hacer.

Hemos dividido el algoritmo de reconfiguración en tres partes, que son:

- **Preparación de los módulos**: Los módulos se comunican entre sí para tomar una serie de decisiones, como la de nombrar entre todos ellos a un módulo que haga de líder, detectar los posibles agujeros que tiene la forma inicial, decidir el árbol generador, y construir una función potencial que guiará la reconfiguración. En esta parte del algoritmo no se ejecuta ningún movimiento, solo se realizan los cálculos y se envian los mensajes necesarios parar preparar el proceso de reconfiguración.

- **Reconfiguración a la forma canónica**: En esta fase los módulos comienzan a moverse. Las hojas del árbol se desplazan siguiendo la regla de la mano derecha, y al moverse, nuevos módulos convertidos en hojas lo hacen también para finalmente convertir el conjunto en una forma canónica, que no es más que una tira rectilínea formada en dirección noreste a partir del módulo que hace de líder.

- **Reconfiguración a la forma final**: Por último, esta fase es la encargada de convertir la forma canónica en la forma final deseada. Los módulos vuelven a moverse utilizando esta vez la regla de la mano izquierda hasta llegar al lugar en el que deben permanecer para pasar a formar parte de la configuración final.

2. Preparación de los módulos

Esta parte del algoritmo consiste sobretodo en un proceso de intercambio de mensajes entre los distintos módulos para decidir quién de ellos será el líder, identificar los agujeros de la configuración, construir un árbol que conecte todos los módulos sin que haya ciclos entre las conexiones y las hojas estén ubicadas en la frontera, y asignarle a cada módulo el valor de su función potencial.

Cabe decir que esta parte del algoritmo no pertenece a este proyecto, sino que ya había sido implementada con antrioridad por Angel Rodríguez, que consiguió adaptar las reglas del algoritmo implementado para el caso cuadrado por Óscar Rodríguez.

A continuación se explican brebemente cada uno de los pasos que realiza esta parte del algoritmo, sin entrar en demasiados detalles debido a que no es objetivo de este trabajo el profundizar en las reglas de esta parte del algoritmo:

(1) **Elegir un líder**: El líder es el módulo localizado en la parte más alta, y más a la derecha de entre todos los módulos. Por lo tanto por fuerza se encuentra en la frontera externa del robot. En un principio todos los módulos que no tengan ningún vecino ni al norte, ni al noreste, ni al noroeste son candidatos a ser el líder, y envian un mensaje que circula a lo largo de toda la frontera para comprobar si existe algún otro módulo que se encuentra más arriba o más a la derecha. Dicho mensaje consta de unas coordenadas enteras relativas y de un booleano. A cada paso, ambos se actualizan adecuadamente. Cuando el mensaje vuelve al emisor, este es capaz de detectarlo por las coordenadas del mismo. Solo uno de los módulos detecta que él es el módulo que está más arriba más a la derecha, y que por tanto es el líder.

(2) **Detección de agujeros**: Se trata de detectar si hay agujeros dentro de la configuración y en caso de haberlos, decidir quién será el módulo respondable de cada agujero. El módulo responsable de un agujero se encarga de abrir el agujero desconectando las conexiones necesarias. Como en el caso del líder, para decidir quien es el responsable de un agujero, se detectan los posibles candidatos y se hace un barrido mediante mensajes a través de la frontera hasta que solo queda uno.

(3) **Construcción del árbol**: Se trata de transformar la configuración del robot en un grafo conexo y acíclico, simulando una desconexión allá donde haga falta hasta que el árbol lógico coincida con las conexiones intermodulares correspondientes a la forma de árbol. Inicialmente todos los módulos están conectados por lo que el algoritmo debe decidir qué módulos deben simular su desconexión, y esto lo hace de manera que en el grafo resultante todos los módulos estén conectados verticalmente y que las columnas formadas sólo se conecten entre sí por la parte más alta posible (excepto en el caso de los responsables de agujero, que deben forzar una desconexión para no formar un ciclo).

(4) **Construcción de la función potencial**: Se trata de asignar a cada módulo un valor de la función potencial. Este valor indica la distancia del módulo hasta el líder. La construcción de esta función se hace extendiendo su valor, empezando desde el líder y siguiendo las conexiones del árbol mediante un recorrido en profundidad con prioridad antihoraria. Esta propagación transporta además los datos de los valores mínimo y máximo del potencial de la rama en la que se encuentra un módulo, y la dirección en la que se encuentra el padre del módulo. Estos datos resultarán fundamentales para guiar la reconfiguración.

3. Reconfiguración a la forma canónica

Durante esta parte del algoritmo, la forma inicial de los módulos se deshace para convertirse en una forma canónica que, como ya hemos dicho, es una tira rectilínea formada en dirección noreste a partir del módulo que hace de líder.

En esta parte del algoritmo se centra la segunda parte de este trabajo. Despues de implementar el simulador, se ha trabajado en esta parte del algoritmo de reconfiguración, estudiando los casos y escribiendo las reglas, y por ello más adelante explicaremos esta parte en detalle.

A continuación se explican brebemente cada uno de los pasos que realiza esta parte del algoritmo, aunque más adelante entraremos mucho más en detalle:

(1) **Activación**: Se produce cuando el módulo es una hoja, bien por que lo era inicialmente, o porque se ha convertido en una debido al movimiento de otros módulos. Un módulo necesita estar activo para poder iniciar cualquier tipo de movimiento.

(2) **Movimiento**: Cuando un módulo ha sido activado, empieza a moverse a lo largo de la frontera siguiendo la regla de la mano derecha. En todo momento se controla que dos módulos en movimiento no puedan entrar en conflicto al intentar desplazarse ambos a una misma posición, para lo cual se establece una prioridad que decide cuál de los dos puede moverse primero.

(3) **Formación de la tira**: Cuando un módulo en movimiento sobrepasa al módulo que hace de líder, sabe que debe pasar a formar parte de la tira, y que cuando

llegue al final de la tira formada hasta ahora, debe detenerse y avisar al módulo que venga detrás suyo de que la tira acaba en su posición. Esto sucede hasta que el último módulo se acopla a la tira, momento en el que esta parte del algoritmo de reconfiguración se da por terminada.

4. Reconfiguración a la forma final

La última parte del algoritmo de reconfiguración es la que convierte la forma canónica en la forma deseada. Para ello los módulos se mueven, deshaciendo la tira y siguiendo la regla de la mano izquierda hasta ocupar su lugar en la forma final.

Esta parte del algoritmo no ha sido aún implementada. Construir esta última parte del algoritmo sería una buena posible futura ampliación del trabajo, o un trabajo totalmente nuevo.

No podemos enumerar los pasos que realiza esta parte del algoritmo porque como ya hemos dicho, es una parte del problema que todavía no se ha estudiado lo suficiente, y haría falta dedicar más tiempo para poder hablar con propiedad del proceso de reconfiguración a la forma final.

Capítulo 4
Reconfiguración a la forma canónica

1. Objetivo

En este capítulo nos disponemos a describir detalladamente el algoritmo y las reglas utilizadas para convertir la configuración inicial en una tira rectilínea formada a partir del módulo que hace de líder. También describiremos los problemas que nos podemos encontrar durante la realización de este proceso, como por ejemplo los cuellos de botella que puede haber y que si no son tratados correctamente, pueden bloquear el movimiento de los módulos.

Para conseguir que el conjunto de los módulos deshaga su forma original y se transforme en la forma canónica, estos han de desplazarse siguiendo la frontera de la configuración, cosa que hacen aplicando la regla de la mano derecha para el recorrido de laberintos. Esta regla consiste en siempre que podamos, movernos hacia la derecha, y cuando haya varios caminos posibles por los que continuar, escoger siempre el de más a la derecha posible. Esta regla de movimiento asegura que cada módulo podrá salir del laberinto que forma la configuración, y que por lo tanto, al final todos los módulos habrán pasado por el máster y formarán parte de la tira.

Para evitar desconectar el árbol, y por lo tanto, la configuración, solo las hojas del árbol pueden desconectarse, moverse y volverse a conectar al mismo o a un nuevo módulo, y a través de este proceso, conforme estas van avanzando por la frontera, otros módulos van perdiendo conexiones y convirtiéndose ellos mismos en nuevas hojas capaces de moverse.

Cuando una hoja se mueve una posición en dirección horaria respecto al módulo al que está agarrado, decimos que la hoja ha realizado un movimiento de rotación, mientras que cuando una hoja se desconecta para conectarse a un módulo diferente, decimos que la hoja ha efectuado un movimiento de cambio de agarre, aunque en este caso la hoja no llega a moverse en lo que a cambiar de coordenadas se refirere.

Finalmente, cuando las hojas han recorrido toda la frontera, y han pasado por el lugar en el que se encuentra el líder, ya saben que han pasado a formar parte de la tira, y que deben detener su movimiento.

2. Fases del algoritmo

Antés de entrar en detalles específicos sobre las reglas y sus propiedades, es conveniente recordar en qué consisten las tres fases que componen esta parte del algoritmo que se encarga de realizar la reconfiguración a la forma canónica a partir de la forma inicial.

La fase de activación se encarga de seleccionar los módulos que pueden empezar a moverse por la frontera. La fase de movimiento por su parte es la encargada de realizar todos aquellos movimientos permitidos que hagan que los módulos acaben formando parte de la forma canónica. Por último, la fase de formación de la tira se encarga de que los módulos dejen de moverse una vez que estos ya forman parte de la forma canónica.

Antes de continuar, debemos recordar que antes de iniciarse esta parte del algoritmo se ha ejecutado la parte de preparación de los módulos, en la que se ha elegido al líder, se han detectado los agujeros y elegido a sus responsables, y se ha formado el árbol lógico.

A continuación, nos disponemos a explicar cada una de las fases con más detalle, enumerando y descibriendo las reglas que se utilizan en cada fase, así como las consideraciones importantes que se han tenido en cuenta durante la implementación de estas reglas.

3. Activación

Las reglas de activación se ejecutan sobre aquellos módulos que están preparados para moverse, es decir, aquellos de los que no cuelga ningún otro módulo.

Ya sea porque el módulo es una hoja del árbol en el momento de su construcción, o porque todos los módulos que colgaban de él se han desplazado hasta que no quede ningun otro por debajo suyo, un módulo intentará activarse en el momento en el que solo un agarre lo mantenga unido al resto de la configuración. En otras palabras en el momento en el que el módulo se haya convertido en una hoja del árbol dinámico, este estará listo para su activación y su posterior movimiento por la frontera.

3.1. Lista de reglas

Existen un total de 13 reglas en la fase de activación, 12 de las cuales estan divididas en 2 subgrupos de 6 reglas cada uno, mientras que la última regla es totalmente independiente de las otras.

El primer subgrupo de 6 reglas sirve para declarar la intención de activación por parte de los módulos. Como ya hemos dicho antes, la condición para intentar activarse es ser una hoja del árbol. Como se muestra en la Figura 1, cada una de las reglas del subgrupo hace referencia a cada una de las 6 posiciones en las que se puede encontrar una hoja, que se correponden con las 6 direcciones por las que puede estar enganchada a la rama del árbol (N, NW, SW, S, SE NE).

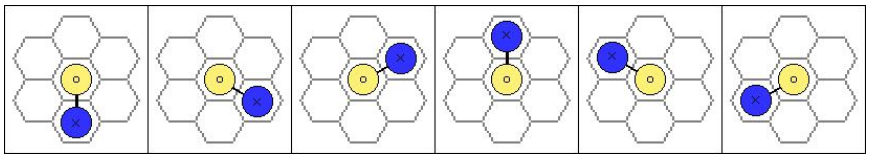

FIGURA 1. Reglas de intención de activación.

El segundo subgrupo de 6 reglas se ejecuta sobre aquellos módulos que hayan expresado su intención de activarse. Tras comprobar si se siguen satisfaciendo las condiciones, el módulo lleva a cabo su activación inmediata realizando un cambio en su estado para identificarlo como módulo activo. Como antes, cada regla del subgrupo se identifica con una de las 6 direcciones por las que puede estar enganchado el módulo (N, NW, SW, S, SE NE), tal como se muestra en la Figura 2.

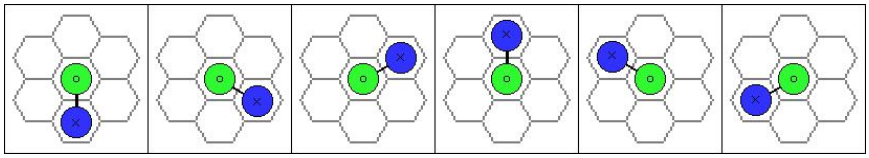

FIGURA 2. Reglas de consecución de activación.

La última regla es la encargada de anular la intención de activación y se muestra en la Figura 3. Cuando un módulo que ha expresado su intención de activarse detecta que ha dejado de satisfacer las condiciones de activación, esta regla se encarga de devolver al módulo a su estado anterior. Esto se realiza mediante prioridades. Cuando el módulo con intención de activarse observa que no satisface las condiciones de ninguna de las reglas de activación, ejecuta esta regla, que es la siguiente más prioritaria.

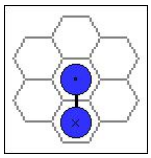

FIGURA 3. Regla de cancelación de activación.

Este proceso en dos fases (declaración de intenciones más activación o cancelación) está diseñado para evitar que se produzcan lo que podríamos llamar conflictos de activación, es decir, para evitar que un módulo activo se desplace sobre un módulo en fase de activación. Esto debe evitarse porque podría dar lugar a módulos activos que no son hojas del árbol y esto a su vez producir movimientos de fuerza lineal,que como ya hemos dicho, no aceptamos en nuestro modelo.

3.2. Descripción de las reglas

Para describir las reglas que se han utilizado en esta fase, explicaremos detalladamente las partes de una sola regla de cada uno de los subgrupos antes mencionados. Puesto que las 6 reglas de cada subgrupo son muy parecidas, no es necesario conocer en detalle cada una de ellas, sino que basta con ver cómo se ha implementado una, y qué haría falta cambiar para obtener el resto.

Empezamos describiendo la regla de intención de activación de una hoja conectada por el sur. El resto de reglas de intención de activación se obtienen mediante modificaciones menores sólo relativas a la condición que comprueba por dónde está conectado el módulo. La regla de intención de activación al sur es la siguiente:

Activate Leaf N Try

8022

Spfc_* !S****M !S****E A000100 T0,-1,pfc_*

SpfcA*

Esta regla tiene más prioridad que las de aceptación o cancelación de activación porque antes de hacer cualquiera de esas dos cosas el módulo debe expresar su intención de activarse.

La precondición de la regla se divide en varias condiciones. Todas ellas deben satisfacerse para que la acción se ejecute:

- **Spfc_***: Afecta a módulos que saben cuál es su posición dentro del árbol.
- **!S****M**: No afecta al líder.
- **!S****E**: No afecta a los módulos que ya forman parte de la tira final.
- **A000100**: Afecta a módulos que sólo están conectados al sur. Debemos cambiar esta condición para obtener el resto de reglas del subgrupo.
- **T0,-1,pfc_***: Afecta a los módulos cuyo vecino del sur también sepa cuál es su posición dentro del árbol. También debemos cambiar las coordenadas de esta condición para formular el resto de reglas del subgrupo.

La acción **SpfcA*** consiste simplemente en cambiar el estado del módulo al nuevo estado de intención de activación. Este cambio de estado será necesario para satisfacer las condiciones del resto de reglas que vienen a continuación.

Las reglas de aceptación de la activación se ejecutan sobre aquellos módulos que han mostrado su intención de activarse. En ellas se realizan prácticamente las mismas comprobaciones que acabamos de ver, pero se lleva a cabo un mayor número de

acciones. A continuación se muestra la regla de aceptación de activación de una hoja conectada por el sur como representante de este subgrupo de 6 reglas:

Activate Leaf N Accept

8021

SpfcA* !S****M !S****E A000100 T0,-1,pfc_*

C000+0,-1,C000 0001 C020+0000 0000 C021+0000 0000 Sfw__A

Esta regla tiene más prioridad que la de cancelación, pero menos que las de intención de activación porque la cancelación sólo puede producirse si no se ha producido la aceptación, y antes de aceptar la activación, un módulo ha debido expresar su intención.

La precondición de la regla se divide de nuevo en varias condiciones que deben satisfacerse para que la acción se ejecute:

- **SpfcA***: Afecta a módulos que han expresado su intención de activación.
- **!S****M**: No afecta al líder.
- **!S****E**: No afecta a los módulos que ya forman parte de la tira final.
- **A000100**: Afecta a módulos que sólo están conectados al sur. Debemos cambiar esta condición para obtener el resto de reglas del subgrupo.
- **T0,-1,pfc_***: Afecta a los módulos cuyo vecino del sur sepa cuál es su posición dentro del árbol. También debemos cambiar las coordenadas de esta condición para formular el resto de reglas del subgrupo.

La acciones a realizar en caso de satisfacer todas las condiciones son las propias de un módulo que empieza a recorrer la frontera de la configuración. A continuación se listan y se describe cada una de ellas:

- **C000 + 0,-1, C000 0001**: Inicializa en el contador C000 el valor de función potencial a partir del valor del módulo sobre el que cuelga añadiéndole una unidad.
- **C020 + 0000 0000**: Inicializa a 0 el contador de movimientos en el eje x que el módulo ha realizado desde el momento de su activación. Esto sirve para saber desde que punto ha viajado un módulo en cualquier momento.
- **C021 + 0000 0000**: Inicializa a 0 el contador de movimientos en el eje y que el módulo ha realizado desde el momento de su activación. Tiene el mismo propósito que la acción anterior.
- **Sfw__A** : Cambia el estado del módulo a estado activo.

Por último, la regla de cancelación de activación solo se ejecuta en caso de que ninguna de las reglas de aceptación haya sido satisfecha. Deducimos que si ninguna de las reglas anteriores se satisface, alguna de las condiciones necesarias para la activación del módulo ha dejado de cumplirse, y por ello el módulo ya no es candidato para ser activado. Más tarde o más temprano el módulo volverá a intentar la activación (cuando vuelva a satisfacer las condiciones) pues el objetivo final del algoritmo es que todos los módulos se activen y caminen por la frontera hasta formar la tira. A continuación descubrimos los detalles de la regla de cancelación:

Activate Leaf Cancel

8020

SpfcA*

Spfc_*

Esta regla tiene menor prioridad que las de intención y que las de aceptación de activación ya que para cancelar la activación, primero ha debido producirse la intención por parte del módulo, y ha debido comprobarse si el módulo satisfacía las condiciones para activarse.

La precondición de esta regla es mucho más simple que la de las anteriores. **SpfcA*** sólo comprueba que el módulo esté en un estado de intención de activación. No es necesario comprobar nada más debido a que la prioridad de esta regla fuerza a que sólo se ejecute si alguna de las condiciones de las reglas de aceptación no se cumple. Si alguna de las reglas de aceptación se satisficiera, esta regla no podría ejecutarse debido a que tiene menor prioridad.

La acción **Spfc_*** simplemente retorna el estado del módulo al que tenía previamente para que esté atento al momento en que pueda intentar activarse de nuevo.

3.3. Juegos de pruebas

Los juegos de prueba utilizados para comprobar que estas reglas funcionan son sencillos debido a que las reglas también lo son. Podremos comprobar que realmente las reglas de activación funcionan correctamente en cualquiera de los juego de prueba que utilizaremos posteriormente para probar el resto de reglas del algoritmo de avance.

4. Movimiento: Rotaciones

Las reglas de rotación son lás más básicas de entre todas las reglas de movimiento. A diferencia del resto de reglas de movimiento (llamadas de cambio de agarre), estas consisten en cambiar de posición un módulo sin soltarse del módulo al que está agarrado, pero cambiando el lado del que se agarra.

El objetivo de este tipo de reglas es permitir que el módulo activo se mueva a traves de la frontera siguiendo la regla de la mano derecha hasta llegar a formar parte de la tira. Por ello solo permitimos rotaciones simples (el menor giro posible) y en el sentido de las agujas del reloj, es decir, sólo consideramos un único movimiento de este tipo por cada posible agarre de cada una de las 6 direcciones existentes en nuestro retículo hexagonal (N, NW, SW, S, SE NE).

Cabe destacar que este tipo de reglas es de las menos prioritarias de entre las reglas de movimiento, exceptuando ciertos casos especiales que serán expuestos más adelante cuando hablemos de las situaciones en que és necesario moverse a pesar de que ello implique un aumento de la función potencial.

4.1. Lista de reglas

Existen un total de 36 reglas relacionadas con el movimiento de rotación. Podemos dividir estas 36 reglas en 6 subgrupos de 6 reglas cada uno. Cada subgrupo corresponde a una de las 6 direcciones posibles por las que el módulo puede estar agarrado en nuestro retículo hexagonal. Como ya hemos dicho, el módulo intenta rotar exactamente una posición en el sentido de las agujas del reloj, por lo que cada uno de estos subgrupos intenta realizar un movimento único respecto al resto.

Las reglas que implementan las rotaciones son muy parecidas de un subgrupo a otro, cambiando solo algunos parámetros y algunas coordenadas de las comprobaciones y de las acciones a realizar pero no cambiando la naturaleza de estas. A continuación mostraremos el total de reglas de rotación, pero no lo haremos dividiéndolas en subgrupos, sino que las mostraremos organizándolas según el tipo de tarea que realiza cada una.

