Reconfiguración distribuida de robots cristalinos

Manuel Perera Paquico

Proyecto de Fin de Carrera
Directora: Vera Sacristán Adinolfi
Departamento: Matemática Aplicada II

Ingeniería informática
Facultad de Informática de Barcelona
Universitat Politécnica de Cataluña

23 de marzo de 2015

La libertad no consiste en tener un buen amo,
sino en no tenerlo.
Marco Tulio Cicerón

Índice general

Capítulo 1

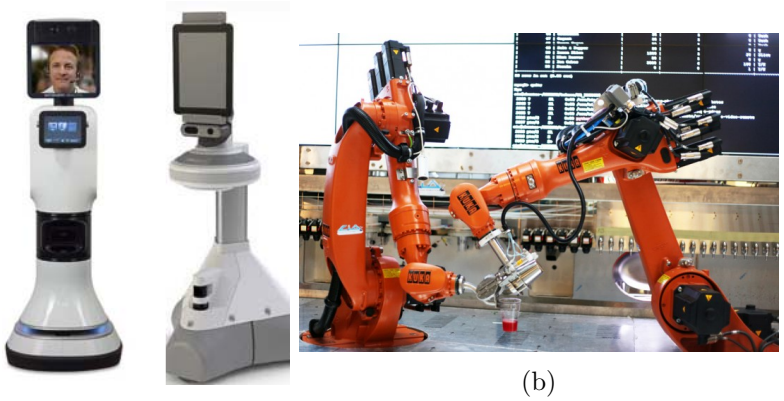
Introducción

Este proyecto está dedicado al diseño, la implementación y el análisis de nuevos algoritmos de reconfiguración de robots cristalinos de forma distribuida. Este capítulo empieza describiendo las características de los robots modulares, sus diferencias con los robots especializados y las características del modelo utilizado en nuestro proyecto. Seguidamente, se describen los objetivos del proyecto y, finalmente, la organización de esta memoria.

1.1. Robots modulares

Un robot es una entidad virtual o mecánica artificial que, mediante técnicas de inteligencia artificial o a través de un programa predefinido, realiza tareas de forma automática. Aunque normalmente un robot puede desarrollar múltiples tareas de manera flexible según su programación, los robots más comunes hoy en día son los robots especializados. Pueden verse algunos ejemplos en la Figura 1.1. Estos robots están diseñados para realizar una única tarea y están limitados por su forma y construcción. Por ejemplo, un brazo mecánico de una cadena de montaje, aunque preciso, es incapaz de cambiar su ubicación a no ser que haya sido dotado de algún sistema de movimiento. En un intento de solventar estas limitaciones se diseñaron los robots modulares.

Un robot modular es aquél que esta formado por módulos o unidades independientes más pequeños, y que es capaz de cambiar su forma para adaptarse a cualquier situación a la que pueda enfrentarse. La Figura 1.2 muestra algunos ejemplos. Los módulos son idénticos y, por tanto, intercambiables entre sí. De esta forma si una unidad resulta dañada durante una acción puede ser substituida por otra, reparando así el robot modular. Siguiendo el ejemplo anterior, si creamos un robot modular con forma de brazo mecánico de una cadena de montaje, una vez que este deje de ser necesario en su linea de la cadena puede cambiar su base para desplazarse a otras lineas y reforzarlas o substituir un robot averiado. Todas estas carac-



(a) (b)

Figura 1.1: Esta figura muestra dos ejemplos de robots especializados. A la izquierda, imagen a, los robots Ava y RPVITA, especializados en teleasistencia y telepresencia. A la derecha, imagen b, el robot Makr Shakr, especializado en la preparación de cócteles. Los tres robots están diseñados para realizar funciones muy específicas y ninguno puede realizar el trabajo de los otros.

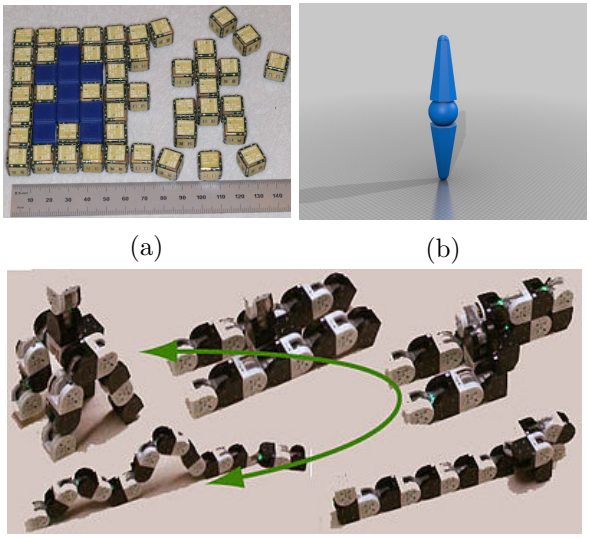
terísticas dan a los robots modulares una gran flexibilidad así como un gran número de usos.

1.2. Robots cristalinos

Los robots con los que trabajamos en este proyecto son robots autoreconfigurables reticulares cuadrados o cúbicos. Cada cara de cada unidad permite tanto acoplarse y desacoplarse a sus vecinos como extenderse en dirección a la normal de la cara y contraerse en dirección contraria. La Figura 1.3 muestra diversos prototipos de este tipo de robot. De esta forma podemos formar toda una estructura de unidades conectadas por sus caras en donde una unidad puede empujar o estirar a su vecino. En nuestros algoritmos, cada unidad del robot es independiente y toma sus propias decisiones sin necesidad de un controlador central. Además, los átomos pueden enviar información a sus vecinos y almacenarla en registros internos.

1.3. Objetivo del trabajo

Este proyecto consiste en implementar y estudiar una serie de algoritmos en dos dimensiones que buscan mejorar el algoritmo de Joan Soler Pascual descrito en su proyecto de final de carrera *Reconfiguració de robots cristal·lins* [3] el cual es una versión distribuida de un algoritmo original de Aloupis et al [2]. Dicho algoritmo consigue que un robot formado por metamódulos de



(a) (b)

(c)

Figura 1.2: Esta figura muestra diferentes tipos de robots modulares tanto reales como ficticios. La imagen *a* muestra un grupo de imanes electropermanentes para materia programable (2010), capaces de reproducir cualquier forma que sus vecinos hayan rodeado. La imagen *b* muestra un modelo de un átomo microbot de la última película de Disney *Big Hero 6* (2014), capaces de recrear cualquier forma, realizar tareas de construcción y de transporte. Por último, en la imagen *c*, podemos ver un robot modular MTRAN3 (2005), que cuenta con la habilidad de cambiar de forma para moverse como una serpiente, caminar o incluso rodar.

robots más pequeños sea capaz de cambiar su forma inicial a cualquier otra forma que se le indique, siempre y cuando tenga suficientes módulos para ello. Para llegar a la forma final el algoritmo interpreta tanto esta como la forma inicial como dos árboles generadores que comparten la misma raíz y que tienen módulos de robots por nodos. Primero mueve los módulos hacia la raíz siguiendo las ramas de la forma inicial para luego expandir dichos módulos formando, de una en una, las ramas de la forma final.

La acción más costosa posible que puede realizar un robot modular es moverse, mucho más que enviar un mensaje o almacenar datos en sus registros, por eso en este proyecto buscamos reducir el número de movimientos del algoritmo original.

El proyecto ha sido dividido en los siguientes objetivos:

- Diseño e implementación de cuatro algoritmos distribuidos en 2D que buscan mejorar el rendimiento del algoritmo original. Todos estos algoritmos se encuentran descritos en el Capítulo 3 de esta memoria.

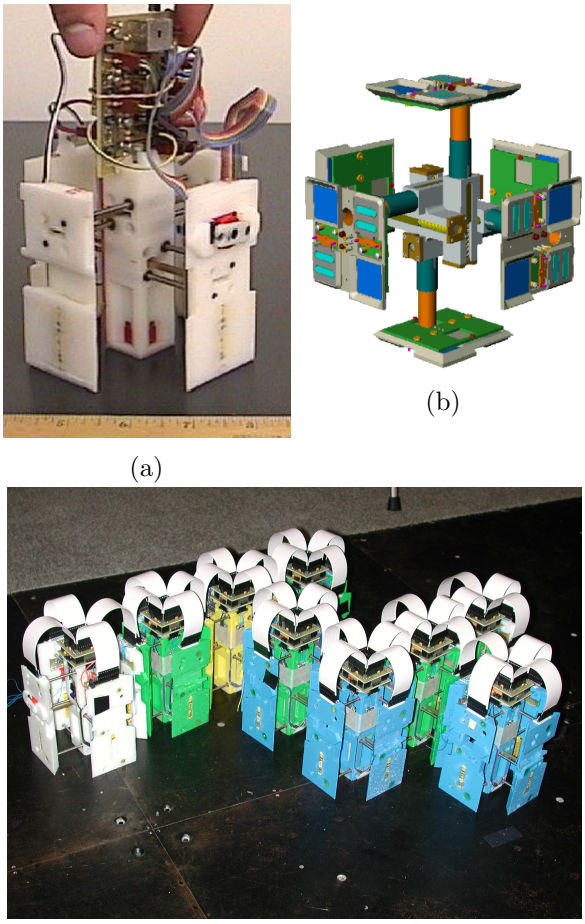
- Análisis y experimentación de los algoritmos 2D.

1.4. Estructura de la memoria

La memoria se divide en ocho capítulos:

- Capítulo 1: Introducción a los robots modulares y breve explicación del modelo usado en nuestro trabajo.
- Capítulo 2: Explicación de los movimientos básicos de los robots cristalinos y del funcionamiento del simulador.
- Capítulo 3: Descripción sencilla de los algoritmos de mejora presentados en el proyecto.
- Capítulo 4: Implementación del algoritmo multilíder, el más complejo de todos los algoritmos presentados en este proyecto.
- Capítulo 5: Estudio de la complejidad y análisis experimental tanto de los algoritmos presentados en este proyecto como del algoritmo original.
- Capítulo 6: Manual de usuario de un analizador de acciones diseñado para facilitar la fase experimental del Capítulo 5.
- Capítulo 7: Gestión del proyecto.
- Capítulo 8: Conclusiones y reflexión sobre el resultado del proyecto.

Al final de esta memoria se puede encontrar una lista de referencias consultadas durante el proyecto y un anexo en donde se detallan las diferentes fases de una reconfiguración y el significado de los estados, señales y registros usados por los átomos del robot modular.



(a) (b)

(c)

Figura 1.3: (*a*) Primer prototipo de robot cristalino [1], el cual solo puede expandir o contraer sus cuatro caras al mismo tiempo. (*c*) Segundo prototipo de robot cristalino [1] que consigue expandir y contraer a la vez caras opuestas, ofreciendo un mayor grado de libertad de movimiento a las unidades del robot. (*b*) Diseño final de un robot telecube[4] que puede mover cada una de sus caras de forma independiente y en tres dimensiones.

Capítulo 2

El modelo y la simulación

En este capítulo se describen las características del simulador de robots cristalinos [6] desarrollado por Reinhard Wallner [5] y del robot modular que simula. Este simulador permite experimentar con robots modulares cristalinos formados por unidades como las descritas en el Apartado 1.2, agrupadas en metamódulos de tamaño 2×2. Esta agrupación de unidades en metamódulos, junto con las operaciones básicas descritas anteriormente (acoplarse, desacoplarse, expandir y contraer), permite realizar operaciones mucho más complejas que no son posibles a nivel atómico [2].

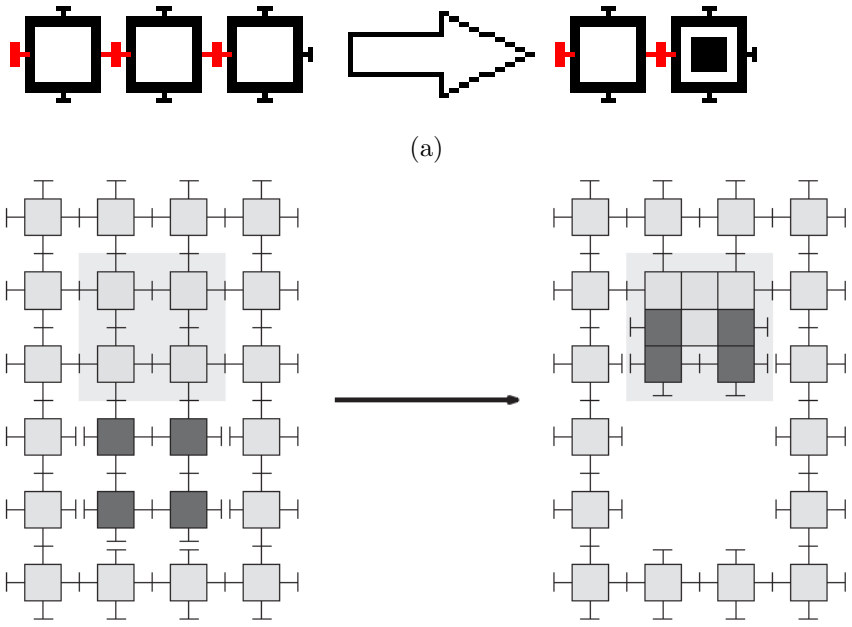
2.1. Movimientos modulares

A continuación se presenta una descripción de estas operaciones e imágenes de cómo se ven desde el punto de vista del simulador. Todas las operaciones descritas se realizan sobre dos módulos vecinos conectados entre sí por las caras de dos de sus unidades. Más información sobre la ejecución de estas operaciones desde un punto de vista atómico puede hallarse en el Apéndice de [2].