Al primer conjunto de reglas las llamamos reglas de declaración de intención. El objetivo de estas es, en caso de que el módulo satisfazca las condiciones para poder girar, declarar su intención de hacerlo mediante un cambio de estado. Esto es necesario para que antes de realizar la rotación, un módulo pueda conocer las intenciones de sus vecinos cercanos y evitar así cualquier colisión, esto es, que dos o más módulos vayan a parar a una misma posición del universo siendo totalmente inconscientes de ello. Gracias a que todos los módulos anuncian su intención de ir a una posición determinada antes de hacerlo, cada uno de ellos puede decidir abortar su movimiento si ve que otro está a punto de invadir la casilla a la que quería ir. En la Figura 4 se observa un esquema de las reglas de declaración de intención.

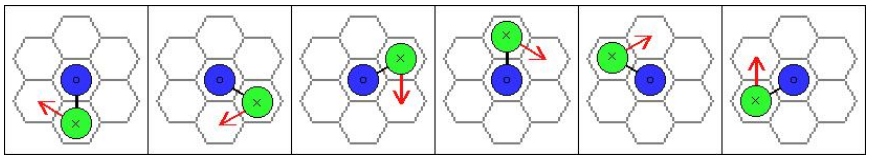

FIGURA 4. Reglas de declaración de intención del movimiento de rotación.

El segundo conjunto de reglas se encarga de la cancelación del movimiento, y se ejecutan sobre aquellos módulos que han expresado su intención de moverse, pero que de repente han dejado de cumplir alguna de las condiciones necesarias para

realizar la rotación. Algunos motivos por los que de repente el módulo ha dejado de ser capaz de rotar pueden ser producidos por el movimento que ya se ha producido de módulos cercanos, que pueden haber invadido u obstruido el desplazamiento que el módulo quería realizar. La Figura 5 muestra las reglas de cancelación de intención.

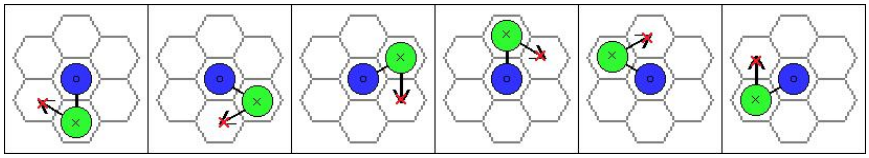

FIGURA 5. Reglas de cancelación de intención del movimiento de rotación.

En el tercer conjunto de reglas incluimos 3 reglas de cada subgrupo, es decir, un total de 18 reglas. El resultado de estas reglas también es la cancelación del giro anunciado, pero en este caso la cancelación se produce por un motivo diferente. En este tipo de reglas el módulo observa que hay otro con intención de moverse a la misma posición a la que él quiere ir y al mismo tiempo. Esto solo puede suceder viniendo el otro módulo desde un número limitado de posiciones del universo. En concreto, solo existen 3 posiciones desde las que el otro módulo puede venir, y de ahí salen las 3 reglas de comprobación. Aunque se podrían juntar todas las condiciones en una sola regla, separándolas en tres conseguimos que el resultado sea más intuitivo y fácil de leer.

Mientras que en las reglas del conjunto anterior se cancelaba el movimiento a causa de una obstrucción ya producida en el momento de la propia declaración de intención, en estas reglas el movimiento se cancela al detectar una colisión que aún no se ha producido pero que se producirá si no lo evitamos. El criterio, en este caso, es que los módulos con función potencial más alta cedan el paso a los de función potencial más baja. En la Figura 6 se muestran este tipo de reglas de cancelación.

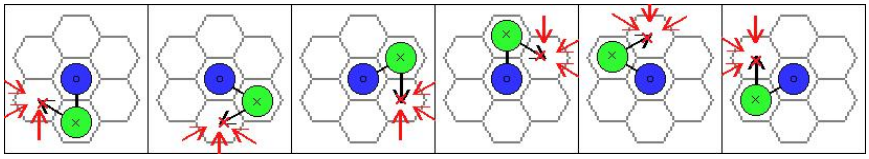

FIGURA 6. Reglas de cancelación por conflicto de la intención del movimiento de rotación.

El último conjunto de reglas es finalmente el que incluye las reglas que ejecutan el movimiento de rotación. Conseguimos que solo se ejecuten si han realizado las comprobaciones de todas las demás reglas correspondientes a este movimiento a base de asignarles a estas una prioridad más baja. Por lo tanto, estas reglas solo se aplican si no se han satisfecho las condiciones de ninguna otra de las reglas mencionadas en este apartado. El resultado de estas es la ejecución del movimiento de rotación y el cambio de estado del módulo al estado que tenía antes de declarar su intención de movimiento. En la Figura 7 observamos el resultado de la aplicación de las reglas de rotación.

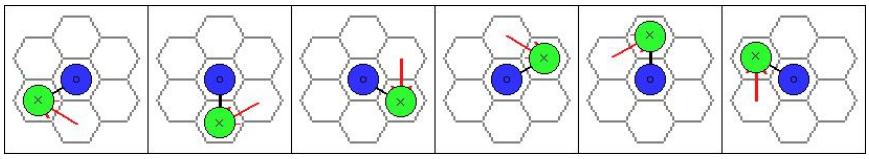

FIGURA 7. Reglas de ejecución del movimiento de rotación.

4.2. Descripción de las reglas

Para describir las reglas de esta fase, explicaremos detalladamente las partes de una sola regla de cada uno de los conjuntos explicados en el apartado anterior, en lugar de hacerlo por subgrupos como hemos hecho con otros tipos de reglas. Esto se debe a que las reglas son muy parecidas de un conjunto a otro, cambiando sólo los parámetros de acciones y condiciones. De esta forma, basta con explicar 4 de las 36 reglas, decir cómo se han implementado estas, y por último qué haría falta cambiar para obtener el resto de ellas.

Empezamos describiendo la regla de declaración de intención de un módulo conectado por el sur. El resto de reglas de declaración de intención correspondientes al resto de direcciones se obtienen mediante pequeñas modificaciones relativas a las posiciones que la regla debe consultar para realizar las comprovaciones necesarias. La regla es la siguiente:

Turn SE 1

8008

Sfw__A (E1,-1 A000100) (!(!(E0,1) !(F0,1 !(!(T0,1,pfc_*) !(T0,1,****A))))

!(!(E1,0) !(F1,0 !(!(T1,0,pfc_*) !(T1,0,****A)))))

SfwSEA

Esta regla tiene más prioridad que las de cancelación o las de ejecución del movimiento porque antes de hacer cualquiera de esas dos cosas el módulo debe haber expresado previamente su intención de moverse.

La precondición de la regla se divide en varias condiciones. Todas ellas deben satisfacerse para que la acción se ejecute:

- **Sfw__A**: Afecta a módulos que han sido activados.
- **A000100**: Afecta a módulos que sólo están conectados al sur.
- **E1,-1**: Afecta a los módulos desde los que la posición a la que quieren dirigirse (siguiedo el sentido de rotación horario) esta vacía.
- **!(!(E0,1) !(F0,1 !(!(T0,1,pfc_*) !(T0,1,****A))))**: Afecta a aquellos módulos que hayan podido analizar todas las alternativas de cambio de agarre al

módulo opuesto. Para ello se comprueba que o bien esa posición esté vacía, o bien esté ocupada pero por un agente que ya haya sido comprobado o que no pueda ser elegido para agarrase.

- **!(!(E1,0) !(F1,0 !(!(T1,0,pfc_*) !(T1,0,****A))))**: Afecta a aquellos módulos que hayan podido analizar todas las alternativas de cambio de agarre al módulo intermedio.

La acción **SfwSEA** consiste simplemente en cambiar el estado del módulo al nuevo estado de intención de movimiento hacia la posición del sureste. Para otras reglas similares del mismo conjunto el estado marcará la dirección en la que intenta moverse el módulo. Este cambio de estado será necesario para satisfacer las condiciones del resto de reglas que se describen a continuación.

Las reglas de cancelación por obstrucción se ejecutan sobre aquellos módulos que han mostrado su intención de moverse, pero que han dejado de satisfacer las condiciones necesarias para realizar el movimiento. A continuación veremos la regla de cancelación por obstrucción de un módulo conectado por el sur:

Turn SE - cant move

8007

SfwSEA !(E1,-1 A000100)

Sfw__A

Esta regla tiene más prioridad que las de ejecución del movimiento pero menos que las de intención porque antes de realizar el movimiento debemos realizar esta comprobación, y antes de realizarla el módulo ha debido expresar su intención de moverse.

La precondición de la regla se divide de nuevo en varias condiciones que deben satisfacerse para que la acción se ejecute:

- **SfwSEA**: Afecta a módulos que han expresado su intención de moverse hacia el sureste. Para el resto de reglas de este conjunto, el estado marcará la dirección en la que intenta moverse el módulo.

- **!(E1,-1 A000100)**: Afecta solo a aquellos módulo para los cuales la posición a la que querían desplazarse ya no está vacía, o bien ya no están agarrados al sur.

La acción **Sfw__A** consiste en cambiar el estado del módulo al estado que tenía antes de expresar su intención de movimiento. Después de este cambio de estado, el módulo volverá a estar listo para intentar moverse tan pronto como vuelva a satisfacer las condiciones de intención de movimiento.

Las reglas de cancelación por conflicto se ejecutan sobre aquellos módulos que han mostrado su intención de moverse, pero que detectan que otro módulo cercano tambien tiene la intención de moverse a esa misma posición y que además tiene menor función potencial. Hay 18 reglas de este tipo, 3 por cada uno de los 6 posibles movimientos. Esto se debe a que en cada movimiento, hay exactamente 3 módulos que pueden querer ir a parar a la posición deseada. Las 18 reglas tienen una estructura similar, y es por ello que aquí sólo presentamos una de ellas y explicamos los cambios necesarios para reproducir el resto.

A continuación veremos la regla de cancelación por conflicto de un módulo conectado por el sur que detecta que otro módulo proveniente del noreste desea moverse a la misma posición:

Turn SE win SE-NE

8007

SfwSEA F2,-1 T2,-1,fwSWA V2,-1,C000 C000

Sfw__A

Esta regla tiene la misma prioridad que las reglas de cancelación por obstrucción por las mismas razones que antes hemos explicado para las reglas de cancelacion de movimiento.

La precondición de la regla se divide de en varias condiciones que deben satisfacerse para que la acción se ejecute:

- **SfwSEA**: Afecta a módulos que han expresado su intención de moverse hacia el sureste. Para el resto de reglas de este conjunto, el estado marcará la dirección en la que intenta moverse el módulo.
- **F2,-1 T2,-1,fwSWA**: Afecta a aquellos módulos que han detectado que desde la única posición posible, otro módulo intenta acceder a la misma posición que él en dirección noreste.
- **V2,-1,C000 C000**: Afecta a aquellos módulos que habiendo detectado el conflicto, tengan mayor función potencial que el otro módulo que desea realizar el movimiento.

La acción **Sfw__A** consiste en cambiar el estado del módulo al estado que tenía antes de expresar su intención de movimiento. Observamos que en caso de conflicto, sólo se cancela el movimiento del módulo con mayor función potencial, de modo que se le permite al otro realizar el movimiento. Después de que el módulo perdedor cambie de estado, este volverá a estar listo para intentar moverse tan pronto como vuelva a satisfacer las condiciones de intención de movimiento.

Por último, la regla de ejecución del movimiento finalmente realiza la rotación si ninguna obstruccion ni ningún conflicto han sido detectados, o si el módulo ha

salido victorioso de todos los conflictos en los que haya participado. A continuación mostramos la regla de ejecución del movimiento de un módulo conectado al sur:

Turn SE - win

8006

SfwSEA

A001000 P1,-1 C020+C020 0001 C021-C021 0001 Sfw__A

Esta regla tiene menos prioridad que las reglas de intención y de cancelación ya que sólo permitimos el movimiento si el módulo ha expresado previamente su intención de realizarlo, y si todas las reglas de cancelación han sido comprobadas y no satisfechas.

La precondición de esta regla es mucho más simple que la de las anteriores. **SfwSEA** sólo comprueba que el módulo esté en un estado de intención de movimiento. No es necesario comprobar nada más debido a que la prioridad de esta regla fuerza a que sólo se ejecute si ninguna de las reglas de cancelación ha sido totalmente satisfecha. Si alguna de las reglas de cancelación se satisficiera, esta regla no podría ejecutarse debido a que tiene menor prioridad.

La acciones a realizar en este caso son las acciones propias de un módulo que camina a través de la frontera. Este módulo debe actualizar todos los contadores que necesitará más adelante para saber donde está y desde donde ha venido. A continuación se listan y se describen todas las acciones que realiza:

- **A001000**: Se desconecta de su vecino del sur y tras el movimiento, se conecta al del sur-oeste. En otras reglas de este tipo cambiaremos los parámetros de la acción.
- **P1,-1**: Realiza el movimiento deseado. En otras reglas de este tipo cambian las coordenadas del movimiento realizado.
- **C020+C020 0001 C021-C021 0001**: Actualiza el valor de las coordenadas relativas al lugar desde el cual el módulo empezó a moverse. A estos contadores se les suma la distancia recorrida tras cada movimiento realizado.
- **Sfw__A**: Por último se ejecuta el cambio de estado para que el módulo vuelva a un estado en el que pueda observar si puede realizar nuevos movimientos.

Observamos que no es necesario cambiar los valores de la función potencial ni los identificadores de la rama en la que se encuentra el módulo, puesto que ninguna de estos valores ha variado debido a que tras el movimiento, el módulo sigue estando enganchado a la misma posición del árbol.

4.3. Juegos de pruebas

Los juegos de prueba para comprobar el funcionamiento los movimientos de rotación son sencillos debido a que solo existe una rotación posible por cada dirección existente, y solo hay tres conflictos posibles por cada intento de rotación. Es fácil realizar un juego de pruebas que compruebe todas estas posibilidades de manera rápida.

5. Movimiento: Cambiar de agarre, siguiente

Las reglas de cambio de agarre al vecino siguiente se ejecutan cuando un módulo activo no puede seguir rotando porque la siguiente posición a la que debería ir está ocupada. Es condición necesaria para producirse el cambio de agarre que el nuevo módulo al que se quiere agarrar esté inactivo y tenga función potencial menor que la del módulo al que esta agarrado.

Este tipo de reglas tienen prioridad mayor que las de rotación, pero menor que las reglas de cambio de agarre que se corresponden a un salto intermedio u opuesto. Por lo tanto si el módulo es elegible para hacer un cambio de agarre al vecino siguiente y al mismo tiempo al vecino opuesto, se ejecutará la regla de cambio de agarre al opuesto por ser más prioritaria. Para asegurarnos de que no se ejecuta un cambio de agarre al vecino siguiente sin haber podido comprobar antes si es posible realizar el resto de tipos de cambios de agarre (por ejemplo, porque los módulos que lo deciden aún no saben su posicion en el árbol), añadimos condiciones a estas reglas que comprueban las posiciones de vecinos de saltos intermedios y opuestos con el fin de no realizar ninguna opción hasta que todas las alternativas hayan sido comprobadas. Decimos que estas condiciones existen para que el módulo tenga la posibilidad de analizar todas las alternativas de movimientos potenciales más prioritarios.

Cuando todas las condiciones de este tipo de reglas se satisfacen, los módulos que lo hacen se desconectan de su agarre actual en el árbol para conectarse con su agarre siguiente visto en orden antihorario, es decir, siguiendo la regla de la mano derecha.

5.1. Lista de reglas

Existen un total de 6 reglas de cambio de agarre al vecino siguiente, 1 por cada posible dirección por la que puede estar agarrado un módulo (N, NW, SW, S, SE NE). En la Figura 8 listamos el esquema de las 6 reglas que forman este tipo de movimiento.

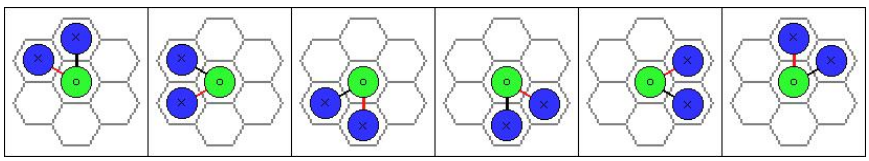

FIGURA 8. Reglas de cambio de agarre, siguiente. En negro el agarre antiguo y en rojo el nuevo agarre.

5.2. Descripción de las reglas

Para describir las reglas correspondientes a este tipo de movimiento, basta con explicar detalladamente las partes de una de ellas, ya que las 6 reglas del grupo son muy parecidas, y por lo tanto nos basta con ver cómo se ha implementado una, y con saber qué hace falta cambiar para obtener el resto.

Vamos a describir la regla de cambio de agarre al vecino siguiente correspondiente a desconectarse del sur para conectarse con el sureste:

Change attachment - Next S to SE

8010

Sfw__A A000100 F1,-1 T1,-1,pfc_* !T1,-1,****A V1,-1,C000 C000
(!(!(E0,1) !(F0,1 !(!(T0,1,pfc_*) !(T0,1,****A))))
!(!(E1,0) !(F1,0 !(!(T1,0,pfc_*) !(T1,0,****A)))))

A000010 C000+1,-1,C000 0001 C003+1,-1,C003 0000 C004+1,-1,C004 0000

Como ya hemos dicho, esta regla tiene más prioridad que las de rotación, pero menos que las reglas de cambio de agarre que se corresponden a un salto intermedio u opuesto. Esto se debe a que las otras reglas de salto avanzan más que esta y pueden evitar entrar en cuellos de botella no deseados. Es por eso que primero se comprobaran las otras reglas, dada su mayor prioridad y cuando se haya confirmado que no se pueden realizar dichos movimientos, se intentará llevar a cabo este.

La precondición de la regla se divide en varias condiciones. Todas ellas deben satisfacerse para que la acción se ejecute. A continuación las listamos y comentamos cada una de ellas:

- **Sfw__A**: Afecta a módulos que han sido activados.
- **A000100**: Afecta a módulos que sólo están conectados al sur.
- **F1,-1 T1,-1,pfc_* !T1,-1,****A**: Afecta a los módulos que tienen un vecino al sureste, que además sabe cuál es su posición en el árbol, pero que no ha sido activado.
- **V1,-1,C000 C000**: Solo afecta a aquellos módulos cuyo vecino del sureste tenga una función potencial menor que la del propio módulo, para que al cambiar de agarre mejore su propia función potencial.
- **!(!(E0,1) !(F0,1 !(!(T0,1,pfc_*) !(T0,1,****A))))**: Afecta a aquellos módulos que hayan podido analizar todas las alternativas de cambio de agarre al módulo opuesto.

- **!(!(E1,0) !(F1,0 !(!(T1,0,pfc_*) !(T1,0,****A))))**: Afecta a aquellos módulos que hayan podido analizar todas las alternativas de cambio de agarre al módulo intermedio.

La acciones a realizar en caso de satisfacer todas las condiciones son las propias de un módulo que continúa su camino a través de la frontera. Este módulo debe actualizar todos los contadores que necesitará más adelante para saber donde está y desde donde ha venido. A continuación se listan y se describen todas las acciones que realiza:

- **A000010**: Se desconecta de su vecino del sur y se conecta al del sureste.
- **C000+1,-1,C000 0001**: Actualiza el valor de la función potencial añadiendo una unidad al valor de la función del nuevo módulo al que ahora está agarrado.
- **C003+1,-1,C003 0000 C004+1,-1,C004 0000**: Actualiza los valores de rama a los de la nueva rama a la que ahora se encuentra agarrado.

5.3. Juegos de pruebas

Las pruebas para este tipo de reglas se encuentran en cualquier juego de pruebas que utilicemos, puesto que este tipo de movimientos son los más comunes después de los de rotación. Por ello cuando presentemos más adelante pruebas especificas para poder probar las reglas más problemáticas estaremos probando al mismo tiempo estas otras.

6. Movimiento: Cambiar de agarre, intermedio

Las reglas de cambio de agarre al vecino intermedio se ejecutan cuando un módulo activo detecta que para seguir avanzando por la frontera puede realizar un salto que le permita esquivar un agujero o un cuello de botella, y por tanto le permita avanzar, en un solo turno, múltiples posiciones a lo largo de la frontera.

La prioridad de este tipo de reglas es en circunstancias normales mayor que las de rotación y las de cambio de agarre al vecino siguiente, aunque existen ciertos casos en que tienen menor prioridad que las reglas de rotación. Veremos a qué se debe este nivel de prioridad variable, que resulta necesario para evitar *deadlocks*.

6.1. Observaciones

A diferencia de las reglas de cambio de agarre al vecino siguiente, que tenían todas ellas como condición necesaria que el nuevo módulo al que se quiere agarrar tenga menor función potencial que el módulo al que esta agarrado, este tipo de reglas de cambio de agarre tiene una casuística más variada. Tanto esta reglas de cambio de agarre al vecino intermedio, como las siguientes que definiremos de cambio de agarre al vecino opuesto, no se pueden guiar únicamente del valor de la función potencial para decidir si realizan el salto o no.

Con este tipo de reglas, se pueden dar situaciones en las que un módulo tenga que realizar el cambio de agarre aunque eso suponga empeorar su propia función potencial, del mismo modo que en otras situaciones un módulo tenga que decidir no cambiar de agarre aunque si lo hiciera mejoraría el valor de su función potencial. Esto se debe a que en ocasiones si solo nos guiara el objetivo de mejorar la función potencial del módulo, podríamos provocar embotellamientos o *deadlocks* que supondrían un fracaso del intento de reconfigurar la forma en una tira.

Para decidir cuándo un módulo debe realizar este tipo de cambios de agarre, hemos tenido que estudiar la casuística de los embotellamientos que nos podemos encontrar en el trancurso del proceso de reconfiguración canónica. En este apartado, nos centramos en definir esta casuística y en explicar las condiciones que nos sirven para programar las reglas que utilizaremos en el algoritmo de reconfiguración, y que deberán ser capaces de decidir en qué caso se encuentra el módulo, y qué acciones se aplican en dicho caso.