2.1.1. Compresión

Los módulos con los que trabajamos tienen la capacidad de alojarse en el interior de sus vecinos siempre y cuando ninguno de los dos contenga ya otro módulo en su interior, tal como se ilustra en la Figura 2.1. Esta operación es posible gracias a que un módulo al expandir las caras interiores de sus átomos, las que están conectadas a otro átomo del módulo, crea suficiente espacio en su interior como para poder alojar otros cuatro átomos. De esta forma, el módulo que se comprime pasa a ocupar la misma posición que el módulo que hace de anfitrión. Una vez realizada la operación, el simulador no permite dar órdenes a un módulo comprimido directamente, por lo que todas las órdenes de movimiento o descompresión que deban aplicarse al

módulo comprimido deben ser ejecutadas por el módulo anfitrión.



(a)

(b)

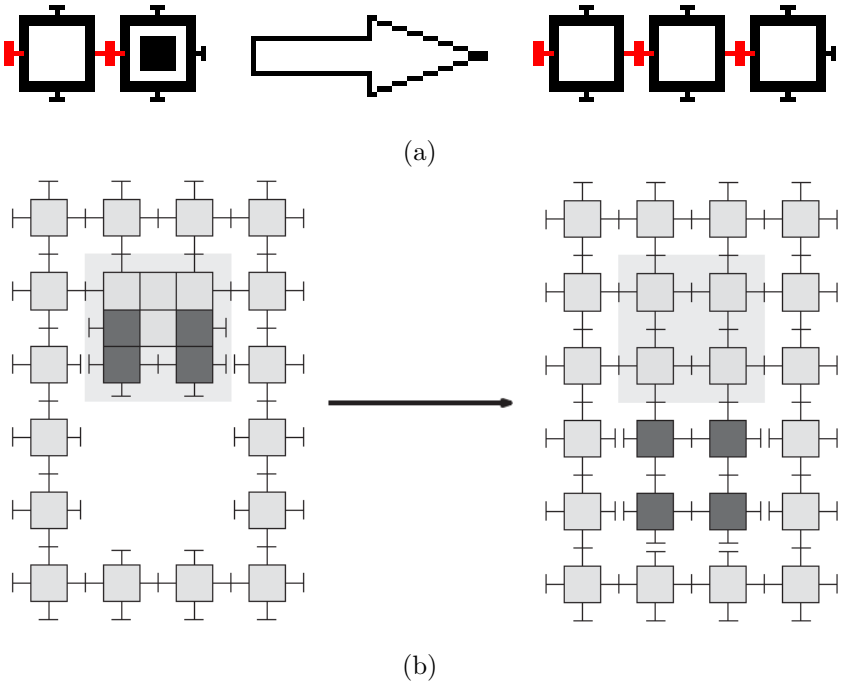
Figura 2.1: El proceso de compresión. En la imagen superior, *a*, a la izquierda vemos tres módulos, uno de los cuales, el derecho, esta a punto de ser alojado por el módulo central. A la derecha vemos el estado de los tres módulos una vez ha acabada la operación de compresión. El módulo de la derecha, representado ahora como un cuadrado negro contenido en el módulo central, ha pasado a ocupar la misma posición del módulo central. La vista superior es modular, la inferior, imagen *b*, es atómica.

2.1.2. Expansión

La expansión, ilustrada en la Figura 2.2, es el movimiento inverso a la compresión. El módulo anfitrión indica al módulo comprimido la dirección por la que debe expandirse y este obedece y pasa a ocupar la posición indicada. Para el correcto funcionamiento de esta operación la posición indicada por el módulo anfitrión debe permanecer vacía durante la ejecución de la operación.

2.1.3. Paso de módulos comprimidos

Esta operación consiste en el paso de un módulo comprimido a uno de los vecinos de su anfitrión. Se ilustra en la Figura 2.3. En el simulador el anfitrión ordena al módulo comprimido que cambie a un nuevo anfitrión indicando la



(a)

(b)

Figura 2.2: El proceso de expansión es el inverso del de compresión.

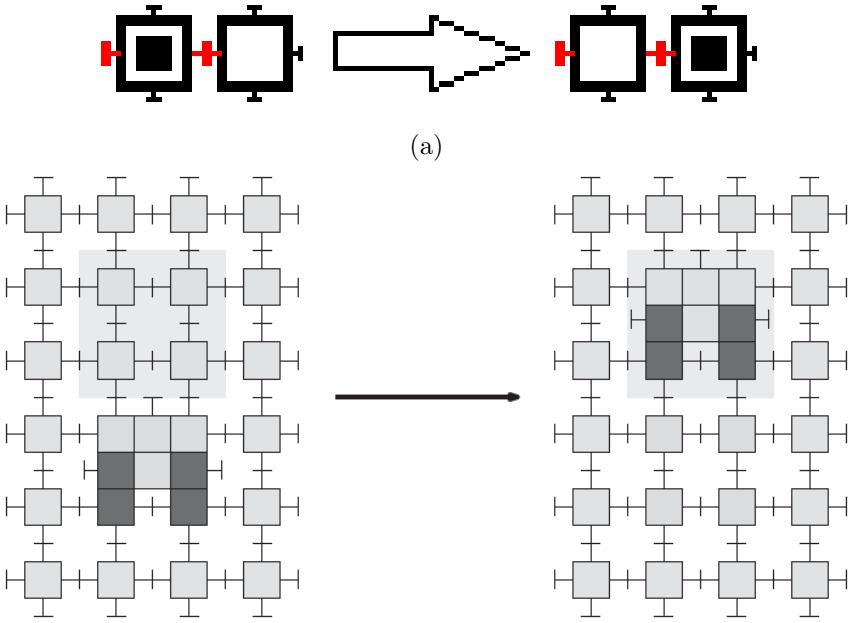
dirección de este. Una vez ha terminado la operación el módulo comprimido ha pasado a ocupar la misma posición que el nuevo módulo anfitrión. Para que la operación se realice con éxito el nuevo módulo anfitrión debe estar vacío mientras dura la ejecución de la operación.

2.1.4. Otras operaciones

El simulador no soporta ningún otro tipo de operación entre módulos a parte del envío de señales, entre las que se incluyen la consulta de estados y registros de módulos vecinos. Aún así, los átomos de este tipo de robot modular pueden realizar otras operaciones, como cambiar los papeles de un módulo anfitrión y su módulo comprimido, pasando el anfitrión a estar comprimido y el comprimido a ser anfitrión, o el envío doble, en donde dos módulos anfitriones vecinos se intercambian sus módulos comprimidos.

2.2. Algoritmo distribuido

Los robots modulares son un sistema distribuido de unidades idénticas e intercambiables conectadas entre sí, por tanto, para explotar toda su capacidad, todas las unidades del robot deben ejecutar un mismo algoritmo que permita que cada unidad funcione de forma independiente. Esta clase



(a)

(b)

Figura 2.3: El proceso de paso de módulo comprimido. En la imagen superior, *a*, a la izquierda vemos tres módulos, uno de los cuales, representado como un cuadrado negro, se encuentra alojado dentro del módulo de la izquierda. A la derecha vemos el estado de los tres módulos una vez ha acabado la operación de paso de módulo comprimido. El módulo de la derecha, que antes se encontraba vacío, es ahora anfitrión del módulo que anteriormente se encontraba alojado en el módulo de su izquierda. La vista superior es modular, la inferior, imagen *b*, es atómica.

de algoritmos, pensados para ejecutarse en un sistema distribuido, se llaman algoritmos distribuidos.

Hasta ahora los algoritmos existentes utilizados para la reconfiguración de robots cristalinos no eran totalmente distribuidos. Por ejemplo, si bien el algoritmo de Joan Soler consigue que la fase de compresión sea totalmente distribuida, todos los módulos se comprimen hacia la raíz siempre que les es posible, la fase de expansión sigue siendo secuencial y dependiente de un único módulo que dirige el proceso.

En este proyecto hemos desarrollado un algoritmo totalmente distribuido que aprovecha todas las características del sistema distribuido.

2.2.1. Las Reglas

Cada una de las reglas del algoritmo distribuido que ejecuta cada uno de los módulos del robot cristalino está compuesta por cuatro lineas que son

interpretadas por el simulador:

1. Nombre: La primera linea de una regla es su nombre. Al principio de esta linea puede verse además la fase a la que pertenece la regla. Esto último no es obligatorio para el funcionamiento del simulador, pero permite al usuario del simulador entender mucho mejor el funcionamiento de las reglas.
2. Prioridad: La prioridad de una regla es un número entero de 1 a 32767 que marca la importancia de la regla respecto a las demás. Una vez se ha decidido que reglas pueden aplicarse a un módulo concreto, estas se ordenan de forma decreciente según su prioridad, siendo así las reglas con mayor prioridad las primeras en ejecutarse.
3. Precondición: Lista de condiciones que deben cumplirse para que una regla pueda aplicarse a un módulo.
4. Acciones: Acciones que lleva a cabo un módulo que cumple las precondiciones establecidas por la regla, al ejecutarla.

En la Figura 2.4 podemos ver una de la reglas del algoritmo multilíder.

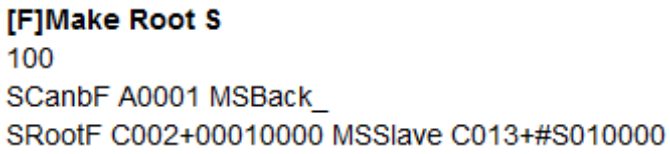


Figura 2.4: Regla del algoritmo multilíder. La primera linea muestra su nombre, la segunda su prioridad, la tercera su precondición y la cuarta su acción.

2.2.2. Precondición

La precondición puede consistir en una o más de las siguientes partes.

Vecinos N____

Indica, para cada dirección del módulo, si el módulo debe tener un vecino o una posición vacía. Las cuatro posiciones que siguen a la N indican las cuatro caras del módulo: norte, oeste, este y sur. El número 0 indica que no debe haber vecino, 1 que debe haber vecino y * que no importa. Por ejemplo, *N0010* indica que el módulo debe tener un único vecino al este.

Espacio vacío Edx,dy

Esta precondición comprueba si la posición relativa indicada por números enteros o contadores del módulo está vacía.

Espacio ocupado Fdx,dy

Esta precondición es la contraria que la de espacio vacío.

Prioridad de los vecinos P____

Comprueba si la prioridad de las reglas aplicadas por los los vecinos del norte, oeste, este o sur es menor o igual a la del módulo. < indica que la prioridad del vecino en esa dirección debe ser menor que la del módulo, = indica que debe ser igual y * que no importa.

Prioridad menor remota Ldx,dy

Esta precondición comprueba si la prioridad de los módulos indicados por la posición relativa dada por números enteros o registros del módulo es menor que la del módulo.

Prioridad menor o igual remota Qdx,dy

Esta precondición comprueba si la prioridad de los módulos indicados por la posición relativa dada por números enteros o registros del módulo es menor o igual que la del módulo.

Conexiones A____

Indica si el módulo debe estar conectado o no a otro módulo vecino. Las cuatro posiciones que siguen a la A indican las cuatro caras del módulo: norte, oeste, este y sur. El número 0 indica que no debe estar conectado, 1 que debe estar conectado y * que no importa. Por ejemplo, *A0010* indica que el módulo debe estar conectado a otro módulo solo por su cara este.

Estado S_____

Estado en que debe encontrarse el módulo. Debe contar siempre con 5 caracteres. El asterisco indica que el carácter en esa posición de la cadena de caracteres no importa. Por ejemplo, *SPaus** indica que el estado debe empezar por *Paus*.

Estado remoto Tdx,dy,_____

Comprueba el estado del módulo indicado por la posición relativa dada por números enteros o por registros del módulo.

Mensaje de texto M_____

Indica que el módulo debe haber recibido un determinado mensaje de texto. La primera posición indica la dirección por la que ha recibido el mensaje (N,W,E,S) mientras que el resto indican el contenido del mensaje. Como el estado, un mensaje debe constar siempre de 5 caracteres. En la precondición, el carácter * indica que el carácter en esa posición de la cadena de caracteres no importa o, si está en primera posición, que no importa la dirección por la que haya recibido el mensaje.

Comparación numérica _ ____ ____

Compara los mensajes numéricos recibidos por el módulo en la iteración actual y el valor de los registros. La primera posición debe estar ocupada por uno de los tres signos de comparación >,< o =. Las siguientes ocho posiciones indican los canales numérico, registros o enteros a comparar. Un canal numérico viene indicado por le símbolo # seguido de la dirección (N,W,E,S) y del número del canal (entre 01 y 08). Los registros se indican con un *C0* seguido del número del registro (entre 00 y 25). Cada valor puede llevar un signo negativo delante (−).

Comparación numérica remota Vdx,dy,C__ ____, Wdx,dy,C__ ____

Funciona igual que la comparación numérica y el resto de comprobaciones remotas. La V comprueba si el primer valor es estrictamente menor que el segundo y la W si el primer valor es menor o igual que el segundo.

Negación !

Niega cualquier parte de la precondición.

Agrupación ()

Agrupa condiciones de la precondición. Normalmente se usa junto con la negación.

2.2.3. Acciones

La última linea de una regla indica las acciones a seguir durante la ejecución de la regla. La nomenclatura de las acciones es la misma que la de las precondiciones de mismo nombre.

Cambio de posición Pdx,dy

Indica la posición relativa a la que debe moverse el módulo. Esta acción no se utiliza en ninguno de los algoritmos distribuidos existentes.

Conexiones A____

Esta acción indica a qué vecinos debe conectarse o de qué vecinos debe desconectarse el módulo.

Estado S____

Esta acción indica el nuevo estado al que debe cambiar el módulo.

Mensaje de texto M_____

Indica la dirección y el mensaje que debe enviar el módulo.

Cálculos numéricos y envío de señales numéricas C__ _ ____ ____, #___ _ ____ ____

Esta acción indica si debe realizarse un cálculo numérico o el envío de una señal numérica. El cálculo numérico se expresa de forma parecida a la comparación numérica, solo que en este caso, antes del signo de la operación, se debe añadir el registro destino del resultado. El envío de señales se expresa de la misma manera que el calculo numérico solo que en lugar de indicar el registro de destino, se indica el canal y la dirección por la que enviar el resultado de la operación. Obligatoriamente, al enviar un mensaje numérico, debe realizarse un calculo numérico. Las operaciones permitidas son la suma (+), la resta (−), la multiplicación (∗), la división (/), el módulo (M), el máximo (A) y el mínimo (I).

Intercambio de módulos comprimidos X_

Indica la dirección del vecino con el cual debe realizarse una operación de intercambio de módulos. Actualmente ningún algoritmo distribuido para el simulador usa esta operación.

Compresión Z_

Esta acción indica la dirección a la que el módulo debe comprimirse.

Descomprimir z_S_____

La primera posición indica la dirección hacia la que el módulo anfitrión debe descomprimir el módulo comprimido que aloja. Las siguientes cinco posiciones indican el estado al que debe pasar el módulo comprimido una vez descomprimido.

Paso de módulo comprimido x_

La primera posición indica la dirección hacia la que el módulo anfitrión debe enviar el módulo comprimido que aloja. La posición indicada por esta acción debe estar ocupada por un módulo que no contenga ningún otro módulo comprimido.

2.3. El simulador

En este apartado se explica, en primer lugar, las diferentes pantallas del simulador y, por último, las características generales de los módulos de átomos con los que trabaja. Para una información más detallada sobre el simulador y su funcionamiento consúltese el manual de usuario en su página web [6].

2.3.1. Universe

El universo es la representación del mundo 2D sobre el que trabaja el simulador y sus características. Ser compone de una serie de botones que permiten viajar en el tiempo del universo, un contador de tiempo y una imagen interactiva del estado del universo en cada momento.

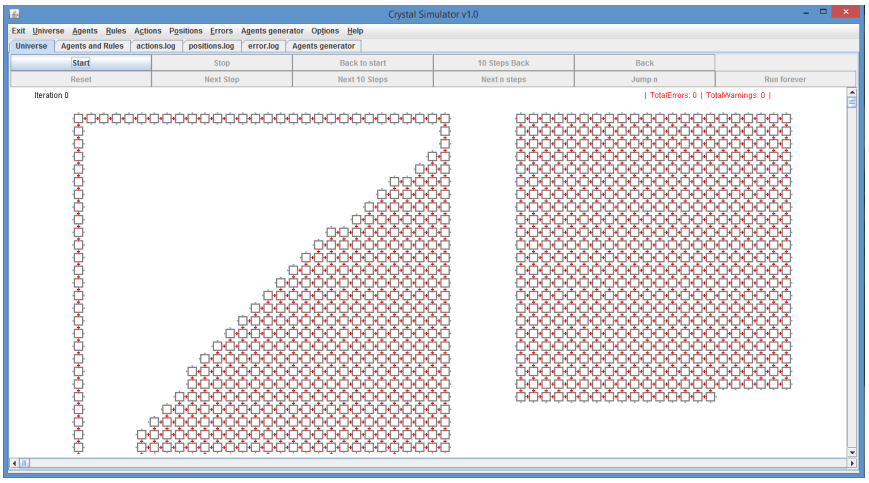


Figura 2.5: Pantalla de *Universe* del simulador.

El tiempo en el simulador se expresa en iteraciones. Una iteración representa una ejecución de todas las reglas por parte de todos las módulos. Como el sistema es distribuido, durante una iteración todos los módulos ejecutan una única vez todo un conjunto de reglas. Aunque este hecho indica que el simulador y todos sus módulos son síncronos, esto no es más que una manera de simular el paso del tiempo. Para llevar a cavo una ejecución asíncrona, habría que introducir un sistema de aleatorización de pausas y ajustar el

sistema de *handshaking* utilizado en las reglas de los algoritmos presentados en este proyecto.

Una vez pulsado el botón *Start*, los botones que controlan el paso del tiempo permiten avanzar o retroceder en el tiempo un número determinado de iteraciones según el botón o, en el caso de ejecutar el algoritmo indefinidamente, pausar la ejecución. Entre los botones y la representación del universo 2D podemos encontrar un contador de iteraciones que indica el momento en el tiempo en que nos encontramos. Toda ejecución comienza en la iteración 0.

A la derecha del contador de iteraciones podemos ver dos contadores que indican el número de errores y avisos generados hasta la iteración actual de la ejecución en curso.

La representación del universo consta de una pantalla cuadriculada con filas y columnas numeradas en donde cada cuadrado representa una posición en el universo 2D. En esta vista podemos ver dibujados los módulos del robot modular, su color (véase la Figura 2.5) y, si pausamos la ejecución y pulsamos con el botón derecho encima del módulo que nos interese, sus datos internos (Figura 2.11).

2.3.2. Agents and Rules

Esta pantalla, tal como se ilustra en la Figura 2.6, permite cargar, visualizar y modificar los ficheros de reglas y de módulos que utiliza el simulador en su ejecución. Una vez ha empezado una ejecución, la modificación de alguno de estos ficheros o el cambio de uno de ellos por otro distinto obliga al simulador a reiniciar la reconfiguración.

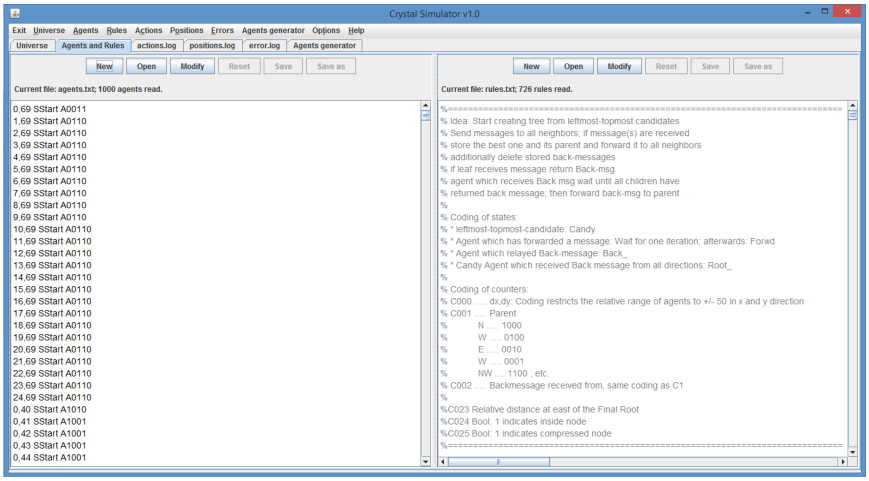


Figura 2.6: Pantalla de *Agents and Rules* del simulador.

2.3.3. Actions

Lista de todas las reglas ejecutadas por cada módulo según la iteración en que tuvo lugar. Desde esta vista, ilustrada en la Figura 2.7, no solo es posible la visualización de la lista de acciones realizada sino que, además, también permite exportar el fichero a nuestro ordenador.

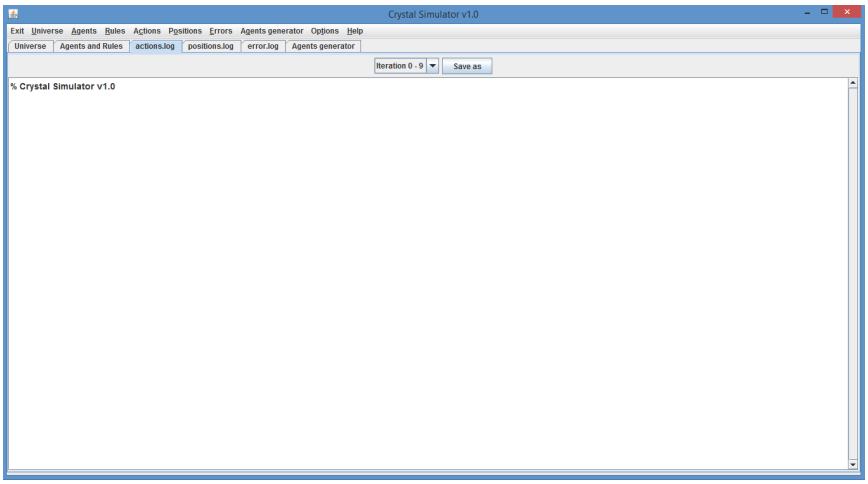


Figura 2.7: Pantalla de *Actions.log* del simulador.

2.3.4. Position

Lista de la posición y los datos internos de cada módulo en cada iteración de la ejecución. Esta ventana (véase la Figura 2.8) también permite exportar el fichero a nuestro ordenador.

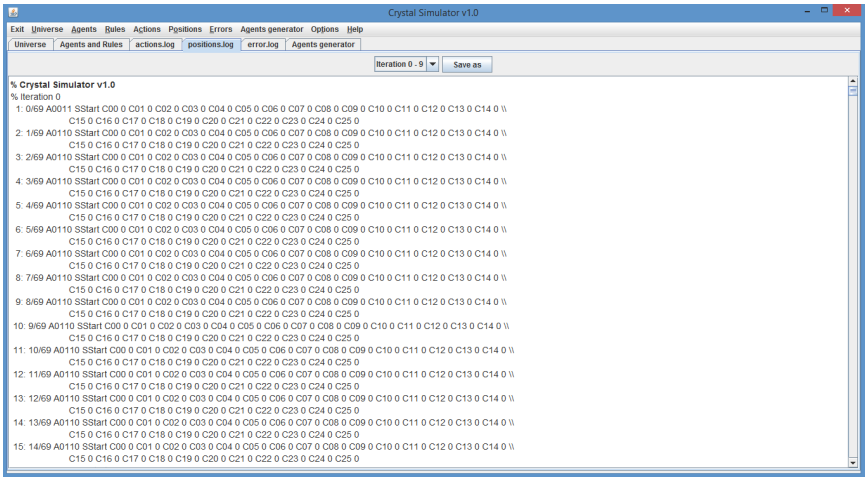


Figura 2.8: Pantalla de *Position* del simulador.

2.3.5. Errors

Lista de los errores y avisos generados por la ejecución del conjunto de reglas en cada iteración de la ejecución. Esta ventana (Figura 2.9) también permite exportar el fichero a nuestro ordenador.