6.2. Cuellos de botella

Como ya hemos dicho, hemos estudiado los diferentes tipos de cuellos de botella que se pueden dar en nuestro retículo hexagonal. Tras analizar todas las posibilidades, hemos llegado a la conclusión de que, en términos de la topología del conjunto, solo se pueden dar ocho tipos de estrechamientos o cuellos de botella.

De estos ocho tipos de estrechamientos, cuatro son producto de cuellos de botella producidos por módulos pertenecientes a la misma rama del árbol lógico, mientras que los otros cuatro se producen a partir de cuellos de botella formados por módulos que pertenecen a dos ramas distintas.

Veremos que los cuatro últimos tipos de estrechamientos (aquellos formados por módulos de dos ramas diferentes) no suponen un problema a la hora de reconfigurar la forma inicial a la forma canónica, pero que son los otros cuatro tipos de estrechamientos (los formados por módulos de una misma rama) los que generan ciertos problemas a la hora de decidir qué acción se debe seguir para lograr el objetivo de la reconfiguración.

Por lo tanto, son estos cuatro casos los que hemos estudiado con mayor detenimiento (auqellos formados por módulos de una misma rama) para establecer cuándo es necesario que un módulo salte (por saltar nos referimos a hacer un cambio de agarre al vecino intermedio o al opuesto) y cuándo no.

En la Figura 9 se muestran los ocho tipos de estrechamientos topológicos que se pueden dar, incluyendo tanto los formados por módulos de una misma rama, como los formados por módulos de ramas diferentes. Recordamos que estos ocho tipos de estrechamiento son los únicos que se pueden producir, no solo en nuestro retículo hexagonal, sino en cualquier otro retículo.

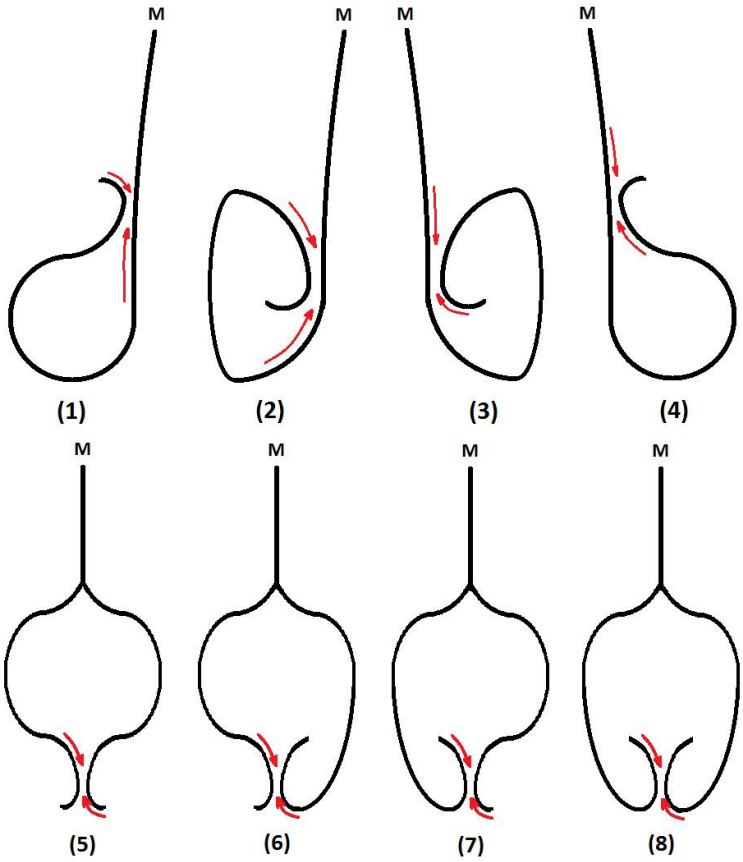

FIGURA 9. Todos los posibles tipos de embotellamientos. La letra M indica la posicion de la raiz del árbol generador.

A continuación vamos a estudiar los casos uno a uno, y a razonar en cuáles de estos casos el módulo que se encuentra en el cuello de botella debe decidir saltar y en cuáles no. Empezaremos estudiando los cuatro últimos casos (aquellos en los que el embotellamiento se crea por módulos de dos ramas diferentes), por considerar que su solución es considerablemente más sencilla que la de los cuatro primeros casos (los formados por módulos de una misma rama).

Tras haber descrito la actuación a seguir para deshacer los cuellos de botella de los cuatro últimos casos, podremos centrarnos en estudiar el resto de casos más cuidadosamente, esto es, ver qué problemas presentan con respecto a los casos más sencillos, y cómo hemos resuelto dichos problemas.

En el embotellamiento producido en el caso identificado con el número 5 en la Figura 9, si un módulo avanza desde el interior siguiendo la regla de la mano derecha y llega al cuello de botella, no debe saltar, porque eso no le ayuda a salir del embotallamiento. Incluso si se encuentra atascado y no puede avanzar, no hay peligro de *deadlock* puesto que por la forma de la rama, podemos garantizar que los

módulos de la misma recorrerán la frontera hasta que el módulo que se encuentra dentro del cuello de botella pueda salir.

En este mismo tipo de estrechamiento, si un módulo avanza desde el exterior y llega al cuello de botella, puede optar por saltar, si mejora el valor de su función potencial, o continuar en caso contrario. Si salta, se habrá ahorrado parte del recorrido por la frontera y llegará antes a formar parte de la tira. Si continúa, puede quedarse atascado al venir otro módulo de la parte interior en dirección contraria, pero incluso si esto pasa, sabemos que el atasco terminará deshaciéndose, y con ello, el cuello de botella.

En el embotellamiento identificado con el número 6 se aplican exactamente los mismos razonamientos que los aplicados en el caso anterior, es decir, es válida la condición de saltar solo si al hacerlo se obtiene una mejora en el valor de la función potencial.

En los embotellamientos de tipo 7 y 8 continua siendo válida la regla de saltar cuando esto suponga una mejora del valor de la función potencial, resultando que en cualquier caso se deshace una de las ramas, y finalmente el cuello de botella.

Llegados a este punto, sabemos como resolver los embotallemientos producidos por módulos de dos ramas diferentes del árbol lógico. Simplemente hemos de fijarnos si al saltar de una rama a otra, mejora el valor de la función potencial del módulo. Pero esta sencilla regla no sirve para resolver los embotallamientos producidos por módulos de una misma rama del árbol. A continuación estudiaremos este tipo de embotallamientos más detenidamente, observando qué problermas presentan, y proponiendo una solución para estos.

En la Figura 10 se observan los cuatro tipos de embotellamientos que nos disponemos a analizar.

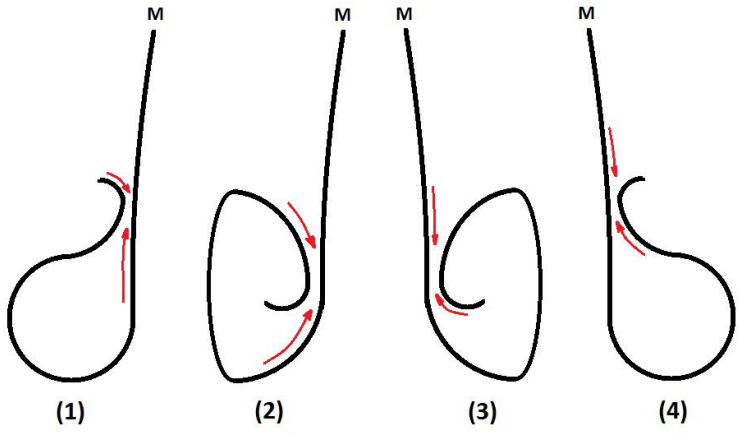

FIGURA 10. Todos los posibles tipos de embotellamientos formados por módulos de una misma rama.

En el primer tipo de estrechamiento, si un módulo avanza desde el exterior siguiendo la regla de la mano derecha y llega al cuello de botella, debe saltar al otro lado

para no impedir la salida de otros módulos que pudieran estar circulando desde el interior. Observamos que en este caso, el módulo que salta desde su posición en una rama hasta otra posición en la misma rama en la que estaba, mejora su función potencial y por lo tanto no se produce ningún riesgo de entrar en un bucle infinito de cambios de agarre.

En este mismo tipo de estrechamiento, si un módulo avanza desde el interior y llega al cuello de botella, este no debe saltar al otro lado, pues si lo hiciera nos encontraríamos en el caso anterior, en el que como ya hemos dicho el módulo volvería a saltar, y entraríamos en un ciclo de repetición sin salida. Además, observemos que si el módulo saltara, este empeoraría su función potencial, por lo que de momento se cumple la condición de solo saltar si esto supone una mejora en la función potencial.

En el segundo tipo de estrechamiento obtenemos los mismos resultados que en el primero. Si un módulo avanza desde el exterior siguiendo la regla de la mano izquierda y llega al cuello de botella, este debe saltar al otro lado para no impedir la salida de otros módulos que pudieran estar circulando desde el interior, mientras que si un módulo avanza desde el interior y llega al cuello de botella, este no debe saltar al otro lado. Por lo tanto, tenemos que todavía se cumple la condición de solo saltar si esto supone una mejora en la función potencial.

En el tercer y el cuarto estrechamiento, tenemos que si un módulo nacido en el interior se encuentra en el cuello de botella, en ninguno de los dos casos debe saltar, por que si lo hiciera el movimiento determinado por la regla de la mano derecha lo obligaría a volver a entrar en el agujero. En este caso tenemos dos ejemplos en los que el módulo debe decidir no saltar a pesar de que si lo hiciera estaría mejorando su función potencial. Observamos pues como la regla de saltar si con ello se mejora la función potencial no se aplica en estos casos.

Si en un estrechamiento del tercer tipo el módulo que se haya en el cuello de botella proviene del exterior, este puede taponar la salida de los módulos que provienen del interior, y por ello debe saltar. Este caso es un claro ejemplo en el que el módulo debe saltar a pesar de que al hacerlo está empeorando su función potencial. Es por ello que deducimos finalmente que la función potencial no es un buen método para tomar las decisiones de si el módulo debe saltar o no para los casos de estrechamiento producido por una misma rama. Deberemos buscar otro método para tomar esta decisión.

Por último, si un módulo proveniente del exterior se halla en un cuello de botella de un estrechamiento del cuarto tipo, este no debe saltar, ya que debido a la forma que tiene el estrechamiento, este acabará deshaciendose conforme los módulos que se haya en el exterior vayan recorriendo la frontera siguiendo la regla de la mano derecha.

Para decidir en qué momentos es necesario saltar y en qué momentos es contraproducente, primero miramos si al hacerlo mejoraríamos la función potencial o no. Hemos comprobado que si al saltar se mejora la función potencial, entonces el módulo debe saltar excepto en los embotellamientos 3 y 4 cuando el módulo proviene del interior. Para diferenciar estos casos de los embotellamientos 1 y 2, realizamos un

barrido antihoriario centrado en el módulo al que nos conectaríamos si hicieramos el salto.

Si al saltar mejoraramos la función potencial, y además al realizar el barrido antihorario el primer módulo de la misma rama que encontramos está más abajo en el arbol (tiene menor función potencial) que el módulo al que estamos a punto de saltar, entonces se realiza el salto. En la Figura 11 vemos como efectivamente estas condiciones permiten el salto que supone una mejora de la función potencial correspondiente a los embotellamientos 1 y 2, e impiden el salto en el caso de los embotellamientos 3 y 4.

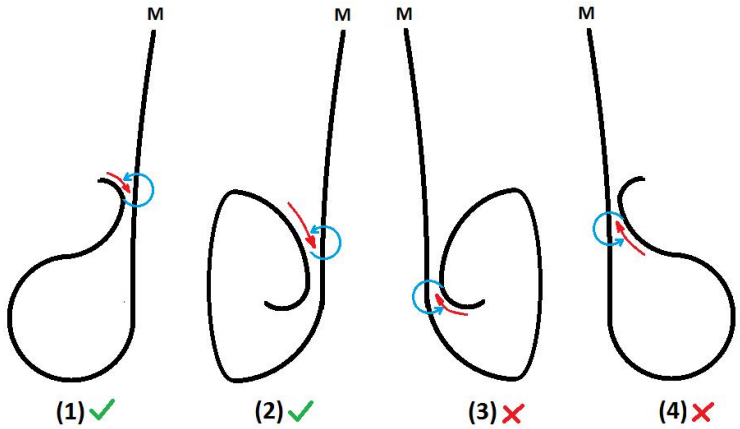

FIGURA 11. Salto con mejora de la función potencial.

Hemos visto que solo es necesario saltar a peor en un estrechamiento de tipo 3 cuando el módulo se introduce en el cuello de botella desde el exterior. Si realizamos el mismo barrido antihorario que hemos definido antes centrado en el módulo sobre el que queremos saltar, podremos diferenciar los embotellamientos 1 y 3 de los embotellamientos 2 y 4. Los cuellos de botella 1 y 3 satisfacen las condiciones del barrido antes descrito, pero debemos asegurarnos que no se produzca el salto si estamos ante un embotellamiento de tipo 1, por lo que añairemos otra condición que haga que en el caso 1 no se produzca el salto, mientras que en el caso 3 si que se produzca.

La diferencia entre ambos casos es que en el caso 1, el módulo que está dudando si saltar o no, puede continuar por la frontera sigiendo la regla de la mano derecha, por que al hacerlo está mejorando el valor de su función potencial, mientras que en el caso 3 el módulo no puede continuar por la frontera por que al hacerlo estaría empeorando el valor de su función potencial.

Por lo tanto, para determinar cuándo un módulo ha de saltar a pesar de que al hacerlo esté empeorando su función potencial se deben satisfacer las condiciones de barrido y no tener la oportunidad de seguir realizando otros movimientos de avance por la frontera. En la Figura 12 observamos el resultado de aplicar estas condiciones a cada caso.

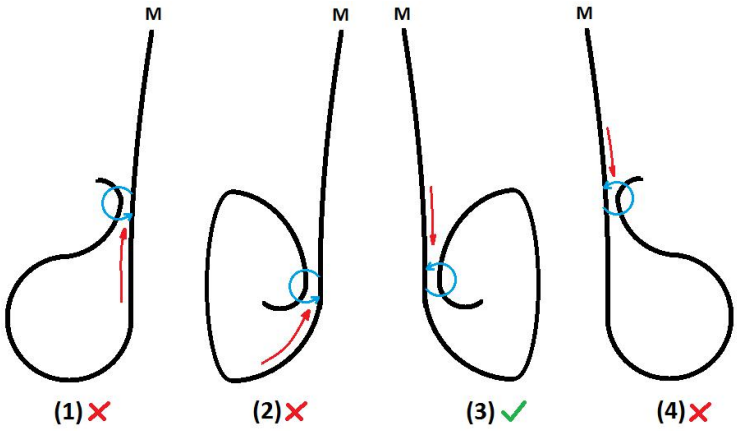

FIGURA 12. Salto con empeoramiento de la función potencial.

Hemos analizado todos los posibles casos de embotellamientos que se pueden producir cuando intentamos realizar la reconfiuración a la forma canónica. Este estudio se traduce en un conjunto de condiciones que tendremos que añadir a la reglas de cambio de agarre al vecino intermedio y de cambio de agarre al vecino opuesto para que el algoritmo de reconfiguración funcione correctamente.

6.3. Lista de reglas

Existen un total de 54 reglas relacionadas correspondientes al movimiento de cambio de agarre al vecino intermedio. Podemos dividir estas 54 reglas en 3 subgrupos. El primer subgrupo contiene 6 reglas y se corresponde con los cambios de agarre al vecino intermedio cuando al realizar este cambio el módulo cambia al mismo tiempo de rama del árbol. Los otros dos subgrupos contienen 24 reglas cada uno y se corresponden con los cambios de agarre al vecino intermedio sin cambiar de rama, pero uno de ellos mejorando su función potencial y el otro empeorándola.

El primer subgrupo, correspondiente al cambio de agarre al vecino intermedio con cambio de rama, es el que contiene un menor número de reglas debido a que la condición para saltar en este caso es mucho más sencilla que en los otros dos. A la hora de decidir si se cambia de agarre basta con mirar si al hacerlo se mejora el valor de la función potencial o no y, en caso afirmativo, se produce el salto. Las 6 reglas de este subgrupo se corresponden a las 6 posibles direcciones en las que se puede producir esta situación (N, NW, SW, S, SE NE). En la Figura 13 se observan las 6 reglas correspondientes a las 6 direcciones.

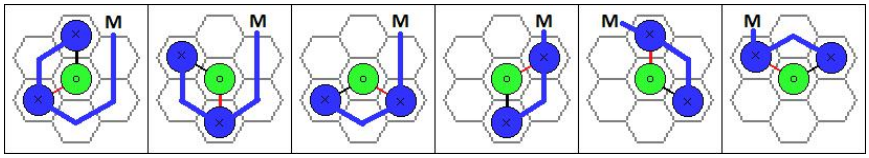

FIGURA 13. Reglas de cambio de agarre al vecino intermedio y de rama.

El segundo subgrupo se corresponde con las reglas de cambio de agarre al vecino intermedio sin cambio de rama y mejorando el valor de la función potencial. En este subgrupo hay un total de 24 reglas, 4 por cada una de las 6 direcciones posibles en las que se puede producir esta situación (N, NW, SW, S, SE NE). El motivo por el que hay 4 reglas por cada dirección es para poder realizar el barrido descrito en el apartado anterior.

Aunque se podría juntar cada conjunto de 4 reglas en una sola que contuviera las condiciones de todas ellas por separado, y de esta forma tener un subgrupo más reducido con sólo 6 reglas, las condiciones de estas serían extremadamente largas y dificiles de entender. Es por ello que aunque las 4 reglas sirvan para un único propósito, hemos preferido implementarlas de esta manera. En la Figura 14 se observan las 6 reglas correspondientes a los 6 cambios de agarre al vecino intermedio sin cambio de rama pero mejorando el valor de la función potencial.

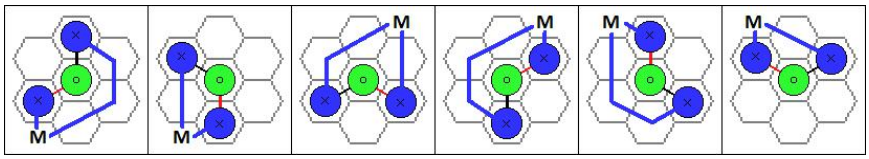

FIGURA 14. Reglas de cambio de agarre al vecino intermedio sin cambio de rama pero mejorando el valor de la función potencial.

El tercer subgrupo, correspondiente a la reglas de cambio de agarre al vecino intermedio sin cambio de rama pero empeorando el valor de la función potencial es muy parecido al segundo en cuanto a su estructura. Aunque las reglas son diferentes, este subgrupo también consta de 24 reglas, habiendo 4 de ellas para cada una de las 6 direcciones de nuestro retículo (N, NW, SW, S, SE NE). Esto sucede porque estas reglas también necesitan realizar el barrido descrito en el apartado anterior y por ello hemos optado por dividir la condición en 4 reglas.

Como en el segundo subgrupo, se podría juntar cada conjunto de 4 reglas en una sola, pero por el mismo motivo que antes, hemos preferido implementarlas por separado. En la Figura 15 se observan las 6 reglas correspondientes a las 6 direcciones existentes en el retículo hexagonal.

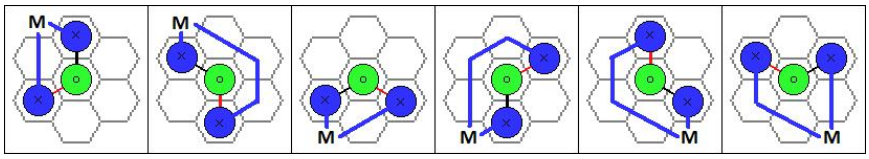

FIGURA 15. Reglas de cambio de agarre al vecino intermedio sin cambio de rama y empeorando el valor de la función potencial.

6.4. Descripción de las reglas

Debido a la gran cantidad de reglas de este tipo y a las similitudes que presentan unas con otras, sólo vamos a explicar una única regla de cada uno de los subgrupos que hemos mencionado.

Primero nos disponemos a explicar una regla del primer subgrupo, correspondiente al cambio de agarre al vecino intermedio con cambio de rama. Hemos escogido la regla que cambia el agarre de un módulo que está enganchado por el sur para agarrarse a su vecino del noreste:

Change attachment - Intermediate S to NE diff branch

8014

Sfw__A A000100 F1,0 T1,0, pfc_* !T1,0, ****A V1,0, C004 C000
!(!(E0,1) !(F0,1 !(!(T0,1,pfc_*) !(T0,1,****A))))

A000001 C000 + 1,0, C000 0001 C003 + 1,0, C003 0000 C004 + 1,0, C004 0000

Esta regla tiene más prioridad que las de rotación y que las de cambio de agarre al vecino siguiente, pero menos que las de cambio de agarre al vecino opuesto. Esto se debe a que este tipo de saltos avanzan más que las rotaciones o que los cambios al vecino siguiente, pero avanzan menos que los cambios al vecino opuesto.