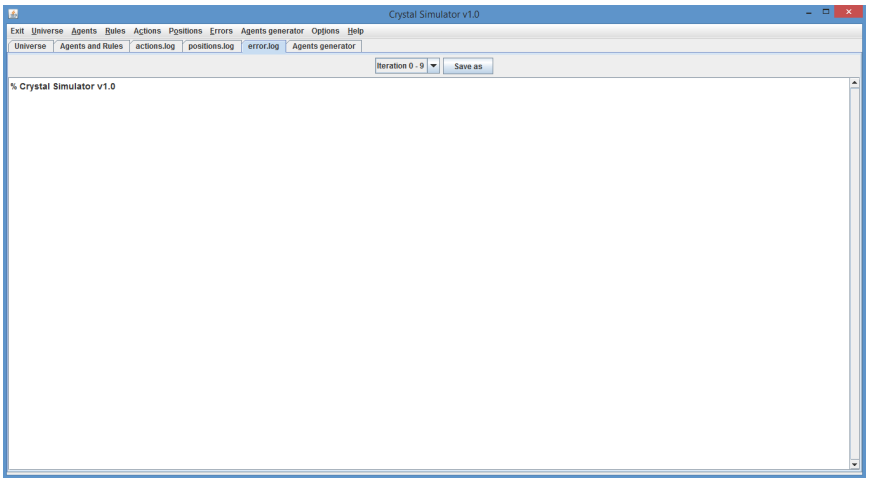


Figura 2.9: Pantalla de *Errors* del simulador.

2.3.6. Agents generator

Editor que permite generar o modificar de forma sencilla un robot modular sobre el que ejecutar un conjunto de reglas. Se ilustra en la figura 2.10.

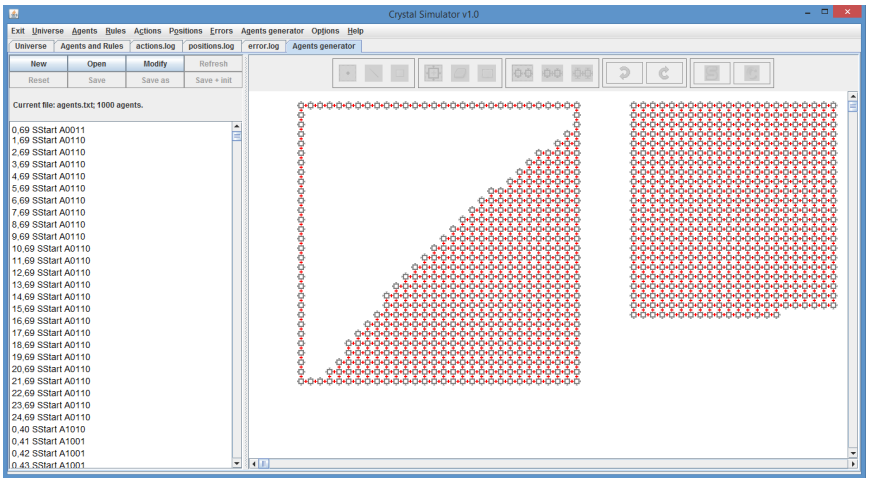


Figura 2.10: Pantalla de *Agents generator* del simulador.

2.3.7. Módulos

Como hemos mencionado anteriormente en este capítulo, el simulador muestra módulos de cuatro átomos. Todos los módulos ejecutan el mismo conjunto de reglas y tienen las mismas limitaciones. Cada módulo tiene cuatro caras (norte, oeste, este y sur), un color, un número de identificación único que es utilizado por el simulador pero no puede usarse en las reglas que siguen los algoritmos, una posición dada por dos coordenadas (x e y) que las reglas tampoco pueden usar, un registro que almacena los vecinos a los que esta conectado, un registro que guarda su estado actual, 26 registros internos de 16-bits en donde almacenar números enteros y un último registro que indica la prioridad del módulo a la hora de aplicar reglas. Cabe destacar que la posición de un módulo es única exceptuando el caso de un módulo anfitrión y un módulo comprimido, que comparten posición. Exceptuando su número de identificación y su orientación respecto el universo del simulador, todos los demás campos pueden alterarse durante la ejecución de un conjunto de reglas.

Cada módulo consta además de 8 canales por los que puede tanto recibir como enviar mensajes de texto y numéricos. Los mensajes de texto tienen un límite de 5 caracteres y los mensajes de texto solo pueden transmitir números enteros de 16-bits. Un módulo solo puede enviar mensajes a sus vecinos inmediatos de cada una de los cuatro puntos cardinales.

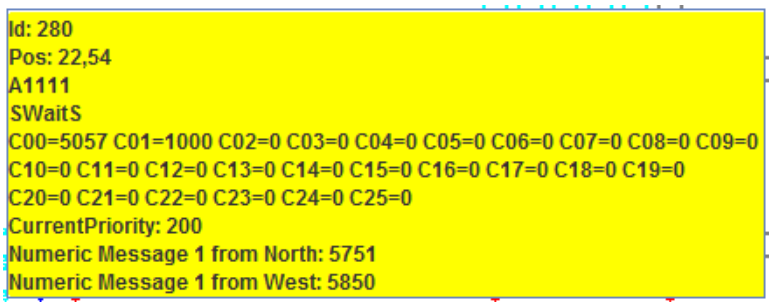


Figura 2.11: Imagen que muestra el simulador cuando se pulsa el botón derecho del ratón sobre uno de los módulos de la pantalla *Universe*. En esta imagen se puede ver toda la información contenida en el módulo: su id, su posición, los vecinos a los que está conectado, su estado, el contenido de sus 26 registros, su prioridad actual y los mensajes, indicando el canal por el que los ha recibido y su dirección de origen, que ha recibido en la iteración actual.

La Figura 2.11 muestra los datos de un módulo que el simulador muestra el pulsar el botón derecho sobre este.

Capítulo 3

Mejoras al algoritmo original

Aunque el algoritmo original es eficaz y resuelve la reconfiguración de robots cristalinos, no es todo lo eficiente que podría llegar a ser. Es por eso que hemos intentado mejorarlo y refinarlo para reducir en todo lo posible los recursos utilizados, como los movimientos de los módulos o el tiempo total de la reconfiguración.

Una de las mayores fuentes de movimientos innecesarios es el movimiento de módulos comprimidos hacia el líder cuando este vuelve de haber completado una rama del árbol generador de la configuración final. Para intentar evitar estos movimientos proponemos una serie de mejoras: señal de parada hasta intersección, señal de parada hasta raíz, señal de parada global y el uso de múltiples módulos líder de forma simultánea.

Las tres primeras modificaciones son versiones del mismo algoritmo. En cambio, la versión multilíder intenta aprovechar la capacidad de trabajo distribuido de los robots junto con lo aprendido en las anteriores modificaciones para sacar aún más partido a la reconfiguración.

A continuación, presentamos dichas modificaciones.

3.1. Algoritmo con señal de parada hasta intersección

3.1.1. Objetivo

El objetivo es reducir el número de movimientos innecesarios en la fase de construcción de la configuración final, por la vía de mantener en estado de inactividad todos los módulos entre un módulo hoja y la siguiente intersección o, de no haberla, hasta la misma raíz del árbol.

Para conseguirlo, se emite una señal de pausa desde el módulo hoja en dirección a la raíz, hasta los módulos mencionados anteriormente.

Además, un objetivo secundario es llevar esto a cabo sólo con el paso de señales y cambios de estado, sin contadores ni nuevas consultas a estados

vecinos.

3.1.2. Estrategia

Nuestro punto de partida en este caso es la iteración siguiente a aquella en que el líder alcanza la posición de un módulo hoja del árbol generador de la configuración final.

Con esto entendemos que el módulo hoja, que estaba expandiéndose, es el último módulo de su rama en la configuración final y que ya ha comprobado que no debe expandirse en ninguna dirección.

En este punto, como en el algoritmo original, el módulo hoja devuelve el estado de líder a su padre. Es en este mismo momento, al cambiar el padre su estado a *líder*, cuando el nuevo líder envía la señal de parada a su padre.

A partir de entonces, cuando un módulo recibe la señal de parada, entra en estado *pausa* y propaga el mismo mensaje que ha recibido en dirección a su padre.

En el momento que la señal llega a un módulo que aún debe expandirse en alguna dirección, o al módulo raíz del árbol, este ignora la señal de pausa, evitando así que se extienda más allá.

Con esto, se crea un camino sin bifurcaciones desde el líder hasta el ya mencionado módulo hoja, todo compuesto de módulos en estado de pausa.

Estos módulos en estado *pausa*, así como los módulos que llevan otro módulo comprimido en su interior, permanecen quietos, sin realizar acción alguna, hasta que el líder les cede su estado de *líder*. Sabemos que todo módulo en pausa acaba recibiendo el estado de líder ya que este se desplaza en la misma dirección que la señal de pausa, en la dirección al módulo raíz, tal como dicta el algoritmo original.

De este modo evitamos que un cierto número de módulos transiten hasta la hoja para luego dar media vuelta y deshacer parte del camino recorrido.

3.1.3. Reglas

Las reglas que se han utilizado o modificado pertenecen todas al grupo [E], más específicamente, dentro de este grupo, a las reglas de retorno del líder al padre y las de cambio de líder de un módulo en fase de expansión a otro en fase de compresión (expansión de una rama usando módulos que estén en el camino del líder).

Las señales de esta categoría (menos *[E]Receive Signal Root Lider*), se han modificado para que propaguen la señal numérica de pausa (*una señal numérica con valor 9999*) en dirección al padre del módulo que las aplica. Otra modificación es la condición en las reglas para poder aplicarlas sobre los módulos en estado *pausa*.

Además, se han añadido una serie de reglas para extender la señal de pausa, una vez recibida, en la dirección del padre. Estas reglas, al mismo

tiempo, cambian el estado del módulo que las aplica a *pausa*. Están preparadas para que no se puedan aplicar ni a un módulo que aún deba expandirse en alguna dirección ni a la raíz.

Por último se ha modificado la función del registro C020 así como todas las reglas de expansión de rama mediante módulos en fase de compresión (paso del estado *líder* de un módulo en fase de expansión a otro en fase de compresión). Todas estas modificaciones se deben a un error del algoritmo original, el cual perdía el módulo líder al intentar pasar este estado de un módulo en fase de expansión a una hoja de una rama en fase de compresión que ya se había comprometido a comprimirse. Estas modificaciones se describen con más detalle en el Apartado (3.1.4).

3.1.4. Problemas

Prioridad de las reglas

Aunque puede no llegar a considerarse un problema de prioridades, si que es cierto que, al menos, deriva de ellas.

Al principio se planteó que la señal de pausa fuera un mensaje de texto, sin embargo, el hecho de tratar una señal de texto y no una numérica, alteraba ligeramente el orden en que el simulador evalúa y, por tanto, ejecuta las reglas.

Había ocasiones en que esta prioridad interrumpía un paso de módulo comprimido ya confirmado (con el mensaje *CANSZ*) entre dos módulos. Esto aparentemente no suponía ningún problema, es más, permitía ahorrar un movimiento extra, sin embargo, este suceso provocaba un error en el simulador, el cual dejaba de funcionar a las pocas iteraciones.

Al intentar cambiar la prioridad de estas reglas para evitar la colisión con las de paso de módulo comprimido tanto al aumentar como al disminuir dicho valor, las reglas dejaban de aplicarse cuando deben. Por tanto, bajar o subir la prioridad de las operaciones de pausa no era una opción, estas operaciones debían ejecutarse con la misma prioridad que el resto de operaciones de la categoría [E] para funcionar.

Al final, cambiando la señal de un mensaje de texto a uno numérico, la ejecución de una regla de pausa frente a una de paso de módulo comprimido quedaba relegada a un segundo puesto. De esta forma, si un módulo debe entrar en estado *pausa* pero ya ha confirmado una operación de paso de módulo comprimido, primero aceptará el nuevo módulo comprimido antes de entrar en pausa.

Esta modificación no afecta en absoluto al resto de reglas.

Perdida del módulo líder

Un error del algoritmo original permitía la desaparición del estado de líder y por tanto obligaba a la reconfiguración a terminar antes de tiempo.

El problema se daba cuando un módulo hoja en fase de compresión se comprimía en dirección a su padre y, durante la misma iteración en que se realizaba la compresión, un módulo líder vecino le intentaba mandar un mensaje de cambio de rama. Para entender la causa del problema hay que fijarse en las opciones del simulador bajo las que funciona tanto el algoritmo original como sus modificaciones. Las opciones especifican que primero se deben mover los módulos que deban moverse antes de enviar los mensajes. Esto quiere decir que primero se evalúan las normas a ejecutar en cada módulo, en este caso la compresión de la hoja y el envío del mensaje de cambio de rama, luego se mueven los módulos, acción que naturalmente comprimía la hoja, y para terminar se envían los mensajes haciendo que el líder enviara su mensaje a un espacio vacío para luego, sin siquiera esperar confirmación que indicase si se había recibido el mensaje, pasara a estado de *expansión*. El algoritmo original no estaba pensado para reaccionar ante esta situación.

Una primera solución fue la de intentar restaurar los valores de los registros del módulo líder cuando este detectara que el módulo hoja se había comprimido, pero el módulo líder no disponía de la información necesaria ni para poder restaurar sus registros ni para detectar el fallo. La siguiente solución que se intentó fue la de alargar en una iteración el proceso de compresión de todos los módulos hoja que tuvieran un líder por vecino para conseguir así recibir la señal de cambio de rama y, así, solucionar el problema.

Esta solución funcionó perfectamente para esta situación pero los cambios que se introdujeron en el algoritmo original para poder solucionar el problema interferían con algunas de las medidas que había implementado el autor del algoritmo original para solucionar los problemas de intentar expandir una rama pasando por una posición ocupada por una hoja que pretende comprimirse. Al no encontrar otra manera posible de solucionar el problema original se optó por buscar nuevas soluciones a este antiguo problema que habíamos reencontrado.

Antes de buscar estas nuevas soluciones decidimos que intentaríamos siempre que nos fuera posible mantener la prioridad de una compresión por encima de un cambio de rama. Si no fuera así, como nos habíamos visto obligados a aumentar la duración de la compresión y, por tanto, el tiempo que el padre de la hoja mantenía en sus registros el dato de que ya disponía de un módulo comprimido (el padre da por supuesto que se realizará la compresión al enviar la confirmación de compresión), podríamos provocar en caso de cancelar la compresión que el padre de la hoja intentara entregar un módulo comprimido que no tiene, parando así la reconfiguración. Una vez decidido esto nos dispusimos a afrontar el nuevo problema.

Este problema se daba cuando un módulo hoja en fase de compresión que había pedido permiso para comprimirse recibía una señal de cambio de rama (de una rama en compresión a una en expansión) emitida por el líder actual. En el momento en que el líder enviaba la señal de cambio de

rama este daba por sentado que el módulo hoja pasaría su estado a líder y que cambiaría de rama, por eso el líder cambiaba su estado a *expansión* (o *Expnd*). Una iteración más tarde, cuando la hoja había pasado a ser líder de la rama en expansión, recibía además el mensaje de que tenía permiso para comprimirse. Esto provocaba que el líder cambiara otra vez a estado de *compresión* (o *Cmprs*) y que se comprimiera en dirección a su antigua rama.

Para evitarlo se tomaron tres medidas: la primera, para ahorrar complicaciones en las reglas, fue incrementar permanentemente las iteraciones que tarda un módulo en ejecutar la orden de cambio de rama en una iteración, la segunda fue el uso del registro C16 para almacenar la dirección por la que se había recibido la señal de cambio de rama además de cambiar el registro 20 para almacenar en el líder hacia donde se ha enviado la señal de cambio de rama en vez de ser simplemente un booleano y la tercera fue la creación de una nueva señal de aviso.

Ahora, al encontrarnos en la misma situación que ocasionaba este problema, al recibir la señal de cambio de rama después de haber pedido permiso para comprimirse, el módulo hoja espera una iteración más para comprobar si recibe confirmación para la compresión. Si no la recibe se realiza el cambio de rama pero, en caso de recibirla, el módulo hoja envía un mensaje de advertencia usando la dirección del registro C16 para indicar al antiguo líder de que no se efectuará el cambio de rama y no ejecuta ninguna otra operación durante esa iteración. Al esperar una iteración y asegurarnos de que el mensaje llega al antiguo líder, ahora en estado de *expansión*, conseguimos sincronizar dos eventos, el de compresión de la hoja y el de la lectura del aviso de que no se efectuará el cambio de rama. Ahora el antiguo líder no solo recupera su estado de *líder* sino que, además, evita enviar otra señal de cambio de rama ya que, cuando trata el aviso enviado por la hoja, este sabe que no debe aplicar ninguna otra regla de expansión de la rama durante esa iteración.

Esta solución se tuvo que aplicar no solo a esta modificación del algoritmo sino también al algoritmo original, ya que le era imposible ejecutar satisfactoriamente alguno de los juegos de prueba usados en el estudio de las modificaciones.

3.1.5. Alternativas

La única alternativa planteada al crear esta mejora del algoritmo fue la de utilizar un mensaje de texto en vez de uno numérico para propagar la señal de pausa. Era una opción más clara de cara a un lector o a un programador humano. Sin embargo al ver los problemas que surgieron al utilizar texto se optó por otra solución.

Para arreglarlo, podríamos haber intentado modificar el simulador en lugar del tipo de mensaje, pero esta opción, además de lenta, podía aca-

rrear consecuencias no deseadas en otros algoritmos ya programados sobre el mismo simulador.

3.1.6. Modelos de prueba

Para comprobar el correcto funcionamiento de estas nuevas reglas se realizaron todo un conjunto de pruebas. Sin embargo, debido a la cantidad de casos necesarios para comprobar que el resto de reglas siguen funcionando correctamente pese a la aplicación de nuestras nuevas reglas, solo mostramos los modelos de prueba más significativos o que han dado más problemas.

Intersección de la señal de pausa y de paso de módulo comprimido

En el modelo de prueba ilustrado en la Figura 3.1 podemos observar cómo la señal de *pausa* llega a un módulo que ya se ha comprometido a aceptar un módulo comprimido, tal como se explica en el Apartado 3.1.4 de problemas de esta modificación. En la iteración siguiente, como intercambio de módulos tiene prioridad sobre el cambio a estado *pausa*, dicho módulo acepta primero el módulo comprimido y después cambia su estado.

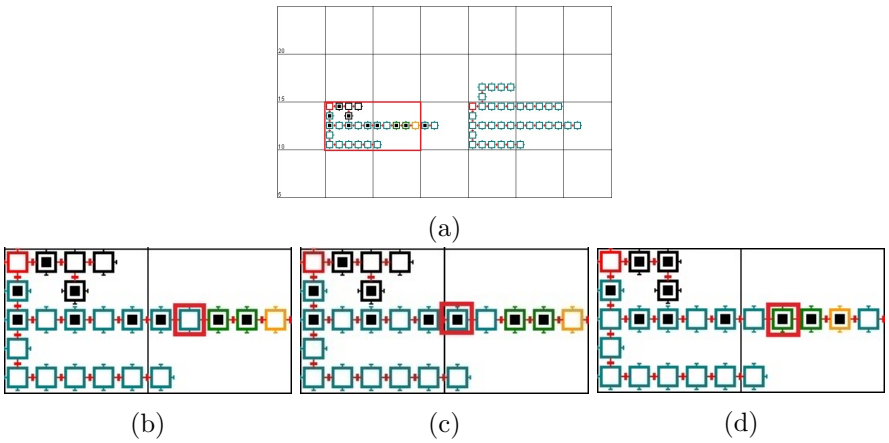


Figura 3.1: Partiendo del árbol actual y generador final mostrados en (a) en una misma iteración el módulo marcado en (b) recibe la señal de *pausa* del este justo cuando comunica a su vecino del oeste, señalado en (c), y se compromete a aceptar su módulo comprimido. Una iteración más tarde, tal como se ve en (d), el módulo mencionado en (b) primero ha aceptado el módulo comprimido del oeste y luego ha entrado en estado de *pausa* (indicado por su color verde).

3.2. Algoritmo con señal de parada hasta raíz

3.2.1. Objetivo

El objetivo de esta modificación es no solo la de evitar que los módulos comprimidos entren en una rama ya completada del árbol generador de la configuración final, sino también evitar que otros módulos pasen de un subárbol de la raíz al otro a no ser que sean necesarios.

Pretendemos así controlar aún más el movimiento de los módulos comprimidos, extendiendo la señal de pausa más allá de un cruce, llevándola hasta la misma raíz del árbol.

Como en el caso anterior, el objetivo es llevar esto a cabo sólo con el paso de señales y cambios de estado, sin contadores ni nuevas consultas a estados vecinos.

3.2.2. Estrategia

El punto de partida es el mismo que en la modificación anterior: la iteración siguiente a aquella en que el líder alcanza la posición de un módulo hoja que ha llegado al final de la configuración de una rama del árbol generador de la configuración final. La estrategia, esta vez, se divide en dos fases:

Transmisión y expansión del mensaje. El procedimiento es el mismo: al retornar la señal de líder en dirección a la raíz, extendemos la señal de pausa de un módulo a otro, de hijo a padre, en dirección a la raíz. Sin embargo, esta vez no detenemos la expansión de la señal al llegar a un cruce que no haya sido completado, sino que seguimos expandiendo la señal hasta llegar a la raíz, la cual ignora por completo la señal de pausa, evitando así que se extienda aún más.

Reactivar los módulos pausados. Esta fase puede llegar a no darse en algunas configuraciones, sin embargo, si entre la raíz del árbol y el módulo líder existe algún módulo que necesite expandir una rama, cuando este módulo obtenga el estado de *líder* e intente expandir el camino la configuración de los módulos quedará dividida en tres partes: entre la raíz del árbol y el módulo líder, compuesta por módulos en pausa, el subárbol que consta del módulo líder y todos sus hijos, y el conjunto de módulos restantes (si los hay), tanto en fase comprimida como ya ubicados en su posición final.

El subarbol del líder, ya parcialmente reconfigurado, puede que no contenga módulos suficientes en estado de compresión o expansión como para completar la rama que el módulo líder quiere expandir. Para solucionarlo, basta con enviar una señal de reanudación en el momento en que el líder intente expandirse y no tenga un módulo comprimido cerca para hacerlo. Esta señal se transmite de la misma forma que la señal de pausa, de hijo a padre, hasta la raíz.

De esta manera, el cuello de botella (módulos en estado de pausa), que bloquean el paso de módulos de una rama a otra se reactiva hasta que la

rama se completa y vuelve a emitir una señal de pausa.

Al final, cuando la señal de líder llega a la raíz no queda ningún módulo en pausa en la configuración.

3.2.3. Reglas

Las reglas que se han utilizado o cambiado para esta modificación pertenecen todas al grupo [E], más específicamente, a las reglas de retorno del líder al padre y a las de expansión. Las señales de este grupo se han cambiado para permitir aplicar el estado de *Pausa* a módulos intersección y se han añadido reglas extra para emitir la señal de reanudación (una señal numérica con valor *9998*).

3.2.4. Problemas

Cambio de estado de una intersección

Al modificar las normas de extensión de la señal de parada para permitir cambiar el estado de los módulo intersección a *Pause*, inicialmente no se previó que el paso de líder de un módulo simple a un módulo intersección tuviera precondiciones diferentes al caso del paso de líder de un módulo simple a otro. Es por esto que aunque el paso de líder de un módulo simple a otro también simple en estado *Pause* funcionaba, al encontrarnos con el paso de líder a un módulo intersección pausado el simulador se quedaba sin reglas que aplicar.

Para solucionarlo se modificaron las reglas de paso de estado de líder a un módulo intersección para que pudieran aplicarse también cuando dicho módulo estuviera en estado de *Pausa*.

Falta de módulos

En un principio no se previó la posibilidad de que el módulo líder que intentaba extender una rama no tuviera suficientes módulos comprimidos activos para acabarla. Esto ocurría porque la señal de pausa cerraba el paso a nuevos módulos muy por encima de dicha rama.

La solución fue sencilla, la creación de la señal de reanudación, que permitía activar el paso de módulos.

Sin embargo, esta solución deja otros problemas pendientes. Al no usar contadores, no disponemos de medios para contar cuántos módulos comprimidos activos hay en la configuración que puedan llegar a un líder en expansión y tampoco podemos saber si una rama de la raíz necesita más módulos para reconfigurarse o si, por el contrario, tiene de sobra. Estos problemas se solucionaron en modificaciones posteriores del algoritmo.

3.2.5. Alternativas

Se consideraron algunas alternativas como añadir contadores y consultas de estado, pero se optó por implementarlas en la última modificación.

3.2.6. Modelos de prueba

Para comprobar el correcto funcionamiento de estas nuevas reglas se realizaron todo un conjunto de pruebas. Sin embargo, debido a la cantidad de casos necesarios para comprobar que el resto de reglas siguen funcionando correctamente pese a la aplicación de nuestras nuevas reglas, solo mostramos los modelos de prueba más significativos o que han dado más problemas.

Cambio de estado de una intersección

En este modelo de pruebas se puede apreciar como un módulo intersección en estado de *pausa* recibe la señal de cambio a líder. Como se ve en la Figura 3.2, el problema de cambio de estado de una intersección descrito en el Apartado 3.2.4 ya no se produce y el módulo intersección pasa a tener el estado *líder*.