La precondición de la regla se divide en varias condiciones. Todas ellas deben satisfacerse para que la acción se ejecute. A continuación listamos y comentamos cada una de ellas:

- **Sfw__A**: Afecta a módulos que han sido activados.
- **A000100**: Afecta a módulos que sólo están conectados al sur.
- **F1,0 T1,0, pfc_* !T1,0, ****A**: Afecta a los módulos que tienen un vecino al noreste, que además sabe cuál es su posición en el árbol, pero que no ha sido activado.
- **V1,0, C004 C000**: Solo afecta a aquellos módulos cuyo vecino del noreste tenga una menor función potencial que el propio módulo, para que al cambiar de agarre mejore su propia función potencial.
- **!(!(E0,1) !(F0,1 !(!(T0,1,pfc_*) !(T0,1,****A))))**: Afecta a aquellos módulos que hayan podido descartar todas las alternativas de cambio de agarre al módulo opuesto, o bien porque esa posición esté vacía, o bien porque esté ocupada por un agente que ya haya sido comprobado o no pueda ser elegible.

La acciones a realizar en caso de satisfacer todas las condiciones son las propias de actualización de sus contadores tras el movimiento. Este módulo debe actualizar todos los contadores que necesitará más adelante para saber dónde está y desde dónde ha venido. A continuación se listan y se describen todas las acciones:

- **A000001**: Se desconecta de su vecino del sur y se conecta al del noreste.

- **C000 + 1,0, C000 0001**: Actualiza el valor de la función potencial añadiendo una unidad al valor de la función del nuevo módulo al que ahora está agarrado.

- **C003 + 1,0, C003 0000 C004 + 1,0, C004 0000**: Actualiza los valores de rama a los de la nueva rama en los que ahora se encuentra tras el cambio de agarre.

A continuación nos disponemos a explicar la última de las 4 reglas correspondientes al segundo subgrupo de reglas de este tipo de cambio de agarre. Estas reglas se corresponden con el cambio de agarre al vecino intermedio sin cambio de rama y mejorando el valor de la función potencial. En concreto vamos a describir la regla que cambia el agarre de un módulo que esta agarrado por el sur para agarrarse a su vecino del noreste:

Change attachment - Intermediate S to NE same branch better Pot 4

8013

Sfw__A A000100 F1,0 T1,0, pfc_* !T1,0, ****A
(V1,0,C000 C000 V1,0,C003 C004 !V1,0,C004 C003)
(!(W1,0,C002 0004 !V1,0,C002 0004)
!(W1,0,C002 0005 !V1,0,C002 0005)
!(W1,0,C002 0006 !V1,0,C002 0006)
F1,1 !T1,1,****A T1,1,pfc_* (V1,1,C003 C004 !V1,1,C004 C003)
!(W1,0,C002 0001 !V1,0,C002 0001))
!(!(E0,1) !(F0,1 !(!(T0,1,pfc_*) !(T0,1,****A))))

A000001 C000 + 1,0, C000 0001 C003 + 1,0, C003 0000 C004 + 1,0, C004 0000

Esta regla tambien tiene más prioridad que las de rotación y que las de cambio de agarre al vecino siguiente pero menos que las de cambio de agarre al vecino opuesto. Sin embargo, esta regla tiene menos prioridad que la regla anterior, porque es mejor comprobar primero la anterior regla que es más sencilla, que todas estas reglas que requieren del barrido.

La precondición de la regla se divide en varias condiciones. Todas ellas deben satisfacerse para que la acción se ejecute. A continuación listamos y comentamos cada una de ellas:

- **Sfw__A**: Afecta a módulos que han sido activados.

- **A000100**: Afecta a módulos que sólo están conectados al sur.

- **F1,0 T1,0, pfc_* !T1,0, ****A**: Afecta a los módulos que tienen un vecino al noreste, que además sabe cuál es su posición en el árbol, pero que no ha sido activado.

- **V1,0,C000 C000 V1,0,C003 C004 !V1,0,C004 C003**: Solo afecta a aquellos módulos cuyo vecino del noreste tenga una menor función potencial que el propio módulo y que además pertenezca a la misma rama.

- **(!(W1,0,C002 0004 !V1,0,C002 0004) !(W1,0,C002 0005 !V1,0,C002 0005) !(W1,0,C002 0006 !V1,0,C002 0006) F1,1 !T1,1,****A T1,1,pfc_* (V1,1,C003 C004 !V1,1,C004 C003) !(W1,0,C002 0001 !V1,0,C002 0001)) !(!(E0,1) !(F0,1 !(!(T0,1,pfc_*) !(T0,1,****A))))**: Afecta a aquellos módulos que satisfagan la condición de barrido, es decir, que detecten que el primer módulo perteneciente a su rama haciendo un barrido en sentido antihorario desde el módulo al que queremos saltar, tenga mayor función potencial que el propio módulo.

La acciones a realizar en caso de satisfacer todas las condiciones son las mismas que las de la regla anterior, dado que aunque la condición y la situación cambie, el resultado es el mismo (saltar al vecino intermedio).

- **A000001**: Se desconecta de su vecino del sur y se conecta al del noreste.

- **C000 + 1,0, C000 0001**: Actualiza el valor de la función potencial añadiendo una unidad al valor de la función del nuevo módulo al que ahora está agarrado.

- **C003 + 1,0, C003 0000 C004 + 1,0, C004 0000**: Actualiza los valores de rama a los de la nueva rama en los que ahora se encuentra tras el cambio de agarre.

Por último nos disponemos a explicar la última de las 4 reglas correspondientes al tercer subgrupo de reglas de cambio de agarre al vecino intermedio. Estas reglas se corresponden con el cambio de agarre sin cambio de rama y empeorando el valor de la función potencial. Vamos a describir de nuevo el caso en que cambia el agarre de un módulo que esta enganchado por el sur para agarrarse a su vecino del noreste:

Change attachment - Intermediate S to NE same branch worse Pot 4

8013

Sfw__A A000100 F1,0 T1,0, pfc_* !T1,0, ****A
(!V1,0,C000 C000 V1,0,C003 C004 !V1,0,C004 C003)
(!(W1,0,C002 0004 !V1,0,C002 0004)
!(W1,0,C002 0005 !V1,0,C002 0005)
!(W1,0,C002 0006 !V1,0,C002 0006)
F1,1 !T1,1,****A T1,1,pfc_* (V1,1,C003 C004 !V1,1,C004 C003)
!(W1,0,C002 0001 !V1,0,C002 0001))
!(!(E0,1) !(F0,1 !(!(T0,1,pfc_*) !(T0,1,****A))))

A000001 C000 + 1,0, C000 0001 C003 + 1,0, C003 0000 C004 + 1,0, C004 0000

Esta regla es muy parecida a la anterior, pero uno de los cambios más importantes que se observan respecto a la otra es su prioridad. Esta regla tiene menos prioridad que las reglas de rotación y por lo tanto que cualquier otra regla de movimiento. Esto se debe al motivo que hemos explicadode diferenciación de 2 casos de embotellamiento, uno de los cuales debía permitir el salto mientras que el otro no. Al hacer que esta regla tenga menos prioridad que las de rotación nos aseguramos de que solo se ejecutará si no se puede ejecutar ninguna de las reglas de rotación, es decir, si no puede seguir avanzando por la frontera.

La precondición de la regla se divide en varias condiciones. Todas ellas deben satisfacerse para que la acción se ejecute. A continuación listamos y comentamos cada una de ellas:

- **Sfw__A**: Afecta a módulos que han sido activados.
- **A000100**: Afecta a módulos que sólo están conectados al sur.
- **F1,0 T1,0, pfc_* !T1,0, ****A**: Afecta a los módulos que tienen un vecino al noreste, que además sabe cuál es su posición en el árbol, pero que no ha sido activado.
- **!V1,0,C000 C000 V1,0,C003 C004 !V1,0,C004 C003**: Solo afecta a aquellos módulos cuyo vecino del noreste tenga una mayor función potencial que el propio módulo y que además ambos pertenezcan a la misma rama.
- **(!(W1,0,C002 0004 !V1,0,C002 0004) !(W1,0,C002 0005 !V1,0,C002 0005) !(W1,0,C002 0006 !V1,0,C002 0006) F1,1 !T1,1,****A T1,1,pfc_* (V1,1,C003 C004 !V1,1,C004 C003) !(W1,0,C002 0001 !V1,0,C002 0001)) !(!(E0,1) !(F0,1 !(!(T0,1,pfc_*) !(T0,1,****A))))**: Afecta a aquellos módulos que satisfagan la condición de barrido, es decir, que detecten que el primer módulo perteneciente a su rama haciendo un barrido en sentido antihorario desde el módulo al que queremos saltar, tenga mayor función potencial que el propio módulo.
- **!(!(E0,1) !(F0,1 !(!(T0,1,pfc_*) !(T0,1,****A))))**: Afecta a aquellos módulos que hayan podido descartar todas las alternativas de cambio de agarre al módulo siguiente, para lo cual se comprueba que o bien esa posición esté vacía, o bien esté ocupada por un agente que ya haya sido comprobado o no pueda ser elegible.

La acciones a realizar en caso de satisfacer todas las condiciones son las mismas que las de las dos reglas anteriores, porque aunque la condición y la situación cambie, el resultado es el mismo (saltar al vecino intermedio).

- **A000001**: Se desconecta de su vecino del sur y se conecta al del noreste.
- **C000 + 1,0, C000 0001**: Actualiza el valor de la función potencial añadiendo una unidad al valor de la función del nuevo módulo al que ahora está agarrado.

- **C003 + 1,0, C003 0000 C004 + 1,0, C004 0000**: Actualiza los valores de rama a los de la nueva rama en los que ahora se encuentra tras el cambio de agarre.

6.5. Juegos de pruebas

Los juegos de pruebas para comprobar el buen funcionamiento de este tipo de reglas son más complejos que los de las reglas anteriores, pero son básicamente los mismos que serán necesarios para el siguiente tipo de reglas, correspondiente al cambio de agarre al vecino opuesto. Es por eso que los introduciremos todos a la vez en el apartado siguiente para no caer en repeticiones innecesarias.

7. Movimiento: Cambiar de agarre, opuesto

Como en el caso del anterior grupo de reglas, las reglas de cambio de agarre al vecino opuesto se ejecutan cuando un módulo activo detecta que para seguir avanzando por la frontera puede realizar un salto que le permita esquivar o un cuello de botella, y por tanto le permita avanzar múltiples posiciones en un solo turno.

La prioridad de este tipo de reglas es en circunstancias normales mayor que las de rotación y las del resto de reglas de cambio de agarre, aunque existen ciertos casos que tienen menor prioridad que las reglas de rotación. Esto se debe a las mismas razones presentadas anteriormente para el caso de las reglas de cambio de agarre al vecino intermedio.

Este tipo de reglas, como las anteriores, pueden decidir realizar un salto aunque eso suponga empeorar el valor de su función potencial, o en el caso contrario, pueden decidir no saltar a pesar de que si lo hicieran mejorarían su función potencial. Para entender esto, exactamente las mismas consideraciones que en el Apartado 6.2 en el que hablabamos de las reglas de cambio de agarre al vecino intermedio son válidas aquí. No duplicaremos aquí la teoría expuesta sobre los cuellos de botella que hemos detallado en el Apartado 6.2.

7.1. Lista de reglas

Existen un total de 54 reglas de cambio de agarre al vecino opuesto. Podemos dividir estas 54 reglas en 3 subgrupos. El primer subgrupo contiene 6 reglas y se corresponde con los cambios de agarre cuando al realizar este cambio el módulo cambia al mismo tiempo de rama del árbol. Los otros dos subgrupos contienen 24 reglas cada uno y se corresponden con los cambios de agarre sin cambiar de rama, uno de ellos mejorando su función potencial y el otro empeorándola.

Obsérvese el gran parecido de estos subgrupos con los subgrupos correspondientes a las reglas de cambio de agarre al vecino intermedio descritas en el Apartado 6.3. No solo la clasificación en subgrupos es idéntica sino que además las reglas de cada subgrupo son muy parecidas.

El primer subgrupo, correspondiente al cambio de agarre al vecino opuesto con cambio de rama, es el que contiene un menor número de reglas debido a que la

condición para saltar en este caso es mucho más sencilla que en los otros dos casos. A la hora de decidir si se cambia de agarre y de rama basta con mirar si al hacerlo se mejora el valor de la función potencial o no. Las 6 reglas de este subgrupo se corresponden a las 6 direcciones posibles en las que se puede producir esta situación (N, NW, SW, S, SE NE). En la Figura 16 se observan las 6 reglas correspondientes a este tipo de movimiento en las 6 direcciones.

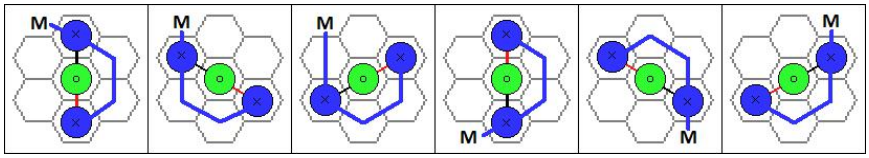

FIGURA 16. Reglas de cambio de agarre al vecino opuesto con cambio de rama.

El segundo subgrupo se corresponde con las reglas de cambio de agarre al vecino opuesto sin cambio de rama y mejorando el valor de la función potencial. En este subgrupo hay un total de 24 reglas, 4 por cada una de las 6 direcciones posibles en las que se puede producir esta situación (N, NW, SW, S, SE NE). La existencia de 4 reglas por cada dirección es justificada con la necesidad de realizar el barrido. En la Figura 17 se muestra el resultado del cambio de agarre en estas circunstancias para cada dirección posible.

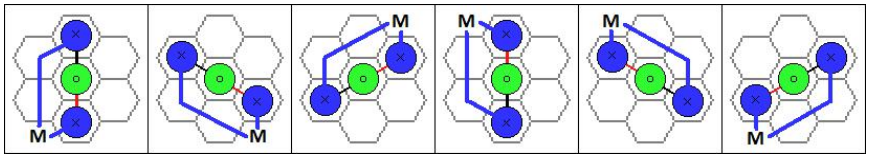

FIGURA 17. Reglas de cambio de agarre al vecino opuesto sin cambio de rama pero mejorando el valor de la función potencial.

El tercer subgrupo, correspondiente a la reglas de cambio de agarre al vecino opuesto sin cambio de rama pero empeorando el valor de la función potencial es muy parecido al segundo en cuanto a su estructura. Aunque las reglas son diferentes, este subgrupo también consta de 24 reglas, habiendo 4 de ellas para cada una de las 6 direcciones de nuestro retículo (N, NW, SW, S, SE NE). Esto sucede porque estas reglas también necesitan realizar el barrido antes mencionado y por ello hemos optado por dividir la condición en 4. En la Figura 18 se muestra el resultado de este tipo de reglas.

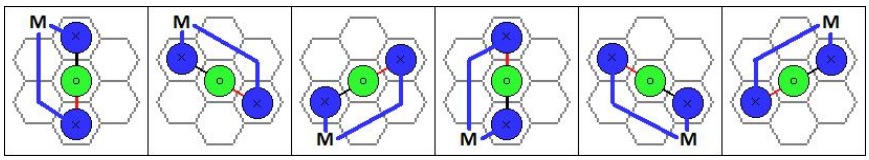

FIGURA 18. Reglas de cambio de agarre al vecino opuesto sin cambio de rama y empeorando el valor de la función potencial.

7.2. Descripción de las reglas

Debido a la gran cantidad de reglas que hay de este tipo y a las similitudes que presentan unas con otras, sólo vamos a explicar una única regla de cada uno de los subgrupos que hemos mencionado.

Primero nos disponemos a explicar una regla del primer subgrupo, correpondiente al cambio de agarre al vecino opuesto con cambio de rama. Hemos cogido la regla que cambia el agarre de un módulo que esta enganchado por el sur para agarrarse a su vecino del norte:

Change attachment - Opposite S to N diff branch

8018

Sfw__A A000100 F0,1 T0,1, pfc_* !T0,1, ****A V0,1, C004 C000

A100000 C000 + 0,1, C000 0001 C003 + 0,1, C003 0000 C004 + 0,1, C004 0000

Esta regla tiene más prioridad que todas las demás reglas de movimiento. Esto se debe a que este tipo de saltos avanzan más que las rotaciones o que cualquier otro tipo de cambio de agarre.

La precondición de la regla se divide en varias condiciones. Todas ellas deben satisfacerse para que la acción se ejecute. A continuación listamos y comentamos cada una de ellas:

- **Sfw__A**: Afecta a módulos que han sido activados.
- **A000100**: Afecta a módulos que sólo están conectados al sur.
- **F0,1 T0,1, pfc_* !T0,1, ****A**: Afecta a los módulos que tienen un vecino al norte, que además sabe cuál es su posición en el árbol, pero que no ha sido activado.
- **V0,1, C004 C000**: Solo afecta a aquellos módulos cuyo vecino del norte tenga una menor función potencial que el propio módulo, para que al cambiar de agarre mejore su propia función potencial.

La acciones a realizar en caso de satisfacer todas las condiciones son las propias de actualización de sus contadores tras el movimiento. Este módulo debe actualizar todos los contadores que necesitará más adelante para saber dónde está y desde dónde ha venido. A continuación se listan y se describen todas las acciones que realiza:

- **A100000**: Se desconecta de su vecino del sur y se conecta al del norte.
- **C000 + 0,1, C000 0001**: Actualiza el valor de la función potencial añadiendo una unidad al valor de la función del nuevo módulo al que ahora está agarrado.

- **C003 + 0,1, C003 0000 C004 + 0,1, C004 0000**: Actualiza los valores de rama a los de la nueva rama en los que ahora se encuentra tras el cambio de agarre.

A continuación nos disponemos a explicar la última de las 4 reglas correspondientes al segundo subgrupo de reglas de este tipo de cambio de agarre. Estas reglas se corresponden con el cambio de agarre al vecino opuesto sin cambio de rama y mejorando el valor de la función potencial. En concreto vamos a describir la regla que cambia el agarre de un módulo que esta enganchado por el sur para agarrarse a su vecino del norte:

Change attachment - Opposite S to N same branch better Pot 4

8017

Sfw__A A000100 F0,1 T0,1, pfc_* !T0,1, ****A
(V0,1, C000 C000 V0,1,C003 C004 !V0,1,C004 C003)
(!(W0,1,C002 0005 !V0,1,C002 0005)
!(W0,1,C002 0006 !V0,1,C002 0006)
!(W0,1,C002 0001 !V0,1,C002 0001)
F-1,2 !T-1,2,****A T-1,2,pfc_* (V-1,2,C003 C004 !V-1,2,C004 C003)
!(W0,1,C002 0002 !V0,1,C002 0002))

A100000 C000 + 0,1, C000 0001 C003 + 0,1, C003 0000 C004 + 0,1, C004 0000

Esta regla tambien tiene más prioridad que las del resto de movimientos. Sin embargo, tiene menos prioridad que la regla anterior, porque es mejor comprobar primero la anterior regla que es más sencilla, que todas estas reglas que requieren del barrido.

La precondición de la regla se divide en varias condiciones. Todas ellas deben satisfacerse para que la acción se ejecute. A continuación listamos y comentamos cada una de ellas:

- **Sfw__A**: Afecta a módulos que han sido activados.
- **A000100**: Afecta a módulos que sólo están conectados al sur.
- **F0,1 T0,1, pfc_* !T0,1, ****A**: Afecta a los módulos que tienen un vecino al norte, que además sabe cuál es su posición en el árbol, pero que no ha sido activado.
- **V0,1, C000 C000 V0,1,C003 C004 !V0,1,C004 C003**: Solo afecta a aquellos módulos cuyo vecino del norte tenga una menor función potencial que el propio módulo y que además pertenezca a la misma rama.

- **((!(W0,1,C002 0005 !V0,1,C002 0005) !(W0,1,C002 0006 !V0,1,C002 0006) !(W0,1,C002 0001 !V0,1,C002 0001) F-1,2 !T-1,2,****A T-1,2,pfc_* (V-1,2,C003 C004 !V-1,2,C004 C003) !(W0,1,C002 0002 !V0,1,C002 0002))**: Afecta a aquellos módulos que satisfagan la condición de barrido, es decir, que detecten que el primer módulo perteneciente a su rama haciendo un barrido en sentido antihorario desde el módulo al que queremos saltar, tenga mayor función potencial que el propio módulo.

La acciones a realizar en caso de satisfacer todas las condiciones son las mismas que las de la regla anterior, dado que aunque la condición y la situación cambie, el resultado es el mismo (saltar al vecino opuesto).

- **A100000**: Se desconecta de su vecino del sur y se conecta al del norte.
- **C000 + 0,1, C000 0001**: Actualiza el valor de la función potencial añadiendo una unidad al valor de la función del nuevo módulo al que ahora está agarrado.
- **C003 + 0,1, C003 0000 C004 + 0,1, C004 0000**: Actualiza los valores de rama a los de la nueva rama en los que ahora se encuentra tras el cambio de agarre.

Por último nos disponemos a explicar la última de las 4 reglas correspondientes al tercer subgrupo de reglas de cambio de agarre al vecino opuesto. Estas reglas se corresponden con el cambio de agarre sin cambio de rama y empeorando el valor de la función potencial. Vamos a describir de nuevo el caso en que cambia el agarre de un módulo que esta enganchado por el sur para agarrarse a su vecino del norte:

Change attachment - Opposite S to N same branch worse Pot 4

8004

Sfw__A A000100 F0,1 T0,1, pfc_* !T0,1, ****A
(!V0,1, C000 C000 V0,1,C003 C004 !V0,1,C004 C003)
(!(W0,1,C002 0005 !V0,1,C002 0005)
!(W0,1,C002 0006 !V0,1,C002 0006)
!(W0,1,C002 0001 !V0,1,C002 0001)
F-1,2 !T-1,2,****A T-1,2,pfc_* (V-1,2,C003 C004 !V-1,2,C004 C003)
!(W0,1,C002 0002 !V0,1,C002 0002))
(!(!(E1,-1) !(F1,-1 !(!(T1,-1,pfc_*) !(T1,-1,****A))))
!(!(E1,0) !(F1,0 !(!(T1,0,pfc_*) !(T1,0,****A)))))

A100000 C000 + 0,1, C000 0001 C003 + 0,1, C003 0000 C004 + 0,1, C004 0000

Esta regla es muy parecida a la anterior, pero uno de los cambios más importantes que se observan respecto a la otra es su prioridad. Esta regla tiene menos prioridad que las reglas de rotación y por lo tanto que cualquier otra regla de movimiento.