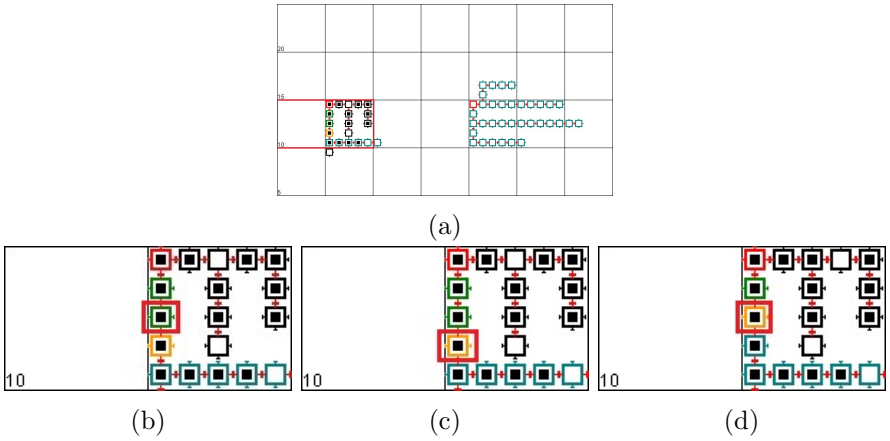


Figura 3.2: Partiendo del árbol actual y generador final mostrados en (a) podemos ver un módulo intersección que aún debe expandir una de sus ramas hacia el este y que además se encuentra en estado *pausa* (b). Este módulo recibe de su vecino del sur (marcado en (c)) una señal para cambiar su estado a *líder* y, como se puede ver en (d), una iteración más tarde el módulo mostrado en (b) ha cambiado su estado a *líder* con total normalidad.

Falta de módulos

En este apartado se muestra la solución aplicada al problema de falta de módulos descrito en el Apartado 3.2.4. En la Figura 3.3 podemos ver que

el módulo líder no dispone de módulos activos suficientes para completar una rama del árbol generador final. En este instante el módulo líder emite una señal de reanudación que se extiende de módulo a módulo permitiendo la entrada del número de módulos necesarios para alcanzar la configuración final. Debido a la falta de contadores, señal se emite siempre que un líder intenta expandir una rama desde un módulo que haya estado pausado, por los problemas mencionados en el Apartado 3.2.4.

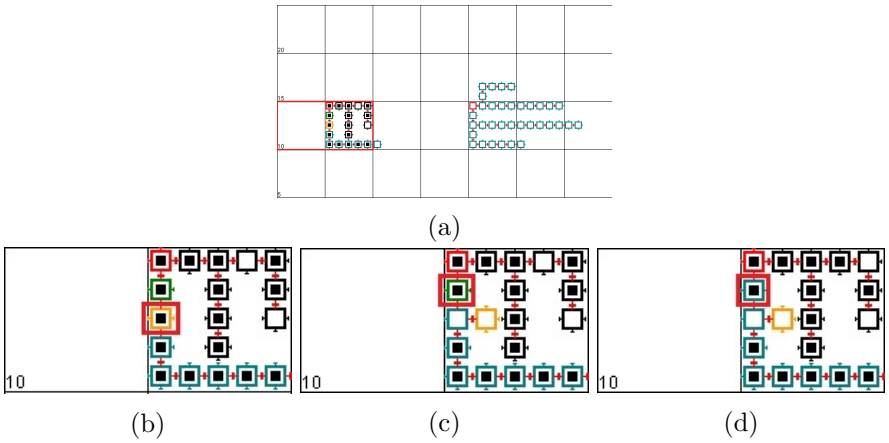


Figura 3.3: Partiendo del árbol actual y generador final mostrados en (a) vemos que el módulo en estado líder marcado en (b) aún debe expandir una de sus ramas hacia el este. En este caso no puede recibir módulos comprimidos de su vecino del norte ya que están en estado *pausa* y, aunque use todos los módulos comprimidos que le puedan llegar por su vecino del sur, e incluso teniendo en cuenta los módulos que pueda incorporar al expandirse, le será imposible completar la expansión de la rama. Para evitar situaciones como esta el módulo líder expande una señal de reanudación a su vecino del norte antes de expandirse (imagen (c)) y este vuelve a estado de expansión (indicado por el cambio de color de verde a azul) a la iteración siguiente (imagen (d)) volviendo a permitir el paso de módulos comprimidos.

3.3. Algoritmo con señal de parada para toda la configuración

3.3.1. Objetivo

Como en las dos modificaciones anteriores, el objetivo es el de intentar evitar al máximo los movimientos innecesarios en la fase de construcción de la configuración final, usando solo mensajes de pausa y cambios de estado, solo que esta vez lo que se busca es ver los beneficios y el coste de mantener en estado de pausa a todos los módulos de la configuración, incluyendo los

que están en estado de compresión.

3.3.2. Estrategia

La estrategia es análoga a las dos modificaciones anteriores, solo que esta vez extendemos la señal de pausa y reanudación no solo de hijo a padre, sino a absolutamente todos los módulos de la configuración.

Esto implica que las condiciones de emisión de las dos señales, pausa y reanudación, son las mismas que antes, sin embargo, a diferencia del anterior caso, no pararán de difundirse hasta que encuentren una hoja de la configuración.

Naturalmente, esto se hace evitando que la señal rebote indefinidamente entre padre e hijo. Tampoco existe ningún peligro de que una misma señal se expanda cíclicamente entre los módulos: al estar todos conectados en forma de árbol, no existen ciclos.

3.3.3. Reglas

Como esta modificación solo afecta a la fase de expansión del árbol, todas las normas afectadas son de la categoría [E]. Se ha creado una gran cantidad de reglas nuevas para poder pausar y reanudar los módulos tanto en expansión como en compresión.

Se han añadido varios estados nuevos como *PausR* o *PausC* para distinguir entre módulos pausados que tenían estado *Root* o *Comp* de los que estaban en expansión.

3.3.4. Problemas

Envío de señales a módulos desconectados

El simulador permite el paso de mensajes a módulos que están desconectados del emisor siempre que sean vecinos inmediatos, es decir, que estén a una unidad de distancia del emisor. En una primera implementación de esta modificación, las reglas enviaban las señales de parada y reanudación a hacia todas direcciones exceptuando la dirección por la que se había recibido dicha señal, con la idea de que si en una dirección no había ningún módulo conectado la señal no se propagaría hacia allí pero sí hacia las demás direcciones. Por supuesto, este conjunto de reglas fallaba por lo explicado al principio de este parágrafo.

Para solucionar este problema se creó una nueva regla por cada combinación posible de entrada/salida de señal (incluyendo dirección de entrada). Es decir, por cada dirección posible de entrada a un módulo de las señales de parada o reanudación, existe una norma para cada caso en que dicho módulo solo esté conectado a 2 módulos (un padre de donde viene la señal y

un hijo por donde enviarla), otra por cada caso en que en que esté conectado a 3 módulos y otra par cada caso en que esté conectado a 4 módulos.

Esta es una de las causas de que, de todos los conjuntos de reglas vistos hasta ahora, este es el más largo con diferencia.

Expansión de señal al terminar la reconfiguración

Una vez alcanzada la forma final al acabar la última rama, el padre de dicha hoja emite una señal de pausa. Esta señal se expande por toda la configuración. Incluso cuando se ha alcanzado la configuración final, si la señal aún no ha alcanzado todos los módulos seguirá propagándose, retrasando el final de la reconfiguración.

Sin el uso de contadores, este problema es imposible de solucionar, ya que no hay manera alguna de que los módulos sepan que han llegado al final de la reconfiguración y que por tanto ya no es necesario extender esta señal. Únicamente se pudo mitigar el problema eliminando reglas que permitían el paso de la señal de parada de la segunda rama de la raíz (la rama del Este) a la primera (la rama Sur).

3.3.5. Alternativas

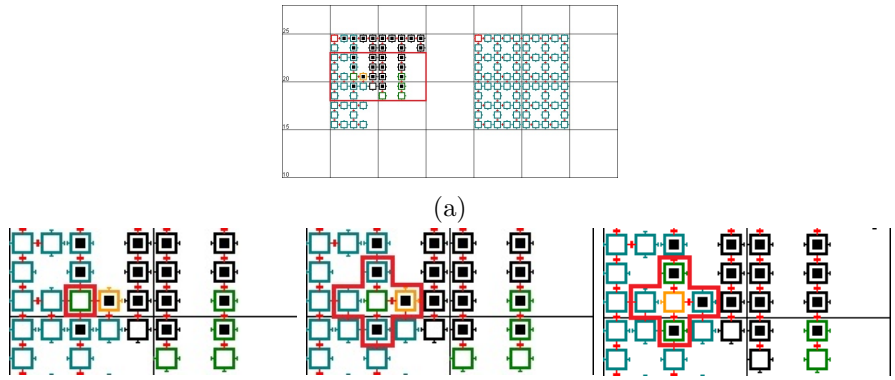
Se consideró la posibilidad de implementar un sistema de contadores en cada módulo que mantuvieran la cuenta de cuántos módulos se necesitan en cada dirección para alcanzar la reconfiguración final. Finalmente se implementó este sistema en el algoritmo multilíder.

3.3.6. Modelos de prueba

Para comprobar el correcto funcionamiento de estas nuevas reglas se realizaron todo un conjunto de pruebas. Sin embargo, debido a la cantidad de casos necesarios para comprobar que el resto de reglas siguen funcionando correctamente pese a la aplicación de nuestras nuevas reglas, solo mostramos los modelos de prueba más significativos o que han dado más problemas.

Envío de señales de pausa desde un módulo intersección

En este modelo de prueba, mostrado en la Figura 3.4, podemos ver un ejemplo del funcionamiento del conjunto de reglas en una situación como la descrita en envío de señales a módulos desconectados (Apartado 3.3.4). Cuando la señal llega a un módulo intersección vecino a otro módulo al que no está conectado, el conjunto de reglas distingue la situación según las conexiones y expande el mensaje de pausa en las direcciones necesarias.



(a) (b) (c) (d)

Figura 3.4: Partiendo del árbol actual y generador final mostrados en (a) podemos observar un caso como el descrito en el Apartado 3.3.4. En este ejemplo podemos ver en (b) un módulo en estado de *pausa* que tiene por vecinos a un módulo en estado *líder* y a tres módulos en estado de *expansión* (véase (c)). Como dicho módulo solo está conectado a dos de sus tres vecinos en expansión, solo envía dos mensajes de pausa, tal como se observa en (d).

Expansión de señal al terminar la reconfiguración

En la Figura 3.5 se muestra un caso de expansión de la señal de pausa una vez llegado al final de la reconfiguración (descrito en el Apartado 3.3.4). Como se puede apreciar en (a) y en (b), una vez acabada la reconfiguración se extiende una señal de *pausa*. Incluso cuando el líder vuelve a la raíz, la señal sigue expandiéndose como muestra (c), hasta pausar todos los módulos posibles como se puede ver en (d). Como ningún módulo sabe si la señal es necesaria o no, es imposible pararla. Sin embargo, como la rama sur de la raíz siempre acaba su reconfiguración antes que la rama este, la raíz no propaga la señal de parada por dicha rama, ahorrando así pasos de señal innecesarios.

3.4. Versión multilíder del algoritmo

3.4.1. Objetivo

El objetivo de esta modificación es el de reducir el número de movimientos innecesarios, mejorando la distribución de los módulos durante la fase de construcción de la reconfiguración final.

Para conseguirlo mejoramos el conocimiento de la configuración final que tiene cada módulo, y aplicamos cambios de estado y paso de señales parecidos a los usados en las modificaciones anteriores.

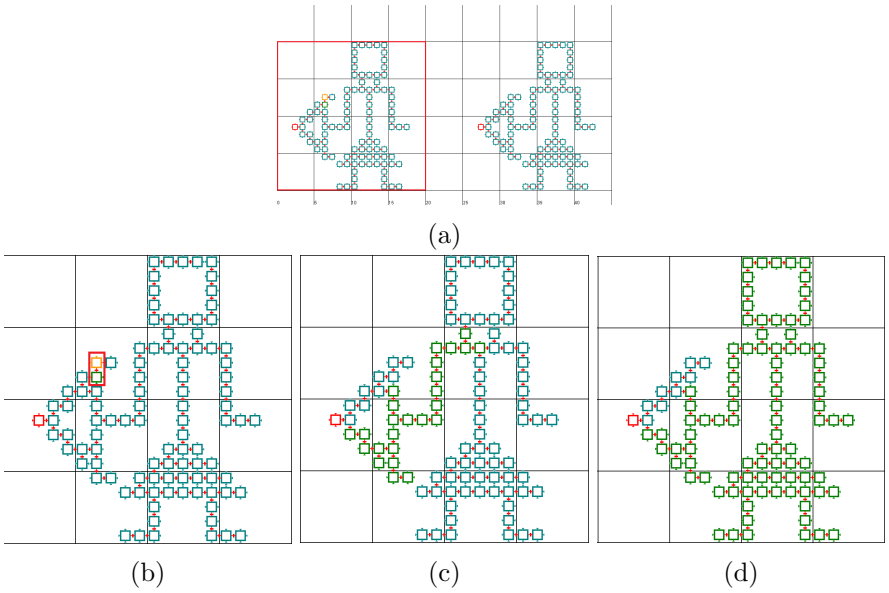


Figura 3.5: Partiendo del árbol actual y generador final mostrados en (a), el módulo líder emite una señal de pausado a su vecino del sur tal como muestra (b). Iteraciones más tarde, incluso cuando la señal de *líder* ha vuelto a la raíz, la señal sigue expandiéndose, véase (c), hasta llegar a todos los módulos posibles. Como se puede observar en (d) los únicos módulos que no quedan en estado de *pausa* son, en este caso, los módulos por los que ha vuelto la señal de *líder* hasta la raíz, reanudando los módulos a su paso.

3.4.2. Estrategia

En este caso, la estrategia depende de que en todo momento, desde el instante en que se crean los árboles generadores inicial y final hasta que finaliza la reconfiguración, cada módulo tanto del árbol generador actual como del final sepa exactamente cuántos módulos cuelgan de cada una de sus ramas. El primer recuento de módulos se realiza en la fase de búsqueda del módulo raíz ya que se aprovecha la señal *Back* que viaja de de las hojas a la raíz para contar los módulos de cada rama. Utilizando esta información, ya que todo líder se encuentra siempre en su posición final de la reconfiguración, cuando un módulo entra en estado *líder* puede comparar el número de módulos que cuelgan de sus ramas con su equivalente en el árbol generador de la configuración final para saber exactamente cuántos módulos le sobran o le hacen falta en cada dirección.

Esto permite coger módulos de las ramas en donde sobren y expandirlos en las direcciones que haga falta. Al expandir por primera vez en una nueva dirección las reglas dan el estado *líder* al primer módulo de la rama, creando así más de un líder y avanzando por múltiples ramas del árbol a la vez. Al repartir los módulos entre las direcciones en las que sean necesarias, las reglas

dan prioridad a las direcciones que necesitan mayor cantidad de módulos.

3.4.3. Reglas

En esta modificación se han editado prácticamente todas y cada una de las reglas de todos los grupos del algoritmo original para que acepten los nuevos cambios de estado, las nuevas señales de recuento de módulos y los efectos derivados de la presencia de múltiples módulos en estado *LIDER*.

Por supuesto, para implementar todos estos mecanismos se han creado una gran cantidad de nuevas reglas, haciendo de esta la modificación con mayor número de reglas de todo el proyecto.

3.4.4. Problemas

Intersección de señales

El algoritmo original no estaba pensado para funcionar con más de un líder activo y era este el que emitía la gran mayoría de las señales, por eso mismo era más sencillo saber cuándo y cómo cambiar de estado o realizar una acción. Ahora, con muchos módulos en estado *líder* expandiéndose a la vez y la gran cantidad de señales que viajan a través de los árboles generadores, la manera de interpretar las señales originales crea situaciones absurdas como la de dar el estado de *líder* a un módulo que ya se ha comprometido con otro a comprimirse o pedir a un módulo que expanda el módulo contenido en su interior cuando en realidad no tiene ninguno.

Estos problemas se deben principalmente a que un módulo no puede comprobar qué mensajes ha recibido o enviado su vecino y, por tanto, desconoce sus intenciones.

Para evitar estas situaciones se ha creado todo un conjunto de estados que expresan la situación de cada módulo y facilitan a los módulos el saber si pueden pedir algo a su vecino o no. De esta forma si un módulo quiere dar el estado de *líder* a otro que está en estado *SASKC1* (esperando a recibir confirmación de que puede comprimirse) el primero sabe que debe esperar a ver si su vecino vuelve al estado *Cmprs* (la compresión no ha podido llevarse a cabo) o si por el contrario se comprime.

Repeticiones cíclicas de un mismo conjunto de estados

La creación de un conjunto de estados que expresan la situación de cada módulo facilita mucho el control sobre cuándo aplicar un determinado tipo de normas o no, sin embargo también puede generar una repetición de cambios de estado que impide el avance de la reconfiguración.

Por ejemplo, un módulo líder puede estar esperando a enviar a su vecino una señal *DISAL* mientras este cambia entre los estados *SCmprs*, *SASKC1* y *SASKC2* de forma cíclica (probar a comprimirse, no recibir confirmación

y volver a intentarlo). Como al intento de enviar una señal *DISAL* le siguen un par de estados de espera de respuesta, y dicha señal solo puede aplicarse si el vecino está en estado *SCmprs*, es posible que siempre se intente enviar la señal cuando este se encuentra en estado *SASKC1* o *SASKC2*, atascando la reconfiguración.

Para solucionar este problema se han creado nuevos cambios de estado que permitan la salida de dichos bucles. En el caso comentado anteriormente, por ejemplo, al recibir la señal *DISAL* en el estado *SASKC1* o *SASKC2*, se pasa a *SCmD*1* o *SCmD*2* respectivamente (donde * es una de las cuatro direcciones) haciendo que después de *SCmD*2* se ejecute la acción provocada por la señal *DISAL* en vez de volver al estado *SCmprs*.

No perder la cuenta de los módulos robados

Como se ha mencionado en la sección *Estrategia* de esta modificación (Apartado 3.4.2) es esencial que cada módulo sepa cuántos módulos cuelgan de cada una de sus ramas. Esta tarea se ve dificultada por la acción *DISAL*, que ocurre cuando un líder decide incorporar un conjunto de módulos que cuelgan de otra rama diferente del árbol generador.

Eliminar un número determinado de módulos de una rama y añadirlos a otra afecta, obviamente, a la cantidad de módulos necesarios para llegar a la configuración final en dichas ramas y, por tanto, a la dirección y cantidad de módulos que circulan por estas. Para evitar que módulos innecesarios se muevan por una rama solo para encontrarse con que ya no son necesarios, se han creado un conjunto de señales como las de *pausa* o *reanudación* de las modificaciones anteriores, que informan a todos los módulos del árbol ubicados entre el módulo afectado y la raíz del árbol de que ese camino específico ha perdido/ganado un número determinado de módulos. Al recibir esta señal, los módulos actualizan los datos sobre los módulos que cuelgan de sus ramas, pudiendo cambiar así la dirección a la que envían o de la que reciben módulos.

Por supuesto esto no soluciona del todo los movimientos innecesarios ya que, hasta que no llega la señal de aviso, los módulos de la rama no saben nada sobre el cambio en el número necesario de módulos.

3.4.5. Alternativas

No se consideraron más alternativas para este algoritmo.

3.4.6. Modelos de prueba

Para comprobar el correcto funcionamiento de estas nuevas reglas se realizaron todo un conjunto de pruebas. Sin embargo, debido a la cantidad de casos necesarios para comprobar que el resto de reglas siguen funcionando

correctamente pese a la aplicación de nuestras nuevas reglas, solo mostramos los modelos de prueba más significativos o que han dado más problemas.

Cambio de estados e intersección de señales

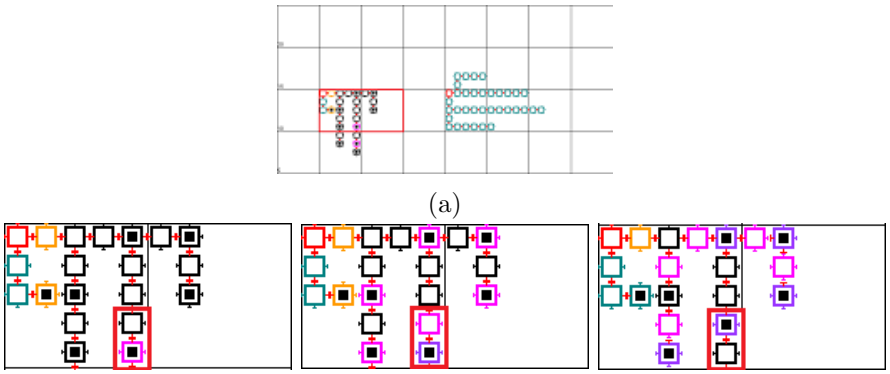


Figura 3.6: Partiendo del árbol actual y generador final mostrados en (a) vemos un par de módulos que acaban de empezar la acción de paso de módulo comprimido. Como podemos ver en (b) uno de los módulos ha enviado a su vecino del norte una señal para iniciar el paso de módulo comprimido y al hacerlo ha pasado a estado *ASKC1* (de color rosa). Una iteración más tarde, como muestra (c) el vecino del norte (ahora en estado *ASKC1*) ha aceptado la señal enviando un mensaje de confirmación al módulo que inició la comunicación, el cual ha pasado a estado *ASKC2* (color morado). Para finalizar el proceso, el módulo en estado *ASKC2* ha enviado el módulo comprimido a su vecino del norte y luego ha pasado a estado *Cmprs*. Dicho vecino acepta el módulo y pasa a estado *ASKC2*. Este último paso puede verse en (d).

En este modelo de prueba se pueden apreciar situaciones como las descritas en los problemas intersección de señales y repeticiones infinitas de un mismo conjunto de estados (ambos problemas están descritos en el Apartado 3.4.4). En la Figura 3.6 se puede ver como un módulo, al pedir permiso para pasar un módulo comprimido a un vecino, va cambiando de estado en cada iteración hasta recibir confirmación de que puede llevar a cabo la operación.

La Figura 3.7 muestra la intersección de una señal *DISAL* con un estado *ASKC1*. En este caso el módulo cambia al estado *CmDN1* para evitar caer en un bucle infinito como el descrito en los problemas citados anteriormente.

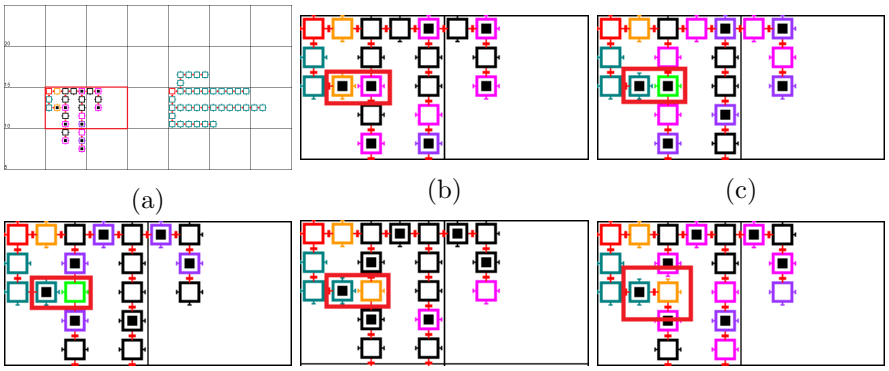


Figura 3.7: Partiendo del árbol actual y generador final mostrados en (a) y fijándonos más concretamente en los módulos señalados en (b), vemos un módulo en estado *ASKC1* (acaba de pedir a su vecino del norte que acepte su módulo comprimido) de color rosa que recibe una señal para desconectarse de su rama actual y conectarse a la rama de su vecino del oeste. Al recibir la señal, el módulo pasa a estado *CmDW1* y su vecino, como emisor de esta, pierde el estado de *líder* y pasa a estado de *expansión* tal y como se puede ver en (c) en donde los módulos tienen color verde y azul respectivamente. Además el módulo en verde ha recibido la confirmación de que puede enviar su módulo comprimido al norte. Una iteración más tarde el módulo en estado *CmDW1* envía su módulo comprimido y pasa a estado *CmDW2* (imagen (d)). Para acabar, el módulo pasa a estado *SDISAW* y luego se conecta a la rama de su vecino del oeste y pasa a estado de *líder* (imágenes (e) y (f) respectivamente).

Capítulo 4

Implementación del algoritmo multilíder

4.1. Árbol Inicial [S]

Este apartado describe el proceso mediante el cual se genera el árbol generador de la configuración inicial.

4.1.1. Inicio del algoritmo

Al principio todos los módulos de la configuración inicial deben encontrarse en estado *Start*. Una vez iniciado el algoritmo todo módulo que no se encuentre conectado a otro módulo ya sea en dirección este y/o sur se considera a si mismo como candidato a raíz del árbol y emite un mensaje, a través del canal *01*, a los vecinos a los que está conectado. Este mensaje indica la posición relativa del módulo vecino respecto al candidato a raíz. Es decir, considerando siempre el candidato a raíz como el centro de una cuadrícula de 100|$times$100, donde la raíz ocupa la posición (50,50), esta envía a su vecino del este la posición (51,50) y la posición (50,51) a su vecino sur. Esta posición relativa se guarda en la variable *C00* concatenando el valor de la coordenada x con el valor de la coordenada y. Es decir, si un módulo tiene una posición relativa (51,50), el valor de su registro *C00* es de 5150. Una vez enviado el mensaje, el candidato a raíz entra en estado *CanbS*.