Esto se debe al motivo que hemos explicado de diferecniación de 2 casos de embotellamiento, uno de los cuales debía permitir el salto mientras que el otro no. Al hacer que esta regla tenga menos prioridad que las de rotación nos aseguramos de que solo se ejecutará si no se puede ejecutar ninguna de las reglas de rotación, es decir, si no puede seguir avanzando por la frontera por alguna razón.

La precondición de la regla se divide en varias condiciones. Todas ellas deben satisfacerse para que la acción se ejecute. A continuación listamos y comentamos cada una de ellas:

- **Sfw__A**: Afecta a módulos que han sido activados.
- **A000100**: Afecta a módulos que sólo están conectados al sur.
- **F1,0 T1,0, pfc_* !T1,0, ****A**: Afecta a los módulos que tienen un vecino al noreste, que además sabe cuál es su posición en el árbol, pero que no ha sido activado.
- **!V1,0,C000 C000 V1,0,C003 C004 !V1,0,C004 C003**: Solo afecta a aquellos módulos cuyo vecino del noreste tenga una mayor función potencial que el propio módulo y además pertenezca a la misma rama.
- **(!(W1,0,C002 0004 !V1,0,C002 0004) !(W1,0,C002 0005 !V1,0,C002 0005) !(W1,0,C002 0006 !V1,0,C002 0006) F1,1 !T1,1,****A T1,1,pfc_* (V1,1,C003 C004 !V1,1,C004 C003) !(W1,0,C002 0001 !V1,0,C002 0001)) !(!(E0,1) !(F0,1 !(!(T0,1,pfc_*) !(T0,1,****A))))**: Afecta a aquellos módulos que satisfagan la condición de barrido, es decir, que detecten que el primer módulo perteneciente a su rama haciendo un barrido en sentido antihorario desde el módulo al que queremos saltar, tenga mayor función potencial que el propio módulo.
- **(!(!(E1,-1) !(F1,-1 !(!(T1,-1,pfc_*) !(T1,-1,****A))))**: Afecta a aquellos módulos que hayan podido descartar todas las alternativas de cambio de agarre al módulo siguiente, para lo cual se comprueba que o bien esa posición esté vacía, o bien esté ocupada por un agente que ya haya sido comprobado o que no puede ser elegible.
- **!(!(E1,0) !(F1,0 !(!(T1,0,pfc_*) !(T1,0,****A)))))**: Afecta a aquellos módulos que hayan podido descartar todas las alternativas de cambio de agarre al módulo intermedio, para lo cual se comprueba que o bien esa posición esté vacía, o bien esté ocupada por un agente que ya haya sido comprobado o que no puede ser elegible.

La acciones a realizar en caso de satisfacer todas las condiciones son las mismas que las de las dos reglas anteriores, porque aunque la condición y la situación cambie, el resultado es el mismo (saltar al vecino opuesto).

- **A100000**: Se desconecta de su vecino del sur y se conecta al del norte.

- **C000 + 0,1, C000 0001**: Actualiza el valor de la función potencial añadiendo una unidad al valor de la función del nuevo módulo al que ahora está agarrado.

- **C003 + 0,1, C003 0000 C004 + 0,1, C004 0000**: Actualiza los valores de rama a los de la nueva rama en los que ahora se encuentra tras el cambio de agarre.

7.3. Juegos de pruebas

Los juegos de pruebas de este tipo de reglas son sin duda los más completos. Incluyen pruebas de todo el resto de tipos de reglas de movimientos y de todos los casos de cuellos de botellas antes mencionados.

Hemos creado cuatro configuraciones de módulos para probar las reglas de salto. Cada una de estas cuatro configuraciones se corresponde con uno de los cuatro tipos de cuellos de botella formados por módulos de una misma rama. En cada configuración, preparada para probar cada uno de los cuatro embotellamientos críticos, hemos replicado el embotellamiento en las 6 direcciones posibles de nuestro retículo (N, NW, SW, S, SE NE) para asegurarnos de que todas las reglas funcionan correctamente.

La Figura 19 muestra el juego de pruebas que hemos creado para comprobar el correcto funcionamiento de las reglas de cambió de agarre al vecino opuesto cuando los módulos se encuentran con un embotellamiento formado por módulos de una misma rama de los del primer tipo.

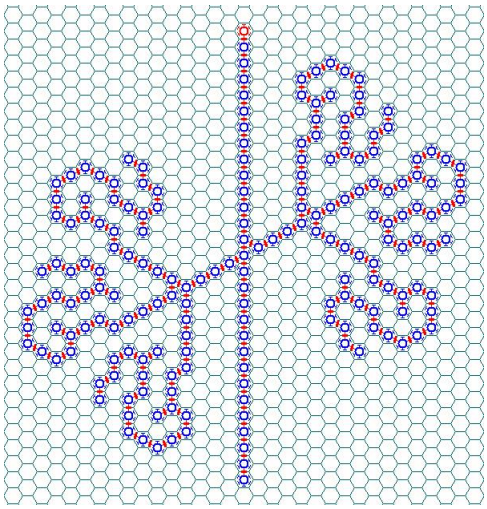

FIGURA 19. Embotellamientos del primer tipo en todas las direcciones.

La Figura 20 muestra el juego de pruebas que realiza las comprobaciones análogas al caso anterior pero sobre los embotellamientos formados por módulos de una misma rama con una topología como las del segundo tipo.

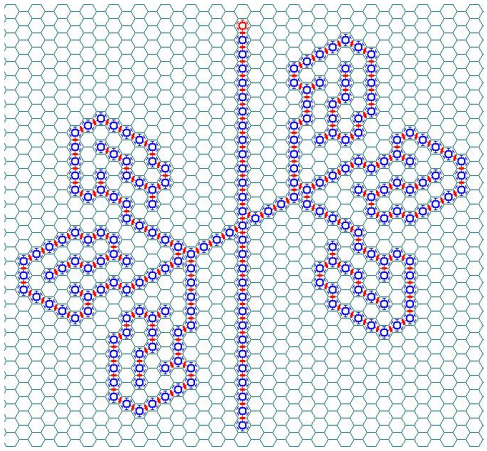

FIGURA 20. Embotellamientos del segundo tipo en todas las direcciones.

La Figura 21 muestra la configuración realizada para comprobar que las reglas funcionan correctamente ante un embotellamiento de los del tercer tipo formado por módulos de una misma rama.

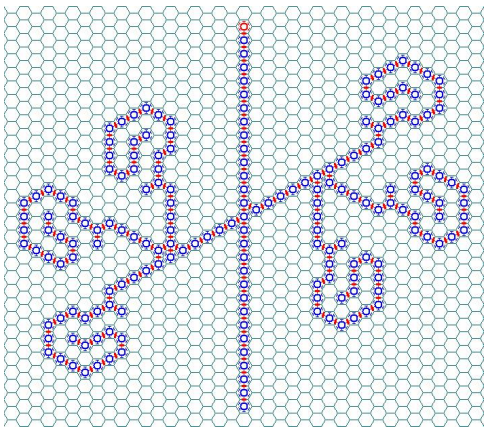

FIGURA 21. Embotellamientos del tercer tipo en todas las direcciones.

Y por último, la Figura 22 muestra el juego de pruebas correspondiente a la comprobación del correcto funcionamiento de las reglas cuando los módulos se hayan ante un embotellamiento de los de cuarto tipo formado por módulos de una misma rama.

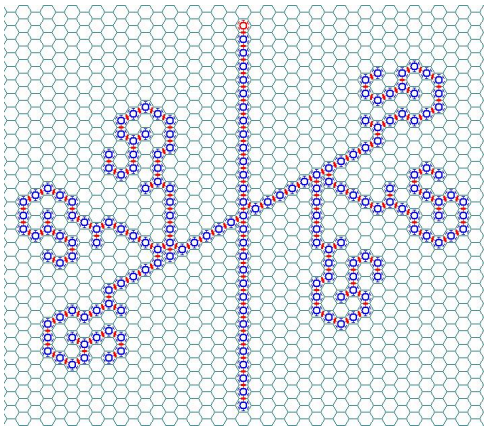

FIGURA 22. Embotellamientos del cuarto tipo en todas las direcciones.

El resultado de todos los juegos de prueba es el esperado. Los módulos deciden saltar o no hacerlo exactamente en las situaciones descritas en el Apartado 6.2. Recordemos que además estos juegos de prueba están diseñados para probar todos los otros tipos de regla, y para todos ellos también obtenemos resultados satisfactorios.

Tras probar todos los casos posibles propensos a error y obtener resultados positivos en todos ellos, nos atrevemos a decir que la implemetación de las reglas es correcta y que estas son capaces de reproducir el algoritmo de reconfiguración a la forma canónica correctamente.

8. Formación de la tira

Las reglas de formación de la tira son las últimas que ejecuta un módulo en la fase de reconfiguración canónica. Estas se ejecutan sobre aquellos módulos que en su movimiento por la frontera ya han pasado por delante del líder y están a punto de pasar a formar parte de la tira que representa a la forma canónica (están moviéndose por encima de esta).

Como sabemos que todos los módulos se han activado durante la reconfiguración, y han recorrido la frontera hasta llegar al líder, tenemos que en algún momento todos los módulos habrán pasado a formar parte de la tira, y cuando esto pase el algoritmo acabará.

8.1. Lista de reglas

Existen un total de 3 reglas en la fase de formación de la tira: una para finalizar el proceso de movimiento de los módulos, y las otras dos para marcar el último módulo que pasa por encima del líder y avisarle de que es el encargado de cerrar la tira.

La regla encargada de finalizar el proceso de movimiento de los módulos que pasan por encima del líder se ilustra en la Figura 23.

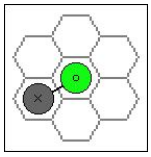

FIGURA 23. Regla de incorporación a la tira. En gris la tira.

Por otro lado, las reglas encargadas de marcar el último módulo y de cerrar la tira tienen la estructura que se puede ver en la Figura 24.

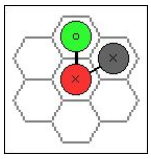

FIGURA 24. Regla de marcaje del último módulo. En rojo el líder.

8.2. Descripción de las reglas

Para describir las reglas que se han utilizado en esta fase, explicaremos detalladamente las partes de cada una de ellas, pues en este caso todas ellas son lo suficientemente distintas entre si como para merecer la pena el explicarlas todas.

Empezamos describiendo la regla que se aplica a los módulos al llegar al final de la tira y que les comunica que deben dejar de moverse mediante un cambio de estado:

Complete Strip

8020

Sfw__* !(!T-1,0, ****M !T-1,0, pfc_E)

Spfc_E
C000 + 0000 0000
C001 + 0000 0000
C002 + 0000 0000
C003 + 0000 0000
C004 + 0000 0000

Esta regla tiene más prioridad que las reglas de movimiento para evitar que el módulo continúe moviéndose una vez que ha llegado a su destino.

La precondición de la regla se divide en varias condiciones. Todas ellas deben satisfacerse para que la acción se ejecute:

- **Sfw__*:** Afecta a módulos en modo de espera.
- **!(!T-1,0, ****M !T-1,0, pfc_E):** Afecta a quellos módulos que tiene a su suroeste al líder o otro módulo perteneciente a la tira.

La acciones a realizar en caso de satisfacer todas las condiciones son las propias de un módulo termina su fase de movimiento:

- **Spfc_E:** Cambia su estado al de módulo perteneciente a la tira.
- **C000 + 0000 0000 C001 + 0000 0000 C002 + 0000 0000 C003 + 0000 0000 C004 + 0000 0000:** Elimina los valores que hubiera sobre estos contadores porque ya no son significativos.

La regla de marcar la última hoja activa se aplica al líder cuando el último módulo activo se encuentra al norte suyo:

Send Mark Leaf Strip

8020

Spfc_M A10000* T0,1, fw__A W0,1, C024 0000

M1 LStrp

Esta regla también tiene más prioridad que las reglas de movimiento para evitar que el último módulo se mueva antes de poder gestionar la marca que le envía el líder.

La precondición de la regla se divide de nuevo en varias condiciones que deben satisfacerse para que la acción se ejecute:

- **Spfc_M:** Afecta al líder.
- **A10000*:** Afecta a los módulos que están conectados por el norte y por ningún otro sitio aparte del noreste.
- **T0,1, fw__A:** Afecta a los módulos cuyo vecino del norte este activo y en espera.
- **W0,1, C024 0000:** Afecta a los módulos cuyo vecino del norte no haya sido marcado aún.

La acción **M1 LStrp** simplemente consiste en enviar un mensaje al vecino del norte para que este sepa que es el lider de la tira.

La regla de recibir la marca de líder de la tira se aplica a la última hoja que pasa por el norte del líder:

Receive Mark Leaf Strip

8020

Sfw*** A000100 T0,-1, pfc_M M4 LStrp

C024 + 0000 0001

Esta regla, al igual que la anterior, tiene más prioridad que las reglas de movimiento para evitar que el módulo se mueva antes de poder gestionar la marca que le envía el líder.

La precondición de la regla se divide de nuevo en varias condiciones que deben satisfacerse para que la acción se ejecute:

- **Sfw*****: Afecta sólo a módulos activos.
- **A000100**: Afecta a los módulos que esten conectados únicamente por el sur.
- **T0,-1, pfc**: Afecta a los módulos cuyo vecino del sur es el líder.
- **M4 LStrp**: Afecta a los módulos que reciben por el sur un mensaje de lider de tira.

La acción **C024 + 0000 0001** marca al módulo como líder de tira poniendo en el contador que lo identifica un 1.

8.3. Juegos de pruebas

Solo existe una prueba posible que hacer con este tipo de reglas, y consiste en comprobar que la tira se cierra correctamente en cualquier configuración que usemos. Por lo tanto, cualquier otro ejemplo o juego de pruebas utilizado para cualquier otro propósito sirve para comprobar de paso este tipo de reglas.

Capítulo 5
Simulador hexagonal

En este capítulo se descubre del proceso de construcción del simulador, que nos ha ocupado durante la primera mitad del trabajo. Recordamos que nuestro primer objetivo era crear un simulador capaz de reproducir la actuación de un sistema de robots modulares organizados de forma distribuida y localizados en un retículo o escenario hexagonal.

El simulador que hemos implementado permite al usuario definir conjuntos de módulos y de reglas, y también de ejecutar las reglas definidas sobre el conjunto de módulos de forma distribuida así como de mostrar el resultado obtenido al aplicarlas. Además, el simulador consta de una parte gráfica que muestra el proceso de aplicación de las reglas al conjunto de módulos para nuestro caso concreto aplicado a un universo de retículo hexagonal.

Comentaremos primero el proceso de implementación de este simulador, para después referirnos a la interfaz de la que dispone el programa para saber cómo puede ser usado. Además, al final de este documento, adjuntaremos el manual de usuario del programa.

Cabe destacar que ya existían simuladores similares de algoritmos distribuidos para sistemas robóticos modulares pero que trabajaban sobre otro tipo de retículos, o que eran de carácter general. Este simulador se especializa en trabajar sobre el retículo hexagonal de forma mucho más específica.

Nuestro simulador se trata de una modificación del simulador realizado por Reinhard Wallner que trabajaba sobre un retículo cuadrado. Así pues no se trata de un programa implementado desde cero, si no de la adapatación de un programa ya existente. Se han modificado cada una de las partes que necesitaban un cambio para poder utilizar este simulador en retículos hexagonales. De este hecho se habla más detalladamente en el siguiente apartado.

1. Implementación

Como ya hemos dicho, nos hemos decantado por modificar un programa ya existente en lugar de por crear uno nuevo. Esto es debido a que consideramos que el programa

realizado por Reinhard Wallner tenía los mismos objetivos que los que nostros nos habíamos marcado en este trabajo y era un producto sólidamente construido.

El programa que hemos utilizado para ser adaptado es capaz de reproducir la actuación de un sistema de robots modulares organizados de forma distribuida pero localizados en un retículo cuadrado, por lo que sólo necesitábamos cambiar todo lo referente al retículo sobre la que trabaja el simulador. Esto implica cambiar una serie de clases del programa, pero permite dejar otras intactas que no interfieren directamente en el retículo que queramos utilizar. Es por eso que hemos decidido modificar este programa en lugar de empezar uno desde cero que contuviera funcionalidades que ya estaban implementadas en este otro.

Durante la adaptación del programa, hemos realizado cambios en una serie de clases. No vamos a listalasr todas, pues ello requeriría una descripción de alto nivel del programa. En lugar de eso, vamos a enumerar una serie de modificaciones que se hemos realizado a causa del cambio de retículo del simulador. Cada una de estas modificaciones se compone de cambios en varias clases y en múltiples funciones, pero lo importante de cada modificación no son los cambios individuales en el código sino el motivo por el que se han tenido que producir cada uno de los cambios.

Antes de listar los cambios realizados, debemos aclarar que esta lista no pretende explicar el funcionamiento del simulador modificado, ni tampoco mostrar cada pequeña diferencia de las reglas o los módulos que hay entre esta nueva versión y la versión del simulador cuadrado. Para esas cosas disponemos del manual de usuario del simulador, que se encuentra adjunto al final de este trabajo.

- **Cambios en la visualización del universo**: Se ha adaptado la visualización principal del universo (en el panel principal), donde se muestran los agentes sobre el retículo que ahora es hexagonal.

- **Nuevas opciones de color**: Se han añadido opciones de color para modificar el color de la nueva rejilla hexagonal y de las posiciones de la rejilla externas al universo definido.

- **Opciones de visualización de los módulos**: Se han añadido opciones para que el usuario pueda elegir la forma en que los módulos se visualizan en el panel principal. Estos pueden mostrarse con forma de hexágono o con formas circulares diversas.

- **Cambios en la naturaleza de los módulos**: Se han modificado ciertos aspectos de la naturaleza de los módulos, como por ejemplo el número de vecinos o posiciones adyacentes, que cambia de 4 a 6 al pasar de un retículo cuadrado a un retículo hexagonal.

- **Cambios en la definición de los módulos**: Se ha modificado el formato de definición de los módulos en el panel de definiciones. Los cambios en la naturaleza de los módulos implican cambios necesarios a la hora de definir unívocamente el estado inicial de estos.

- **Cambios en las precondiciones de las reglas**: Algunas precondiciones que tenian como parámetros ciertos aspectos de un módulo que han cambiado al cambiar el retículo (como por ejemplo el número de vecinos) han sido modificadas para que la nueva definición sea coherente con el nuevo entorno.

- **Cambios en las acciones de las reglas**: Del mismo modo que para las precondiciones, ciertas acciones tenían como parámetros aspectos de los módulos que han cambiado, y estas también han sido modificadas para poder trabajar sobre el nuevo retículo.

Después de listar los cambios más significativos que se han producido en el simulador cuadrado para adaptarlo al simulador de retículo hexagonal, vamos a listar algunos aspectos importantes del programa que no han variado, para dar una idea general de por qué optamos por modificar este programa en vez de realizar uno nuevo.

- **Visualización general y disposición de las opciones**: La visualización general de la aplicación no ha cambiado. Si bien el panel general muestra el nuevo retículo, la organización de las opciones y del resto de paneles es la misma que la de la aplicación original. Asi pues los paneles de definición y de información no muestran cambios significativos.

- **Aplicación de las reglas de forma distribuida**: La forma en la que los módulos definidios aplican el conjunto de reglas no ha cambiado. Aunque ahora las reglas puedan ser distintas, la forma en que estas son aplicadas por cada módulo de manera independiente no ha variado.

- **Opciones de avance del algoritmo**: Las opciones de aplicación de las reglas definidas sobre el conjunto de módulos, como por ejemplo “start”, “next step”, “run forever”, “reset”, etc. no han cambiado.

- **Ficheros .log**: La actualización de los ficheros .log así como los resultados que estos muestran son los mismos que en la aplicación original.

- **Menús**: El sistema de menús así como las opciones que hay en cada menú son los mismos que había anteriormente.

- **Definición de las opciones**: El archivo de opciones, la forma de extraerlas, así como las opciones por defecto son iguales que en el simulador de retículo cuadrado.

- **Modificación de las opciones**: la interfaz de las opciones, el lugar en el que estaban las opciones antiguas, y la forma de modificar las opciones no han cambiado, excepto por las opciones nuevas que han sido añadidas.

Llegados a este punto, hemos listado lo que consideramos que han sido los principales cambios en la adaptación del simulador, así como las partes principales de este que no han necesitado ninguna modificación para que el lector pueda hacerse una idea del trabajo realizado durante la primera parte del proyecto.

2. Interficie

Una vez hemos listados tanto los cambios más importantes realizados en la implementación del simulador, como algunas de las partes del simulador que no han sufrido cambios, es un buen momento para ver qué aspecto tiene la nueva versión del simulador.

Nos disponemos a mostrar una serie de vistas (las que consideramos que son más representativas) de los principales paneles y opciones del simulador para enseñar el programa que hemos implementado durante la primer parte del trabajo, que además es el entorno que hemos utilizado durante la segunda parte.

Debemos notar de nuevo que no tenemos intención de mostrar todas las vistas o de explicar cada uno de los detalles de cada vista. Para disponer de todos los detalles de la interficie del simulador así como para saber dónde se encuentra cada una de las opciones que tiene el programa, el lector habrá de dirigirse al manual de usuario, que se encuentra al final de este trabajo.

La primera vista importante es la del panel al que llamamos panel principal y que se puede observar en la Figura 1. En este panel se encuentra dibujado el universo en su estado actual, es decir, un entorno en el que podemos navegar para visualizar las posiciones en las que puede haber agentes, así como los agentes definidos en su correspondiente posición. Además, en este panel podemos encontrar las opciones de ejecución de las reglas sobre el conjunto de módulos definido, como pueden ser las opciones “start” (activar la ejecución de reglas), “next step” (ejecutar sobre cada módulo el conjunto de reglas una sola vez), “run forever” (ejecutar la acción anterior hasta que lo paremos manualmente o hasta que ya no se pueda ejecutar ninguna regla más), “reset” (volver al estado inicial), etc.

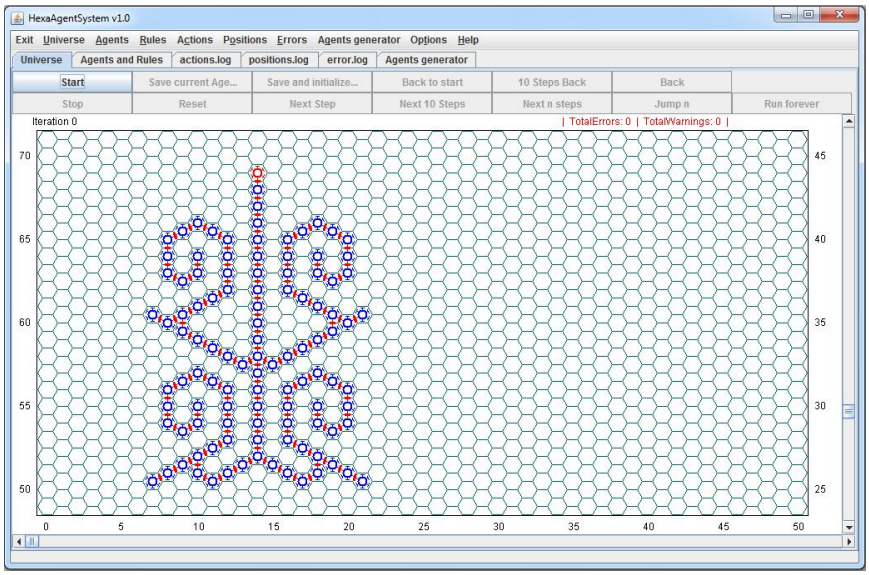

Figura 1. Vista principal del simulador hexagonal.

En la Figura 1 podemos observar una configuración de módulos en una zona del interior del universo. Podemos mover la cámara para ver otras zonas del universo hasta llegar a los extremos de este según su definición. Observamos que no se ha ejecutado ninguna regla de momento, por lo que podemos decir que la configuración que se observa en el panel principal es la configuración inicial definida en el programa.

La segunda vista es la del panel en la que se definen el conjunto de agentes y de reglas y que se puede observar en la Figura 2. Este panel esta dividido en dos secciones en las que podemos escribir, en una la definicón del universo y de los módulos, y en la otra las reglas que los módulos pueden ejecutar. En cada una de las secciones podemos encontrar opciones para modificar, abrir, guardar, o descartar los cambios realizados sobre el fichero en el que se guardan las configuraciones.

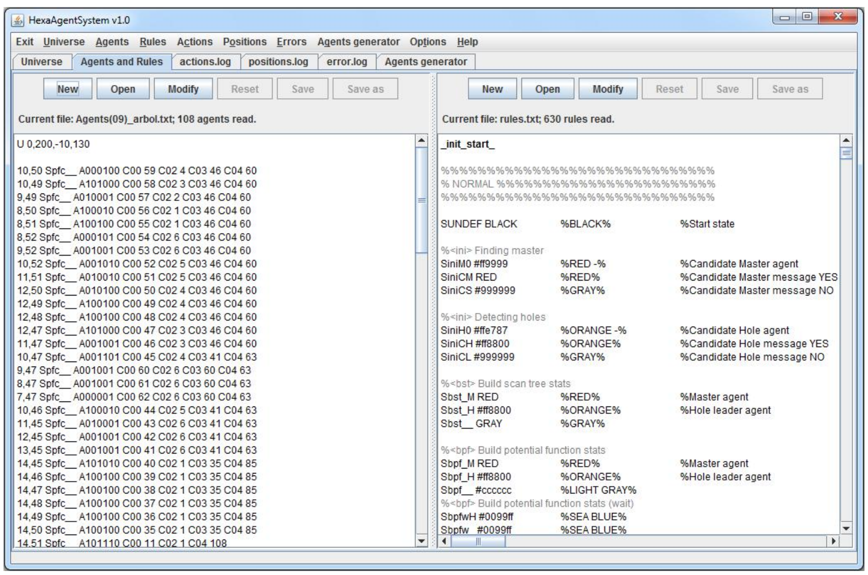

FIGURA 2. Vista del panel de agentes y reglas del simulador hexagonal.

Para entender el contenido de lo que se observa en ambas secciones de este panel conviene consultar el manual de usuario de la aplicación, que no solo explica el formato de definición de agentes y reglas, sino que además nos dice qué función exacta reproduce cada uno de los botones que se muestran en la imagen.

La tercera vista que vamos a mostrar es la de los logs del programa. El simulador cuenta con multiples logs para saber en todo momento cómo transcurre la ejecución de las reglas, y si se han producido o no errores en alguna de las iteraciones.

En concreto la Figura 3 muestra el log de posiciones que guarda para cada iteración de las que se han ejecutado, en qué posición exacta del universo se encontraba cada agente, así como información sobre este como el valor de sus contadores o el estado de sus agarres. En la vista de la Figura 3 hemos utilizado una configuración con suficientes agentes para que se pueda observar un poco el formato que tiene este

log. No obstante solo se pueden ver las posiciones que tiene algunos módulos en la iteración 0, es decir, su posición inicial por definición.

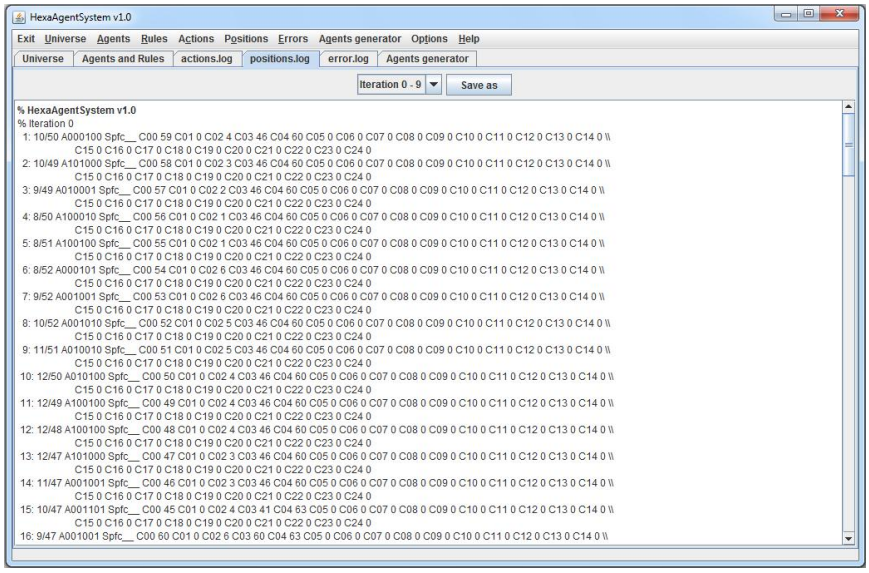

FIGURA 3. Vista del fichero de posiciones del simulador hexagonal.

La última vista que queremos mostrar es la de las opciones del programa y se puede ver en la Figura 4. Para acceder al panel de opciones hay que hacerlo a través del menú "Options" y hacer clic sobre la opción con el mismo nombre. Las opciones que podemos modificar a través de este panel son muchas y lo mejor para conocerlas todas es navegar a través de sus pestañas o consultar el manual de usuario.

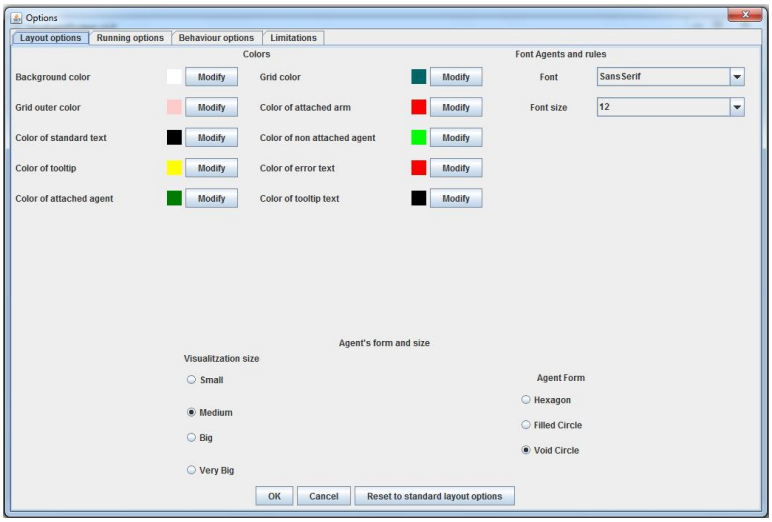

FIGURA 4. Vista del panel de opciones del simulador hexagonal.

3. Acerca del simulador

El proceso de adaptación del simulador ha sido más interesante de lo que parecía en un principio. Lejos de convertirse en un trabajo tedioso de búsqueda infructuosa a través del código hasta hallar las partes que necesitaban un cambio, he podido observar que la metodología con la que estaba implementada la aplicación original era impecable y facilitaba enormemente el entendimiento del código. Me ha parecido relevante mencionar la facilidad con la que se puede entender el código de la aplicación, y considero que debo animar a cualquier persona que esté interesada en estudiarlo, o incluso en ampliarlo.

Es especialmente gratificante después de realizar un trabajo de implementación el hecho de saber que la aplicación producida va a ser utilizada por otras personas para facilitar el desarrollo de sus trabajos o investigaciones. Me produciría una desiluisión saber que una aplicación en la que he estado trabajando tanto tiempo no fuera a ser usada nunca más, pero me complace saber que esto no sucederá.

Por último, he de mencionar que el código de la aplicación permanecerá, como no podía ser de otra forma, abierto a cualquier persona que desee estudiarlo, y que estaré especialmente complacido de saber que alguien desee ampliarlo en un futuro.

Capítulo 6
Conclusiones

Durante la realización de este trabajo se han satisfecho todos los objetivos que se deseaba alcanzar al hacer un trabajo de final de grado. Uno de ellos era no hacer un trabajo puramente teórico, sino que tuviera parte de implementación y parte aplicación matemática. Este objetivo ha sido satisfecho plenamente al tener la suerte de poder trabajar en un proyecto de simulacion de sistemas modulares distribuidos.

Entre las cosas aprendidas están, aparte de avanzadas técnicas de programación y conceptos interesantes relacionados con los algoritmos distribuidos, el uso de las matemáticas en procesos de simulación de estas características, y la combinatoria detrás de toda la casuística de combinaciones de conjuntos de agentes en un retículo hexagonal. He aprendido a como detectar y diferenciar los distintos casos que se pueden dar cuando intentas definir un proceso global en una teselación del plano hexagonal, y debo decir que la tarea me ha parecido de lo más interesante.

En cuanto al problema concreto resuelto durante la segunda parte del trabajo, la reconfiguración de un conjunto de módulos a una forma canónica, ha sido ua tarea interesante en muchos sentidos. Primero de todo, porque considero que es un problema de actualidad con posibles aplicaciones en el futuro próximo. Ya sean módulos cuadrados, cúbicos o hexagonales, la reconfiguración de un conjunto de pequeños robots puede dar mucho juego. Segundo y quizás más importante para el trabajo que nos ocupa, el problema ha conseguido ayudarme a abrir mi mente en el sentido de ser capaz de explorar todas las posibilidades, y de detectar todos los casos posibles que se pueden dar en este tipo de problemas combinatorios, para más adelante, resolverlos.

Cuando uno realiza un trabajo de estas características, y dedica tantas horas a hacer algo que considera útil, se complace de saber que su esfuerzo va a servir para que otras personas puedan continuar su camino. Por eso, a uno le resulta un aliciente al implementar un simulador como este, el hecho de saber que incluso después de haber acabado el trabajo otras personas van a seguir usandolo, y que les va a facilitar el estudio de este y de otros problemas que se puedan simular con un sistema distribuido en un retículo hexagonal. Por eso estoy especialmente satisfecho de saber que el programa va a ser usado en el departamento y que el esfuerzo realizado no muere cuando acabe este trabajo.

También desearía decir que no habría sido posible realizar desde cero un simulador tan completo en tan poco tiempo, y que fue realmente una buena decisión el coger un simulador parecido, con el permiso de la persona que lo hizo, y adaptarlo al caso que requería este trabajo. La verdad es que fue una gozada tener que modificar un programa que desde el primer momento se mostró sólido y perfectamente acabado, y por ello deseo expresar mi respeto a la persona que lo hizo, Reinhard Wallner.

Por último, desearía dar las gracias también a mi tutora en el trabajo, Vera Sacristán, así como a Ángel Rodríguez, que han trabajado conmigo en el problema de reconfiguración y que me ha ayudado siempre que lo he necesitado.

Capítulo 7
Compact User Guide of the hexagonal Agent System

1. Introduction

This document describes the functionalities of the simulation program of distributed algorithms for modular robotic systems in hexagonal lattices. If refers to the HexaAgentSystem version 1.0 program. It is a modification of an existing simulation program for modular robotic systems in square lattices named AgentSystem, therefore some of its functionalities are very similar.

Thanks are given to Birgit Vogtenhuber, Reinhard Wallner and Óscar Rodriguez for the square lattice Agent System simulator as well as for the structure of this user guide

2. Program flow

This section is dedicated to explain how the system evaluates the precondition and the priority and how it executes the action of each agent. Since the system supports priorities of the rules, the evaluation model is more complex than it would be without the usage of priorities. But priorities provide a helpful extension of the rules to enable advanced rule sets with simple rules.

The program starts by reading the initial set of agents as well as the set of rules. In every calculation step, the following operations are performed (in the order listed here; alternatively sending messages can be done before performing actions).

(1) Check and get valid rules (each step for all agents)
  (a) check all rules whether they would apply (ignoring priorities) and store valid rules sorted by priority in descending order

(b) store the highest priority of valid rules as current priority (if > 0) and set the agents priority to `PRIORITY_OPEN`
(c) for all agents sorted by priority in descending order (if priority is `PRIORITY_OPEN`):
do while not all agents have `PRIORITY_FIXED` :
- fetch current rules to current priority
- for all rules: check priority-conditions
  - if they are fulfilled, set priority to `PRIORITY_FIXED`
  - if one of them is not fulfilled, remove this rule from the specified agent and if the agents rule list is empty, reduce the current priority to the highest priority of the remaining rules
  - if a circular dependency between rules on different agents is detected, remove all related rules

(d) remove all rules with priority lower than the priority of the agent

(2) For all agents, for all previously stored rules for the agent: perform from the actions (if they are part of the action):
(a) detach
(b) compute the attachment component
(c) move the attachment component (including collision detection test)
(d) update attachments
(e) update the state
(f) swap agents

(3) Send and deliver messages
(a) for all agents, for all previously stored rules for the agent:
- do all calculations (in the order they are listed in the rule) and send numerical messages to the post-office
- send all text messages to the post-office

(b) for all messages at the post office: deliver them to their recipients.

3. Program inputs

The program `HexaAgentSystem.jar` uses three text files as input.
The file `files/config.txt` is a mandatory file. If it doesn't exist or an error on reading this file occurs, an error display is shown instead of the main window. A similar message appears if any file in the directory `img` does not exist.

The files `files/agents.txt` and `files/rules.txt` are the standard input files for the initial setting. If one of these files doesn't exist or doesn't match the specifications respectively described in sections 3.1 and 3.2 an error dialog occurs at the beginning. The dialog can be closed and necessary files can be chosen (see 4.2).

: Syntax to launch the program:

```
java [parameters] HexaAgentSystem.jar
```

The program can be alternatively started with commandline parameters. If input files are chosen with commandline parameters (arguments `-a` or `-r`), these files must exist instead of the files listed above. Available parameters are:

- `-a <agents_file>` this file is used instead of `files/agents.txt`
- `-r <rules_file>` this file is used instead of `files/rules.txt`
- `-n <number_of_steps>` performs after start the given number of steps with delay (see sections 3.3.2 and 4.1).
- `-nx <number_of_steps>` same as option `n` but additionally the program is stopped after performing the given number of steps.
- `-j <number_of_steps>` performs after start the given number of steps in jump mode (see 4.1).
- `-jx <number_of_steps>` same as option `j` but additionally the program is stopped after jumping the given number of steps.
- `-e` performs after start an endless run with delay (see sections 3.3.2 and 4.1).
- `-ex` same as option `e` but additionally the program is stopped if no further rule can be performed. Note that the endless run only stops if none of the preconditions of any rule is fulfilled. It doesn't stop if any rule tries to perform a step, but an error on performing occurs (see 4.1).
- `-s <agentsfile>` stores automatically the resulting agents configuration after performing multiple steps. Hence this argument is just allowed in combination with the `-nx, -jx` or `-ex` argument.
- `-h` help to commandline arguments

The file `files/config.txt` contains the options of the program (see 3.3). The file `files/agents.txt` contains the definition of the initial setting, meaning the initial positions, states, attachments, and counters of all agents (see 3.1). The definition of what an (arbitrary) agent will do is contained in the file `files/rules.txt` (see 3.2). In all files, one-line-comments can be written by pre-pending the percentage-symbol (`%`) to the comment (meaning everything in the line after `%` is ignored). White spaces as well as empty lines, are ignored. Thus these two symbols may not be used as text. Further, for any kind of pattern matching, the asterisk (`*`) matches every character.

The error handling when starting the program is done in different ways because the program on the one hand can be used by manually starting and navigating it, or on the other hand it can be used by batch processing. To enable both cases with a maximum of usability, the error cases are handled as follows:

(1) If the arguments contain syntax errors or irregular combinations of arguments are detected, a command line help with the information about the usage of the program is shown. The GUI does not start in that case.
(2) If the program is started without any argument, but there exist errors in the agents or the rules files, then a dialog box appears after starting the graphical user interface with an appropriate message.
(3) If the program is started with the arguments `-a` and/or `-r`, but no other parameters, and errors are found in the agents or in the rules files, the same behavior as in Case 2 appears.

(4) If the program is started with the arguments `-a` and/or `-r`, but no other parameters, and any of the input files is not found, the GUI starts and an error message about this fact is displayed in a dialog box.
(5) If any of the command line arguments `-nx`, `-jx` or `-ex` is used and any of the agents or the rules files is not found, then an error message is shown in the command line to ensure that a batch routine can proceed its work without interruption.
(6) If the same arguments as in Case 5 are used and all files are found, but an error is detected in the agents or the rules file, then the program starts the GUI and shows the error to inform the user.

Moreover note that the usage of one of the arguments `-nx, -jx, -ex` has a side effect. When starting the program with one of those parameters, the `STOP` button is enabled to give the user the possibility to stop the process. But if the `STOP` button is pressed, then the program does not stop processing until the given number of steps are performed!

3.1. Definition of the initial setting. The initial setting is stored in the file `files/agents.txt`. Each line of the file defines one agent by its initial (global) coordinates (mandatory) plus (optional) its state, its attachments and initializations for (some of) its counters. Optionally it is possible to change the size of the universe in the agents file with the parameter `U`.

Initial (global) position: `x,y`
The initial position is written as integer x-, y-coordinates, separated by a comma (see 3.3.4 for the range of coordinates).

State: `S_____`
The state of an agent consists of exactly 5 characters, written with a leading S.
Default (if no argument given): `S*****` (no information).

Attachments: `A______`
The attachments of an agent are written as A followed by 6 booleans (0 for not attached, 1 for attached), in the order North, North-West, South-West, South, South-East, North-East, that is counterclockwise starting from North
Default (if no argument given): `A000000` (nothing attached).

Counters: `C__ _____`
Each agent has 25 integer counters, `C00`,...,`C24`, which can be set to any 16-bit integer between -32767 and 32767 (-32768 is defined as no-number).
Default (if no argument given): any not explicitly initialized counter is set to zero.

Limitation of the universe size: `U minX,maxX,minY,maxY`
This parameter must be positioned at the beginning of the file. If this parameter is used the option `coordinates_range` of the file `files/config.txt` is overridden for this agents configuration (see 3.3.4)!

Small example:

```
U -5,20,-10,20 % optional line
0,0 A000001 S1ST-- C0 -5 C1 17
1,0 A001000 S2ND--
2,0 SALONE
5,6
5,7
6,6 C0 1
7,3 C1 -234   C5 43   C7 999
```

3.2. Definition of the rules. The definition of what the agents will do is stored in the file `files/rules.txt`. Each rule definition consists of 4 lines:

(1) the name of the rule
(2) the priority of the rule
(3) the precondition
(4) the action

The name is a nonempty string. The priority is either a positive integer (at least 1 and at most 32767), where higher priority 'wins' lower priority, or it is written with the sign `$`. In the latter case the rule is a non-priority rule. The precondition defines whether or not an agent may apply the rule. Finally, the action defines what has to be performed, if an agent applies a rule.

3.2.1. **The precondition.**

An agent can apply a rule if it fulfills all the statements included in the precondition of the rule. The precondition may consist of the following parts, where each part is optional. In total, the precondition of a rule cannot be empty. Additionally each "sub-condition.of the precondition can be negated (see the end of this section).

Neighbours: `N______`
The definition of the direct neighbours (North, North-West, South-West, South, South-East, North-East). For each of them
- `:0` denotes empty (no agent),
- `:1` denotes full (an agent), and
- `:*` denotes don't care.

For example `N01**1*` denotes that there must be an empty field to the North of the agent, and the North-West and South-East fields need to be full.

Empty field: `Edx,dy`, `EC__,dy`, `Edx,C__`, `EC__,C__`
An empty field requirement. Written as an E, followed by the relative coordinates of the field that needs to be empty, separated by a comma. Alternatively instead of each value `dx` or `dy` a name of any counter can be inserted, where a counter starts with a `C`, followed by the two digit number of the counter. If a counter name is written, the value of the counter of the current agent is inserted as this coordinate. Of course it is possible to mix numbers and counters. Note that the usage of counter values for coordinates may lead to relative

coordinates outside the universe. In that case a warning is shown but the rule is performed.

Filled field: `Fdx,dy`

A full field requirement. The restrictions and the syntax are the same as in the "Empty fieldçondition.

Neighboring priorities: `P_____`

The priority of (the applied rule/s) of the direct neighboring positions (North, North-West, South-West, South, South-East, North-East). For each of them:

- `<` denotes that the priority of the corresponding agent needs to be strictly smaller,
- `=` denotes smaller or equal, and
- `*` denotes don't care.

Note that the priority of an empty field is always zero.

Remote smaller priority: `Ldx,dy`

A strictly smaller priority field requirement. The L is followed by the relative coordinates of the agents that is required to have smaller priority, separated by a comma. The usage for counters is the same as in the "Empty field" condition.

Remote smaller or equal priority: `Qdx,dy`

A less or equal priority field requirement. Syntax and usage is analogous to those of the "Remote smaller priorityçondition.

Attachments: `A_____`

The attachment states to the direct neighbors (North, North-West, South-West, South, South-East, North-East), where

- `0` denotes not attached,
- `1` denotes attached, and
- `*` denotes don't care.

For example `A0**1**` denotes that the agent must not be attached North and needs to be attached South (implying that an agent to the South must exist).

State: `S_____`

Every agent has a state, consisting of exactly 5 characters. This state can be required to match a simple pattern, where an asterisk matches any character. For example `SA***Z` denotes that the state of the agent must start with `A` and end with `Z`.

State of a remote agent: `Tdx,dy,_____`

This option is a combination of the "Filled field.[a]nd the "State"preconditions. Written as a T, followed by the relative coordinates of the field that is required to be full, and ended by the state that the remote agent must have. The usage of counters is the same as in the "Empty field" condition.

Incoming text messages:

`M*_____`, `M1_____`, `M2_____`, `M3_____`, `M4_____`, `M5_____`, `M6_____`

Every agent has six text messages from its direct neighbours (`*`=any, `1`=North, `2`=North-West, `3`=South-West, `4`=South, `5`=South-East, `6`=North-East), consisting of exactly 5 characters. Any of these messages can be required to match a pattern, where the asterisk matches any character but at most four asterisks are allowed.

Numeric comparisons on counters and incoming numeric messages: `<(-)____ (-)____`, `>(-)____ (-)____`, `=(-)____ (-)____`

Every agent has 25 counters `C00`...`C24`, and additionally $6 * 8 = 48$ numeric messages from its direct neighbors:

- Messages: `#101`, ..., `#108` (from North), `#201`, ..., `#208` (from North-West), `#301`, ..., `#308` (from South-West), `#401`, ..., `#408` (from South), `#501`, ..., `#508` (from South-East), `#601`, ..., `#608` (from North-East)
- Use `#*01`, `#*02`, `#*03`, `#*04`, `#*05`, `#*06`, `#*07`, `#*08` to check whether a numeric message is received from any direction at the specified channel.

Any of these numeric values can be required to fulfill a comparison with respect to any other such value or to any four digit number, where a negative number contains additionally a leading `-`.

Note that messages at different channels are independently from each other.

Remote numeric comparisons: `Vdx,dy,C__ (-)____`, `Wdx,dy,C__ (-)____`

These options compare the first value with the second value:

`V`: the first value is strictly smaller than the second;

`W`: the first value is smaller or equal than the second.

The first numeric value is a counter from another agent at relative coordinates `dx`, `dy`. It requires the agent to exist. The second numeric value can be a counter, a numeric message from a neighbor or any four digit number, where a negative number contains additionally a leading `-`. Recall that every agent has 25 counters and $6 * 8 = 48$ numeric messages from its direct neighbors (enumerated in the previous "Numeric comparisonçondition).

Negation: `!`

A negation can be applied to any part of the precondition. The negation means that the result of a "sub-condition"will be negated.

Parenthesis: `( )`

To enable grouping of conditions as well as offering a disjunction between different conditions, parenthesis can be set around a group of conditions. It is also possible to interleave parenthesis (example: `!( N**1*0* !(A101*00 L-3,2) )`).

3.2.2. **The action.**

The action of a rule defines what happens when an agent applies the rule. It may consist of the following parts, were each part is optional (but in total the action cannot be empty).

Position change: `Pdx,dy`

Written as a P, followed by the relative coordinates of the field to which the agent moves (separated by a comma). The usage for counters is the same as in the "Empty field" condition.

Attachments: `A______`

Attachment behavior for the six directions North, North-West, South-West, South, South-East, North-East (in this order), with the following possibilities for each direction:

`:` `0` for detaching (if was attached) before moving, and staying detached afterwards

- `:1` for detaching (if was attached) before moving, and attaching afterwards (if possible)
- `:*` for detaching (if was attached) before moving, and attaching afterwards if was attached before (and possible)
- `:+` (only allowed if the agent is attached at the beginning) staying attached all the time. If the agent moves, every (directly or indirectly) attached agent is moved in the same way.

Default (if no argument given): `A******` (try attaching as before in all directions).

State: `S_____`

New state of the agent, consisting of exactly 5 characters. An asterisk denotes that the according character remains unchanged.

Default (if no argument given): `S*****` (no change).

Outgoing text messages:

`M*_____`, `M1_____`, `M2_____`, `M3_____`, `M4_____`, `M5_____`, `M6_____`

Messages that are sent to neighbours (`*`=any, `1`=North, `2`=North-West, `3`=South-West, `4`=South, `5`=South-East, `6`=North-East), consisting of exactly 5 characters. At most four asterisks are allowed (for example `M3RUN**` is allowed, but `M3*****` is not).

Assign calculation results to counters and outgoing numerical messages:

`C___ _ ____ ____`, `#___ _ ____ ____`
`C___ _ dx,dy,C___ ____`, `#___ _ dx,dy,C___ ____`
`C___ _ C__,C__,C___ ____`, `#___ _ C__,C__,C___ ____`

Every calculation action starts with a position to write the result to (counter or outgoing message), followed by the operation to be performed, and two (readable) values on which the operation is performed.

Possible operations are `+` (add), `-` (subtract), `*` (multiply), `/` (divide), `M` (modulo), `A` (maximum), and `I` (minimum).

As values for an operation, either four-digit-numbers, or internal counters, or one remote counter, or incoming numerical messages can be used. Every agent has 25 counters `C00` ... `C24` which can be read and written. Further, it has 32 (incoming) numeric messages from its direct neighbors, which it can only read, and 32 numeric (outgoing) messages to its direct neighbors, which it can only write.

The remote counter is defined by first indicating the coordinates (`dx,dy` or `C__,C__`) of the module and then the counter to be used. Note that the coordinates are restricted to three digits (`-999` to `999`), and that it is required that there exists an agent at the indicated location.

Available message senders/recievers:

- Messages `#101`, ..., `#108` (from North), `#201`, ..., `#208` (from North-West), `#301`, ..., `#308` (from South-West), `#401`, ..., `#408` (from South), `#501`, ..., `#508` (from South-East), `#601`, ..., `#608` (from North-East)
- Use `#*01`, `#*02`, `#*03`, `#*04`, `#*05`, `#*06`, `#*07`, `#*08` to send a numeric message to all neighbors at the specified channel.

Note that a calculation with undefined result (for example division or modulo by `0`) leads to an error. If such an error occurs no value will be sent to the receiver of the result.

Swap: `X1`, ..., `X6`
Written as a `X`, followed by the desired swap neighbor (remember that `1`=North, `2`=North-West, `3`=South-West, `4`=South, `5`=South-East, `6`=North-East)

3.2.3. **Include rules.**

To enable combinations of different rule sets for one agent configuration, it is possible to include other rule files. The syntax for an include is

```
include:<file_path>:<Priority_Offset>
```

where include statements must be added at the beginning of the file. If just a file name is given the file will be searched in the `files` directory of the program. If a relative path definition is given in the include statement, the path separator sign must be chosen from the user according to the operating system. Note that spaces are not allowed in the path according to 3. It is possible to insert as many include statements as desired and it is also possible to cascade include statements. If include statements are cascaded priority offsets will be accumulated.

If an error in any of the included files occurs, an error message is shown in the specified line of the main rules file. Additionally an error message occurs if the include statement is behind the init block (see 3.2.4) or any defined rule (see 4.2.2).

For example `include:foreign_rules.txt:100` means that all rules from the file `foreign_rules.txt` are included in the ruleset, where the priority of each rule defined in `foreign_rules.txt` will be incremented by `100`.

Note that the usage of include statements can simplify the definition of complex rulesets. But this can lead to some problems, for example the usage of the same state name in different files or the usage of same counters in different files, what definitively would lead to undesired behavior.

3.2.4. **State colors.**

A helpful function for visualization is the color of agents. The standard color for attached and detached agents can be defined in the file `files/config.txt` file or alternatively in the options menu. But in the visualization it is also essential which state an agent at a specified step has. Of course it is possible to check the state in the tooltip function (see 3.3.1). But to get an overview about the state of all agents this would need a lot of time. Therefore a color can be assigned to each state of the agents.

Assigning colors to states can be made in the current rules file. In order to do so an init block is inserted in the rules file. The init block needs to be located after the include lines and before the rules. In the init block each line assigns a color to a state. Assigning a color to a state can be made in two different ways. The first possiblity is to write the name of the state followed by the name of the color, where possible colors are `BLACK`, `BLUE`, `CYAN`, `GRAY`, `MAGENTA`, `ORANGE`, `PINK`, `RED`, `YELLOW`. The names are listed additionally in the help menu (described in 4.13). An alternative option is to write the specified state followed by a `#` and the hexadecimal `RGB` color code.

The following is an example of a rules file:

```
include:foreign_rules.txt:50

_init_start_
SRIGHT CYAN
SDOWN_ #ff0000 %means the same as SDOWN_ RED
_init_end_

Go Down
100
SDOWN_ >C0000000
P0,-1 C000-C0000001 A++++++
```

Note that color names are written with capital letters and that the hexadecimal code uses lower case letters. Further in the hexadecimal code the first two digits stand for the red color, the middle two digits for the green color and the last two digits for the yellow color. If an agent gets a state where no color definition is made in the rules file, the agent gets the standard color as defined in the options (see 3.3.1). If any error on reading the state colors is found, an error message is written in the specified line (see 4.2.2).

3.3. Program options. The definition of available options is stored in the file `files/config.txt`. All options that are available can be set in the file or in the Options menu(see 4.12). The syntax of options is always the same: `[OPTION_NAME]=[OPTION_VALUE]`.

3.3.1. **Layout options.**

Available layout options are color options, font options [and the grid option].

(1) Color options
The color option value has the syntax: `[R],[G],[B]` whereas `R` means the red color value, `G` means the green color value and `B` means the blue color value. The values for `R`, `G` and `B` can be chosen in the range 0-255.
: Example:
`color_bg=255,255,255`
The following colors can be modified:
Background color: Color of the universe background.
- Option name: `color_bg`
- Standard value: `[255,255,255]`: (White)

Grid color: Color or the universe grid.
- Option name: `color_grid`
- Standard value: `[0,0,0]`: (Black)

External grid color: Color or the grid positions outside the universe.
- Option name: `color_void_grid`
- Standard value: `[255,204,204]`: (Light grey)

Color of not attached agent: Color of agents who are nowhere attached, and for attachment arms which aren't attached.
- Option name: `color_agent_not_att`
- Standard value: `[0,255,0]`: (Green)

Color of attached Agent: Color of agents who are anywhere attached.
- Option name: `color_agent_att`
- Standard value: `[0,124,0]`: (Dark Green)

Color of attached arm: Color of attachment arms which are attached.
- Option name: `color_attached_arm`
- Standard value: `[255,0,0]`: (Red)

Standard text color: Color of the iteration number in the universe.
- Option name: `color_text`
- Standard value: `[0,0,0]`: (Black)

Error text color: Color of the error message text in the universe.
- Option name: `color_error_text`
- Standard value: `[255,0,0]`: (Red)

Color of tooltip background: Color of the tooltips background.
- Option name: `color_tooltip`
- Standard value: `[255,255,0]`: (Yellow)

Color of tooltip text: Color of the tooltips text.
- Option name: `color_tooltip_text`
- Standard value: `[0,0,0]`: (Black)

(2) Grid options
Here we list the options related to the size and the shape of the agents in the grid.

Size of the grid: Determines the size of the grid and the agents in the screen.
- Option name: `agentSize`
- Available values:
  - `small`: The smallest size of all them.
  - `medium`: A slightly bigger size of the grid and agents.
  - `big`: A even bigger size of the elements of the screen.
  - `veriBig`: The biggest size.
- Standard value: `small`

Shape of the agents: Determines the shape that the agents have over the grid.
- Option name: `agentForm`
- Available values:
  - `hexagonal`: The agents have hexagonal shape.
  - `filledCircle`: Round filled shape.
  - `voidCircle`: Round shape, not filled.
- Standard value: `voidCircle`

(3) Font options
Font options are font type and font size for the agents, rules, actions, positions, and error panels.

Font type: Type of the font in the agents, rules, actions, positions and error panel.
- Option name: `current_font`

- Standard value: `[SansSerif]`.

Font size: Size of the font. Available range is from 6 to 32 dots per inch.

- Option name: `current_font_size`
- Standard value: `[12]`.

3.3.2. **Running options.**

These options are used for better usability and performance.

(1) History options
The program has a history option to provide reverse steps.
- Option name: `history`
- Available values:
  - `all`: All iterations are stored in the history. Take into account, though, that the system is limited to a maximal stored number of history entries (see 5).
  - `last20`: The last 20 iterations are stored in the history (at most 20 back steps are possible).
  - `off`: No history entries are stored. Thus no back steps are possible.
- Standard value: `all`

(2) Logging options
For details to the log files see 6.
- Option name: `logging`
- Available values:
  - `all`: All log files are written.
  - `error`: Just the error.log file is written.
  - `off`: No log files are written.
- Standard value: `all`

(3) Delay options
Delay between multiple steps (10 Steps Back, Next 10 Steps, Next n steps, Run forever; see 4).
- Option name: `timeout`
- Available values:
  - The timeout in milliseconds.
- Standard value: `500`

(4) Rules warning option
This option enables or disables the warning for duplicate rule names (see 4.2.2).
- Option name: `rules_warning`
- Available values:
  - `on`: Rules warnings are shown in rules panel.
  - `off`: Rules warnings are not shown in rules panel.
- Standard value: `on`

3.3.3. **Behavior options.**

These options are concerning the actions in each iteration.

(1) Agent options
This option takes effect on applying rules on each agent. In the standard way all rules which can be applied to one agent are performed in the order they

are listed in the rules file `files/rules.txt`. This option allows to change this in the following way: rules which don't have an action including a position change can be applied before rules which imply a position change. This option has an effect on each agent, but does not influence the global order of the rules performance: rules are always applied to the agents in the order agents are listed in `files/agents.txt`.

- Option name: `position_changes_last`
- Available values:
  - `on`: All stored rules of an agent implying a position change are performed after all rules without position changes.
  - `off`: All rules are applied in the order listed in the rules file.
- Standard value: `off`

(2) Message delivering option
In the standard way messages are sent and delivered after performing all actions (described in [**?**]). With this option this can be changed so that after checking all rules all messages are sent and delivered and after that all actions are performed.

- Option name: `perform_messages_first_move_last`
- Available values:
  - `true`: All messages are sent before performing all actions to all agents.
  - `false`: All messages are sent after performing all actions to all agents.
- Standard value: `false`

3.3.4. **Limitations.**

Limitations to the universe range can be set. Attention: This option does not have any effect if the parameter `U` is used in the agents file (see 3.1).

(1) Coordinates range
This option has the syntax: `min_X, max_X, min_Y, max_Y`, where the values must be integers.

- Option name: `coordinates_range`
- Example:
  - `coordinates_range=-5,70,-5,27`
- Standard value: `-5,70,-5,27`

4. Program window

The main window consists of a menu and a tabbed panel with five tabs. The
Short key: `'Ctrl'+'T'` allows to navigate to the next tab, and the
Short key: `'Ctrl'+'Shift'+'T'` sends to the previous tab.

4.1. Universe. This panel visualizes all read agents in the universe. In the upper side of the universe panel the iteration is shown and if an error occurs a message is shown, too. If more than one error appears in one iteration a `+` sign is added in front of the error field. Clicking on the `+` sign opens a window with all error

messages appeared in this iteration, where the first error is on the first position in the window. If more than 10 errors appears, an additional line
`< < < < Further errors occurred! For details see error.log! > > > >`
is shown in the last line of this window. Furthermore, the number of `TotalErrors` and `TotalWarnings` is shown in the upper side of the universe panel.

Several navigation buttons are available (back steps are only available if the history function is on, see 3.3.3). For multiple steps buttons (10 Steps Back, Next 10 Steps, Next n steps, Run forever) a timeout between the steps is passed (see 3.3). Alternatively for each button at least one short key is available.

Universe panel functions:

Start: Before iterations can be performed this button has to be clicked.
Short keys: `'s'`, `'S'`.

Reset: A click on this button resets the configuration to its initial values. This means that agents and rules are read once again and all history entries are removed.
Short key: `'F2'`.

Save current agents: With this button the current agents setting (positions, states, counters, attachments) can be stored in a file. Note that input messages on agents are not stored in the agents file. Hence this option can only be used in an appropriate way if no input messages are stored on any agent.
Short key: `'Ctrl'+'S'`.

Save and initialize agents: Stores the current agents setting (positions, states, counters, attachments) in a file and additionally loads the setting from the new file and defines it as the new start configuration. Note that input messages on agents are not stored in the agents file. Hence this option can only be used in an appropriate way if no input messages are stored on any agent.
Short key: `'Ctrl'+'I'`.

Back to start: A click on this button resets the universe to its initial position (Iteration0)
Short key: `'Home'`.

10 Steps Back: With this button 10 reverse steps are performed.
Short keys: `'PgUp'`, `'CursorUp'`.

Back: Just to go one step back.
Short keys: `'CursorLeft'`, `'Backspace'`.

Next Step: Go one step forward.
Short keys: `'CursorRight'`, `'Enter'`.

Next 10 Steps: Go 10 steps forward. If no further rule can be performed a message is shown.
Short keys: `'PgDown'`, `'CursorDown'`.

Next n steps: Go several steps forward (1-999).
A click on the button opens a combo box where the number of steps to move can be introduced. If no further rule can be performed a message is shown.
Short keys:

- `'Shift'+'CursorDown'`, `'Shift'+'PgDown'` opens the combo box. In the opened box the following short keys are available:

  - `'CursorUp'`, `'CursorDown'`, `'PgUp'`, `'PgDown'`, `'[0-9]'`
    With these keys the number of steps to perform can be chosen.
  - `'Backspace'` deletes introduced numbers.
  - `'Enter'` closes the combo box and performs the chosen number of steps.
  - `'Esc'`, `'Pause'` closes the combo box

Jump n: A click on this button performs a chosen number of steps without visualization and without timeout (see 3.3.2, item 3). After clicking on the button a dialog appears. The dialog has a text field to insert the desired number of steps, a button `Jump` to perform the chosen number of steps and a button `Cancel` to close the dialog without performing steps. After clicking on `Jump` the steps are performed. To update the user about the remaining steps a message `Wait n steps` is shown on the top of the universe panel (where n indicates the remaining number of steps). If no further rule can be performed a message is shown.
Short keys: `'j'`, `'J'`.

Run forever: Performs endless forward steps. If no further rule can be performed a message is shown.
Short key: `'End'`.

Stop: Stops executing steps (10 Steps Back, Next 10 Steps, Next n Steps, Run forever). Short keys: `'Esc'`, `'Pause'`.

Tooltip function: A helpful function to check the current values of each agent is the Tooltip function. A click on an agent shows a tooltip window.
The information displayed on this windows is:

always shown:: (on each agent)
: Id
: Global position
: Attachments
: Counter values
: Current priority

optionally shown: (if information is stored)
: State
: Text messages
: Numeric messages

Zoom: The size of the grid cells and the agents can be increased by pressing the
Short key: `'Ctrl'+'+'` and decreased by pressing the
Short key: `'Ctrl'+'-'`.

Speed: This function is only available with the
Short key: `'Ctrl'`.
If multiple steps are performed a timeout between steps is passed. To avoid this, the timeout can be disabled. But this can be done in another way, too: while multiple steps are performed the Short key `'Ctrl'` can be hold and the timeout while holding the key is 0.

4.2. Agents and rules. The tab consists of two text panels. On the left side the current agents file is shown and on the right side the current rules file is shown. To enable changing focus between the two text panels if editable, the mouse or

alternatively the short key: ‘Ctrl’+‘P’ can be used. If just one text panel is editable the focus grabs to that panel.

The font type and size of the agents and rules panels as well as of the actions, positions and error panel can be modified in the options menu (see 3.3.1).
Short key: ‘Ctrl’+‘+’ increments the font size.
Short key: ‘Ctrl’+‘-’ decrements the font size.

4.2.1. **Agents panel.**

The panel has on the upper side buttons to create, open and modify agents files. The text panel contains the current agents file. Lines with successfully read agents (valid initial setting) are shown in black, comment lines are shown in gray and lines with errors are shown in red. An agent definition is valid if and only if all settings in the specified lines fulfill the specification. For further information see 3.1.

Possible error messages:

- "% AGENT already exists! Ignored! occurs if an agent on this position is already stored.
- "% Position OUT_OF_RANGE! Ignored!" is shown if a position value of an agent isn't in the coordinate range of the universe (see also Sections 5 and 3.1).
- "% % Counter OUT_OF_RANGE! Ignored!" occurs if any counter value inserted is out of the specified range (see 3.1).
- "% ATTACHMENTS ERROR: not symmetrical! Attachment ignored!" is displayed if an attachment definition between two agents exists that is not symmetrical.
- The message "% ATTACHMENTS ERROR: to attach agent doesn't exist! Attachment ignored!" shows that an attachment definition refers to a non existent agent.
- "% SYNTAX ERROR OR IRREGULAR LIMITATIONS! Ignored!" occurs if a syntax error on an agent definition or the universe limitation is found, or if the universe limitation is not at the beginning of the file.

New: To create a new agents file. After clicking on the button the text panel will be cleared. Agents can be defined and saved. To interrupt the process click on Reset. To store the new setting click on Save.
Short key: ‘Ctrl’+‘N’.

Open: Opens an agents setting file. A dialog to save all log files appears before opening the file (see button Save) because when opening the agents file the log files are overwritten.
Short key: ‘Ctrl’+‘O’.

Modify: With this button the current agents setting can be modified.
Short key: ‘Ctrl’+‘M’.

Reset: If a new agents setting is created or an old one is modified, this button discards the new setting (the last read agents become active).
Short key: ‘Ctrl’+‘R’.

Save: To save an existing agents setting. If the button is clicked all log files are overwritten and the new agents setting is read immediately. No dialog appears

to store existing log files!
Short key: ‘Ctrl’+‘S’.

Save as: To save a new agents setting or a modified one. If this button is clicked all log files are overwritten and the new agents setting is read immediately. A dialog to save all log files appears before loading the new setting.
Short key: ‘Ctrl’+‘W’.

Undo: If the agents file is modified in the agents panel this function undoes the last modification. If modifications in the rules panel are done too, the current focus decides on which panel the function is performed. This can be done via the menu or the
Short key: ‘Ctrl’+‘Z’.

Redo: If the agents file is modified in the agents panel and undo steps are performed this function redoes the last undone operation. If modifications in the rules panel are done too, the current focus decides on which panel the function is performed. Redo can done via the menu or the
Short key: ‘Ctrl’+‘Y’.

Find: If the agents file is editable in the agents panel a find dialog is opened with the
Short key: ‘Ctrl’+‘F’.
Available options in this dialog are search direction, search scope, wrap search and case sensitive search. After clicking `Find` the dialog will be closed and eventually the found text will be marked. The opened dialog can be canceled with the short key ‘Esc’ and the find process can be started with the short key ‘Enter’, too. If the rules file is also editable, the current focus determines on which panel the find process takes effect.

Find next: The next occurrence of an already searched text can be found with the
Short key ‘F3’.
Search options are the same as for “Find”.

Replace: A replace dialog for the agents panel can be opened with the
Short key ‘Ctrl’+‘H’.
Finding options are the same as in the “Find”dialog. The dialog is opened until the `Close` button or the short key ‘Esc’ is pressed. The button `Find` or the short key ‘Enter’ searches the next occurrence in the text; the button `Replace` replaces the first occurrence of found text after the current cursor position; the button `Replace and Find` replaces the first occurrence and searches the next occurrence of the searched text. Finally the button `Replace All` replaces all occurrences of the searched text. If the rules file is also editable, the current focus determines on which panel the find and replace process takes effect.

4.2.2. **Rules panel.**

The panel has on the upper side buttons to create, open and modify rules files. The text panel contains the current rules. Rules with successfully read priority, preconditions and actions are shown in black, comment lines are shown in gray and rules with errors are shown in red. A rule definition is valid if and only if all settings in the specified lines fulfill the specifications. For further information see 3.2.

Possible error messages:

- If an error on reading an include file is found the message `"% Error: Specified file not found or error in nested file! Please check it!"` is displayed.
  Possible causes are: the file doesn't exist; the path to the file contains spaces, which is not allowed; some rule in some include file (they can be nested) contains irregular expressions or some file contains an irregular include statement. It would be possible that a negative priority offset causes problems, too.
- If an include statement is found at an irregular position the message `"% Error: Include statement only allowed at the beginning of the file!"` is shown.
- If an irregular syntax in an include line or in any of the agent colors is found the message `"% Syntax error!"` is shown.
- If the priority isn't an integer > 0 the message `"% Wrong syntax! Rule ignored!"` is displayed in the second line of the rule.
- `"% Irregular Precondition! Rule ignored!"` shows that the precondition contains either a syntax error or that an `Empty Field`, `Filled Field`, `Smaller Priority`, `Smaller or Equal Priority` or `State of a remote agent` refers to a position outside the universe.
- `"% Irregular Precondition! Priority conditions and NonPriority conditions are mixed! Rule ignored!"` is shown if a NOT condition (`!`) contains priority and non priority conditions (see 3.2.1).
- `"% Irregular Action! Rule ignored!"` occurs if a syntax error in the action line is found or if a `Position change` would lead to move outside the universe.

Possible warnings (can be disabled, see 3.3.2):

- `"% WARNING: DUPLICATE RULE NAME!"`
  shows that a rule with the same name has already been defined.

In the rules panel, the following options can also be found:

New: To create a new rules file. After clicking on the button the text panel will be cleared. Rules can be defined and saved. To interrupt the process click on Reset. To store the new definitions click on Save.
Short key: `'Ctrl'+'Shift'+'N'`.

Open: Opens a rules file. A dialog to save all log files appears before opening the new definition (see button Save) because when the rules file is opened, the log files are overwritten.
Short key: `'Ctrl'+'Shift'+'O'`.

Modify: With this button the current rules file can be modified.
Short key: `'Ctrl'+'Shift'+'M'`.

Reset: If a new rules definition is created or an old one is modified this option discards the new definition, and the last read rules become active.
Short key: `'Ctrl'+'Shift'+'R'`.

Save: To save an existing rules definition or a modified one. If this button is clicked all log files are overwritten and the new rules file is read immediately.

No dialog appears to store existing log files!
Short key: `'Ctrl'+'Shift'+'S'`.

Save as: To save a new rules definition or a modified one. If this button is clicked all log files are overwritten and the new rules file is read immediately. A dialog to save all log files appears before loading the new definition.
Short key: `'Ctrl'+'Shift'+'W'`.

Undo: If the rules file is modified in the rules panel this function undoes the last modification. If modifications in the agents panel are done, too, the current focus decides on which panel the function is performed. Undo is possible via the menu or the
Short key: `'Ctrl'+'Z'`.

Redo: If the rules file is modified in the rules panel and undo steps are performed, this function redoes the last undone operation. If modifications in the agents panel are done too, the current focus decides on which panel the function is performed. Redo can be done via the menu or the
Short key: `'Ctrl'+'Y'`.

Find: If the rules file is editable in the rules panel a find dialog is opened with the
Short key: `'Ctrl'+'F'`.
Available options in this dialog are search direction, search scope, wrap search and case sensitive search. After clicking `Find` the dialog will be closed and eventually the found text will be marked. The opened dialog can be canceled with the short key `'Esc'` and the finding process can be started with the short key `'Enter'`, too. If the agents file is also editable, the current focus determines on which panel the find process takes effect.

Find next: The next occurrence of an already searched text can be found with the
Short key `'F3'`.
Search options are the same as for "Find".

Replace: A replace dialog for the rules panel can be opened with the
Short key `'Ctrl'+'H'`.
Options are the same as for "Find". The dialog is opened until the `Close` button or the short key `'Esc'` is pressed. The button `Find` or the short key `'Enter'` search for the next occurrence in the text; the button `Replace` replaces the first occurrence of found text after the current cursor position; the button `Replace and Find` replaces the first occurrence and searches for the next occurrence of the search text. Finally the button `Replace All` replaces all occurrences of searched text. If the agents file is also editable, the current focus determines on which panel the find and replace process takes effect.

4.3. actions.log. This panel shows the log file `files/actions.log` (see 6). The panel contains in its upper side a combo box to select the iterations to be visualized. If an iteration range is performed more than once a message `later iterations:` is inserted to the text panel.

Let us examine an example. First the iterations 1-48 are performed. Then 20 back steps are done, and finally the iterations 29-42 are performed. If now on the actions panel the set `Iteration 40-49` is selected, the following lines are shown:

```
% Iteration 40
  Agent 1 (28/9): SOUTH
% Iteration 41
  Agent 1 (28/8): SOUTH-WEST
% Iteration 42
  Agent 1 (29/8): SOUTH-WEST
% Iteration 43
  Agent 1 (30/8): SOUTH-WEST
% Iteration 44
  Agent 1 (31/8): SOUTH-WEST
% Iteration 45
% Iteration 46
  Agent 1 (32/8): SOUTH-WEST
% Iteration 47
  Agent 1 (33/8): SOUTH-WEST
% Iteration 48
  Agent 1 (34/8): SOUTH-WEST
-----------------------------------------------------
 later iterations:

% Iteration 40
  Agent 1 (28/9): SOUTH
% Iteration 41
  Agent 1 (28/8): SOUTH-WEST
% Iteration 42
  Agent 1 (29/8): SOUTH-WEST
```

Combo box iteration n-m: Selects the shown iteration range.
Save as: Button to store the `files/actions.log` file to another file.
Short key: `'Ctrl'+'S'`.

4.4. positions.log. This panel shows the log file `files/positions.log`, which stores the situation of all agents at each iteration (for more details on the content of this file, see 6). The handling is analogous to the one described in 4.3.

4.5. error.log. This panel shows the log file `files/error.log`, which stores all warning and error messages (see 6 for more details on the content of this file). The handling is analogous to the one described in 4.3.

4.6. Agents generator. The agents generator is still under construction.

4.7. Exit menu. To end the program.

4.8. Universe menu. This menu contains all functions of the universe panel. For details see 4.1.

4.9. Agents menu. This menu contains all agent functions of the panel "Agents and Rules". More details can be found in 4.2.1.

4.10. Rules menu. This menu contains all rules functions of the panel "Agents and Rules". More details can be found in 4.2.2.

4.11. Agents generator menu. See 4.6.

4.12. Options menu. After clicking on this menu a dialog with option values is opened. The values which can be set are described in 3.3. In the dialog the options can be modified and stored. Modifications can be discarded and options can also be reset to their standard values.

4.12.1. **Layout options.**

Available layout options, color options [and the grid option]. In contrast to the color options in the file, in the panel the colors can be chosen from a color chooser dialog without knowledge about RGB-values. See 3.3.1.

4.12.2. **Running options.**

Described in 3.3.2.

4.12.3. **Behavior options.**

Described in 3.3.3.

4.12.4. **Limitations.**

Limitations to the universe range are available. See 3.3.4 for details.
Attention:

- If new limitations are stored, the current universe state is invalid. Thus all agents and rules must be read once again and all log files are overwritten. To avoid data loss a dialog window to save all log files is shown.
- This option does not have any effect if the parameter `U` is used in the agents definition (see 3.1).

4.13. Help menu. This menu contains an entry `Shortkeys` which gives an overview of available short keys. "Color names for states" lists all available names, which can be used in the rules file (refer to 3.2.4). Finally it contains an entry `About`.

5. Limitations

The system has the following restrictions.

- The program needs a minimal display resolution of 800×600 pixels. If the display width is less than or equal to 1024 the buttons of the universe panel are arranged in two raws. If the width is greater than 1024 the buttons are arranged in one row.

- The duration of one step in the program differs on different modes. The fastest mode is the jump-mode because only the last step is shown. Multiple steps forward (Next 10 steps, Next n steps, Run forever) in speed-mode (see 4.1) are almost as fast as jump steps. Single steps forward are a little bit slower than multiple steps. Backwards steps are the slowest steps, because all agents are displayed at each step.
- If the option History (3.3.2) is set to `all` the application is limited because of limited memory resources. If resources are exhausted an `OutOfMemory` error occurs. In this case an error dialog is shown. On the dialog the button `End Program` exists to end the program.
  On Windows XP with a standard stack size of 64MB this limit is reached with approximately 35000 agents stored in the history. This can be reached for instance with 200 agents in 175 iterations if all agents act at each step.
  The size of the stack can be set manually with a JVM-Flag `-Xmx[n]m` whereas the value for n is the stack size in MB.

6. Program outputs

In addition to the graphical representation, the program generates three output files:

- `files/positions.log`
- `files/actions.log`
- `files/error.log`.

For better readability of these files, the agents are given numerical id's according to the order they appeardefinition in the file `files/agents.txt`, starting with number 1.

The file `files/positions.log` contains a complete state history of the agents. Starting from the initial setting, and then after every (global) calculation step, all agents are listed in the order of their numerical id's. For each agent one line is printed, containing its numerical id, its current global position, its attachments, its state, the values of all its counters, and all current incoming messages (numerical and text).

Example with numerical messages:

```
% Iteration 4
  1: 0/5 A001010 SLSent C0 0 C1 0 C2 0 C3 0 C4 0 C5 0 C6 0  . . . C24 0
  2: 1/6 A001101 SLSent C0 0 C1 0 C2 0 C3 0 C4 0 C5 0 C6 0  . . . C24 0
  3: 2/6 A010101 SFORWD C0 0 C1 2 C2 0 C3 0 C4 0 C5 0 C6 0  . . . C24 0
  4: 1/5 A111010 SFORWD C0 1 C1 2 C2 0 C3 0 C4 0 C5 0 C6 0  . . . C24 0
  5: 2/5 A111010 SUNDEF C0 0 C1 0 C2 0 C3 0 C4 0 C5 0 C6 0  . . . C24 0 #12 #32
  6: 3/5 A010101 SUNDEF C0 0 C1 0 C2 0 C3 0 C4 0 C5 0 C6 0  . . . C24 0
  7: 3/4 A101010 SUNDEF C0 0 C1 0 C2 0 C3 0 C4 0 C5 0 C6 0  . . . C24 0
  8: 4/4 A010010 SUNDEF C0 0 C1 0 C2 0 C3 0 C4 0 C5 0 C6 0  . . . C24 0
```

Example with text messages:

```
% Iteration 16
```

```
  1: 0/5 A001010 SLSent C0 0 C1 0 C2 0 C3 0 C4 0 C5 0 C6 0  . . . C24 0
  2: 1/6 A001101 SLSent C0 0 C1 0 C2 0 C3 0 C4 0 C5 0 C6 0  . . . C24 0
  3: 2/6 A010101 SFORWD C0 0 C1 2 C2 0 C3 0 C4 0 C5 0 C6 0  . . . C24 0 M1WRONG
  4: 1/5 A111010 SFORWD C0 1 C1 2 C2 0 C3 0 C4 0 C5 0 C6 0  . . . C24 0 M6OK___
  5: 2/5 A111010 SFINIS C0 2 C1 3 C2 0 C3 0 C4 0 C5 0 C6 0  . . . C24 0
  6: 3/5 A010101 SFINIS C0 0 C1 4 C2 0 C3 0 C4 0 C5 0 C6 0  . . . C24 0
  7: 3/4 A101010 SFINIS C0 4 C1 0 C2 0 C3 0 C4 0 C5 0 C6 0  . . . C24 0
  8: 4/4 A010010 SFINIS C0 0 C1 5 C2 0 C3 0 C4 0 C5 0 C6 0  . . . C24 0
% Iteration 17
  1: 0/5 A001010 SLSent C0 0 C1 0 C2 0 C3 0 C4 0 C5 0 C6 0  . . . C24 0 M6OK___
  2: 1/6 A001101 SLSent C0 0 C1 0 C2 0 C3 0 C4 0 C5 0 C6 0  . . . C24 0 M6WRONG M4WRONG
  3: 2/6 A010101 SFINIS C0 0 C1 2 C2 0 C3 0 C4 0 C5 0 C6 0  . . . C24 0
  4: 1/5 A111010 SFINIS C0 1 C1 2 C2 0 C3 0 C4 0 C5 0 C6 0  . . . C24 0
  5: 2/5 A111010 SFINIS C0 2 C1 3 C2 0 C3 0 C4 0 C5 0 C6 0  . . . C24 0
  6: 3/5 A010101 SFINIS C0 0 C1 4 C2 0 C3 0 C4 0 C5 0 C6 0  . . . C24 0
  7: 3/4 A101010 SFINIS C0 4 C1 0 C2 0 C3 0 C4 0 C5 0 C6 0  . . . C24 0
  8: 4/4 A010010 SFINIS C0 0 C1 5 C2 0 C3 0 C4 0 C5 0 C6 0  . . . C24 0
```

The file `files/actions.log` contains a complete history of which agent applied which rules, and when. In every global calculation step, the printing is done after the decision is taken of which agents will perform which rules, but before the rules are applied. For every agent and every applied rule one line is printed, containing the numerical id of the agent, its current position, and the name of the rule to be applied.

The following example shows an `actions.log` file. In iteration 3 agent 4 applies two rules. The possibility of performing multiple rules on one agent is a very powerful tool, which allows simplifying the rules set. It must be handled very carefully!

```
% Iteration 1
  Agent 1 (0/5): MAKE_LMOST_TRY
  Agent 2 (1/6): MAKE_LMOST_TRY
% Iteration 2
  Agent 1 (0/5): SEND_LMOST
  Agent 2 (1/6): SEND_LMOST
% Iteration 3
  Agent 3 (2/6): STORE_FW_MSG_FROM_NORT-WEST
  Agent 4 (1/5): STORE_FW_MSG_FROM_NORTH
  Agent 4 (1/5): STORE_FW_MSG_FROM_NORT-WEST
% Iteration 4
  Agent 3 (2/6): FW_MSG_IF_W
  Agent 4 (1/5): FW_MSG_IF_N_AND_W_C1__
% Iteration 5
  Agent 5 (2/5): STORE_FW_MSG_FROM_NORTH
  Agent 5 (2/5): STORE_FW_MSG_FROM_NORT-WEST
% Iteration 6
  Agent 5 (2/5): FW_MSG_IF_N_AND_W_C1__
% Iteration 7
  Agent 6 (3/5): STORE_FW_MSG_FROM_NORT-WEST
% Iteration 8
  Agent 6 (3/5): FW_MSG_IF_W
% Iteration 9
```

```
  Agent 7 (3/4): STORE_FW_MSG_FROM_NORTH
% Iteration 10
  Agent 7 (3/4): FW_MSG_IF_N
```

The file `files/error.log` contains a complete history of occurred errors. In the following example warning and error messages are shown. Each warning starts with `warning(` and ends with a `)`; error messages start with the string `error(` and end with a `)`.

```
% Iteration 122
warning(Perform Rule: COLLISION IN UNIVERSE)
  error(Consistency Error B:
    Universe[32,15] = 16
    Agent[13] = 32,15)
% Iteration 123
% Iteration 124
warning(Internal Error! Agent on 32,15 mark Attached to agent 10, but isn't there!)
warning(Internal Error! Agent on 32,15 mark Attached to agent 10, but isn't there!)
```

7. Requirements

The program is developed in Java 1.4.2.15. in order to run the application Java JRE 1.4 or higher must be installed.

8. Installation

To install the program, unzip the zip file in a folder of your choice. The program folder contains the program file `AgentSystem2D.jar` and a folder `files` with the files

- `config.txt`
- `agents.txt`
- `rules.txt`

The file `files/config.txt` is necessary to run the program.
The files `files/agents.txt` and `files/rules.txt` are used as the starting initial setting.
The files

- `files/actions.log`
- `files/positions.log`
- `files/error.log`.

are not initially necessary: they are created on starting the program.

Referencias

[1] M. Yim, W.-M. Shen, B. Salemi, D. Rus, M. Moll, H. Lipson, E. Klavins, G. S. Chirikjian. Modular Self-Reconfigurable Robot Systems: Challenges and Opportunities for the Future. *IEEE Robotics and Automation Magazine, 14(1):43-52*, (2007).

[2] S. Murata, H. Kurokawa.
Self-Organizing Robots.
Springer Tracts in Advanced Robotics, Vol. 77, (2012)

[3] O. Aichholzer, T. Hackl, V. Sacristán, B. Vogtenhuber, R. Wallner.
Simulating distributed algorithms for lattice agents.
Submitted to Proc. XV Spanish Meeting on Computational Geometry, June 26-28, (2013)

[4] R. Wallner.
A System of Autonomously Self-Reconfigurable Agents.
Diploma Thesis, Institute for Software Technology, Graz University of Technology, (2009)

[5] F. Hurtado, E. Molina, S. Ramaswami, V. Sacristán.
Distributed universal reconfiguration of 2D lattice based modular robots.
European Workshop on Computational Geometry, Braunschweig, March 17-20, (2013)

[6] O. Rodríguez.
Simulació de l'actuació distribuïda de robots modulars.
Degree Thesis in Computer Science, (2013)

[7] http://www-ma2.upc.edu/vera/local-reconfiguration