4.1.2. Cadena de mensajes candidatos

Al recibir los mensajes de los candidatos a raíz con su posición relativa, un módulo trata los mensajes de la manera siguiente manera:

1. Si es el primer mensaje que recibe, guarda, en el registro *C01*, la dirección de la que ha recibido el mensaje como la dirección hacia la que

se encuentra la raíz, y envía a los vecinos a los que está conectado, exceptuando el vecino del que ha recibido el mensaje, su correspondiente posición relativa.

2. Si ha recibido algún otro mensaje anteriormente y el nuevo mensaje contiene un valor diferente de la posición relativa que el módulo ha almacenado, compara ambos valores, escoge el mejor de los dos (el que indique que la raíz está lo más al oeste posible y, en caso de empate, más al norte) y, en caso de escoger el nuevo mensaje, cambia su posición relativa, guarda la nueva dirección de la raíz y envía la nueva posición relativa a sus vecinos.

3. Por último, si ha recibido otro mensaje anteriormente y el nuevo mensaje contiene el mismo valor que el que el módulo tiene almacenado, evento que indica la existencia de un ciclo en el grafo de conexiones entre módulos, el módulo se desconecta de todos los vecinos que le han enviado el mismo mensaje menos uno. Este módulo al que permanece conectado se elije según prioridad de la dirección: norte >oeste >este >sur.

Una vez tratado el mensaje, el módulo entra en estado *WaitS*. Además, si un candidato a raíz recibe un mensaje de otro candidato a raíz, trata el mensaje como en el caso *2)* pues esta situación solo se da si existe un candidato a raíz mejor que él.

4.1.3. Mensaje recibido en las hojas

Una vez que un mensaje con una posición relativa llega a un módulo que no tiene a quién reenviarlo, ha llegado a una hoja del árbol, y esta empieza una nueva serie de mensajes en dirección a la raíz del árbol, o a lo que la hoja cree que es la raíz. Esta serie de mensajes consta de un mensaje de texto y otro numérico que son enviados al mismo tiempo: uno de ellos indica que se ha alcanzado una hoja (*MWBack_*) y el otro, enviado por el canal *01*, indica el número total de hijos del módulo que envía el mensaje (en caso de una hoja este valor es 0). Una vez que un módulo ha enviado estos dos mensajes en dirección a la raíz este entra en estado *BackS*.

4.1.4. Cadena de mensajes de las hojas

Cuando un módulo recibe un mensaje *Back_* quiere decir que el camino de la raíz a una de sus hojas se ha completado correctamente solucionando los ciclos que haya encontrado en su camino. Al recibir el mensaje, el módulo en cuestión guarda en *C02* la dirección del emisor. Es decir, al recibir el mensaje el módulo suma a su registro *C02* el valor 1000, 100, 10 o 1 dependiendo de si ha recibido el mensaje de su vecino del norte, oeste, este

o sur respectivamente. Una vez ha recibido el mensaje *Back_* de todos los módulos a los que está conectado exceptuando a su padre, del cual no puede llegar a recibirlo nunca, el registro *C02* contiene las direcciones de los hijos del módulo. En este momento, el módulo envía el mensaje *Back_* a su padre y sigue la cadena.

Además, como con cada mensaje *Back_* recibido el módulo también recibe una señal numérica indicando su número de descendientes en la dirección del emisor, el módulo guarda el valor de la señal numérica en su registro *C10*, *C11*, *C12* o *C13* dependiendo de si ha recibido el mensaje de su vecino del norte, oeste, este o sur respectivamente. A cada actualización de dichos registros, reescribe el valor del registro *C14* con la suma de sus descendientes en todas sus direcciones.

4.1.5. Creación de la raíz

Cuando un módulo candidato a raíz recibe el mensaje *Back_* de todos sus hijos entonces el candidato pasa a ser la raíz. En este momento el árbol generador inicial está completo, y cada módulo conoce el número de sus descendientes en cada una de las cuatro direcciones y todas las cadenas de mensajes *Back_* han llegado a la raíz. Una vez alcanzado este punto la raíz entra en estado *RootS* y envía un mensaje a sus hijos para iniciar la reconfiguración.

4.1.6. Conocer la configuración objetivo

Para simular la transmisión a los módulos de los datos necesarios sobre la configuración final que deben alcanzar, situamos una copia de dicha configuración objetivo en el mismo universo del simulador. Todos sus módulos se caracterizan por encontrarse en estado *Final*, y su módulo más oriental se encuentra a la misma altura (coordenada y) que el de la configuración inicial, y a su izquierda.

En el momento en que cambia a estado *RootS*, la raíz de la configuración inicial inicializa el valor de su registro *C23* a 0. Este registro se incrementa a cada iteración y contiene el valor de la distancia a la que se encuentra la raíz del árbol generador inicial de la raíz del árbol generador de la copia de la configuración final. Una vez que el registro *C23* alcanza el valor real de la distancia entre ambas raíces, este para de incrementarse y la raíz pasa a estado *RootL* para indicar que es raíz y líder al mismo tiempo. Este registro es la manera que tiene el algoritmo de simular que la raíz conoce todos los datos necesarios para completar la reconfiguración.

Asimismo, con el mismo objetivo, el conjunto de reglas descritas en los apartados 4.1.1 a 4.1.5 se aplica también a la copia de la configuración final, que se usa para simular la transmisión a la raíz de los datos de la configuración objetivo. En esta caso, sin embargo, la raíz pasa a estado

RootF y deja de actuar.

4.1.7. Cadena de mensajes Slave

En el instante en que se encuentra la raíz del árbol, esta envía a sus hijos el mensaje *Slave*, que indica a todo el que lo recibe que debe entrar en estado *Cmprs*. Esta cadena continúa pasando de padres a hijos hasta que alcanza las hojas del árbol generador inicial que, al no tener hijos, no pueden continuar la cadena.

4.2. Reglas de compresión [C]

Este apartado explica el funcionamiento de las reglas de compresión del algoritmo multilíder.

4.2.1. Compresión

Una vez la señal *Slave* ha llegado a una hoja, y esta pasa a estado *Cmprs* y, por tanto, entra en fase de compresión, se realiza la primera compresión física de un módulo. La compresión consta de tres pasos:

1. El módulo hoja envía a su padre un mensaje *ASK_Z*.
2. El padre, si no contiene ningún módulo comprimido ni ha recibido una señal *DISAL*, que se describe más adelante, contesta a su hijo con la señal *CAN_Z*.
3. Cuando la hoja recibe el mensaje *CAN_Z*, pasa a comprimirse dentro del padre en la iteración siguiente. Si no recibe el mensaje *CAN_Z* en dos iteraciones, vuelve al primer paso.

Además, cada vez que un módulo implicado en la compresión envía un mensaje *CAN_Z* o *ASK_Z*, para evitar interferencias al recibir otros mensajes y proteger la compresión, entra en un ciclo de estados en donde cada estado dura una iteración: *Cmprs* $\rightarrow$ *ASKC1* $\rightarrow$ *ASKC2* $\rightarrow$ *Cmprs*.

Una vez se ha comprimido la hoja, esta lo indica dando el valor 1 a sus registros *C24* y *C25*. El módulo padre, para indicar que contiene un módulo comprimido en su interior, da valor 1 a sus registros *C25* y *C15*.

Cuando un módulo recibe varias señales *ASK_Z*, de varios hijos diferentes, el módulo atiende las peticiones según la prioridad de compresión. Esta prioridad es, de mayor a menor: norte, oeste, este y sur.

Por supuesto, al comprimir un módulo y, por tanto, perder un descendiente, el padre de la hoja descuenta un descendiente de sus registros *C14* y *C10*, *C11*, *C12* o *C13* dependiendo de si la compresión se realiza por el norte, oeste, este o sur respectivamente.

4.2.2. Paso de módulos comprimidos en fase de compresión

Una vez comprimidos, los módulos viajan en dirección a la raíz del árbol. Para ello utilizan la operación de cambio de módulo comprimido, *SWZIP*.

Esta operación realiza los mismos pasos que la operación de compresión, pero ahora, utiliza el mensaje *ASKSZ* en lugar de *ASK_Z* y la señal de *CANSZ* en lugar de *CAN_Z*:

1. El módulo que desea pasar el módulo comprimido envía a su padre un mensaje *ASKSZ*.
2. El padre, si no contiene ningún módulo comprimido ni ha recibido una señal *DISAL*, contesta a su hijo con la señal *CANSZ*.
3. Cuando el hijo recibe el mensaje *CANSZ*, pasa a enviar el módulo comprimido a su padre en la iteración siguiente. Si no recibe el mensaje *CANSZ* en dos iteraciones, vuelve al primer paso.

Como en la operación de compresión, al enviar cualesquiera de los dos mensajes, los módulos implicados en la operación entran en el mismo ciclo de protección de la compresión: *Cmprs* $\rightarrow$ *ASKC1* $\rightarrow$ *ASKC2* $\rightarrow$ *Cmprs*.

Aunque el módulo padre recibe el módulo comprimido mientras está en estado *ASKC2*, no permitimos que ni el padre ni el hijo que envía el módulo comprimido ejecute ninguna otra regla hasta no estar ambos en estado *Cmprs*, aumentando el número de iteraciones entre movimientos de un mismo módulo comprimido a cuatro. Estas cuatro iteraciones en lugar de las tres que dura el paso de módulos comprimidos en una rama en fase de expansión son necesarias para poder reaccionar correctamente a las operaciones de cambio de rama sin perder la cuenta de los descendientes y ascendientes de cada módulo.

Esta operación de paso de módulo comprimido tiene menos prioridad que la operación de compresión. Es decir, si un módulo recibe al mismo tiempo la señal *ASK_Z* y la señal *ASKSZ*, el módulo atiende entes la señal *ASK_Z* sin importar la dirección de la que proceda.

Al pasar un módulo y, por tanto, perder su módulo comprimido, el emisor del módulo pone a 0 el valor de sus registros *C15* y *C25*.

4.3. Reglas de expansión [E]

Este apartado describe el funcionamiento de las reglas de expansión del algoritmo multilíder.

Cabe destacar que la fase de expansión y la fase de compresión coinciden en el tiempo durante parte de la ejecución, por lo que es necesario tener en cuenta la interferencia de reglas de ambas fases. Es por esta razón por lo que se ha decidido dar a las reglas de la fase de expansión prioridad sobre las reglas de la fase de compresión, simplemente para evitar interferencias.

4.3.1. Expansión del líder

Una vez la raíz ha recibido los datos del árbol generador final, es decir, cuando ha entrado en estado *RootL*, y un módulo comprimido ha llegado a la raíz, empieza la fase de expansión.

El primer paso que realiza la raíz es el de propagar la fase de expansión por las ramas indicadas por la información del árbol generador final. Esta información puede provocar dos situaciones distintas: o la raíz no tiene descendientes en la dirección indicada (la rama no existe), en cuyo caso hay que usar el módulo comprimido contenido en la raíz para expandir la rama, o la raíz ya tiene descendientes y puede simplemente propagar la señal de *líder* en esa dirección.

Al ser este un algoritmo que acepta más de un módulo en estado *líder* al mismo tiempo, si la raíz del árbol generador final tiene descendientes tanto al sur como el este, la raíz del árbol generador actual expande ambas ramas al mismo tiempo generando situaciones en las que ambas ramas ya existan o que una rama exista y la otra no. Jamás se da una situación en la que ambas ramas no existan ya que, por fuerza, el módulo raíz estaba conectado, al menos, a la rama por la que ha recibido el módulo comprimido.

En el caso en que no existan descendientes en la dirección en la que debería expandirse una rama, la raíz del árbol generador actual expande el módulo comprimido que contiene en su interior en la dirección que corresponda (realizando un movimiento *UNZIP*) y le otorga el estado de *líder* informándole a la vez de que ya no esta comprimido mediante un mensaje numérico por el canal *02*. Si por el contrario ya existen descendientes de la raíz en la dirección en la que debe expandirse el árbol, simplemente se le otorga el estado de *líder* al hijo de la raíz en esa dirección. Al enviar el mensaje *LIDER* a sus hijos para que pasen a estado *LIDER*, el módulo raíz da el valor 1 a su registro *C20* para proteger el paso del estado de *líder* evitando la ejecución de otras reglas.

Independientemente del caso, los valores del módulo que ha pasado a estado *LIDER* se actualizan con la información del árbol generador final:

- *C02* pasa a tener valor 0 ya que el módulo líder actual no tiene ningún hijo en estado de expansión.

- *C04* indica el número de hijos del líder actual que aún necesitan expandirse y la dirección de estos. El valor del registro se consigue sumando al registro 1000, 100, 10 y 1 si tiene hijos al norte, oeste, este o sur respectivamente.

- *C05* recibe el valor de la dirección a la que el módulo debe enviar los módulos comprimidos que reciba. La dirección se indica de la misma manera que en el caso del registro *C04*.

- *C16*, *C17*, *C18* y *C19* indican el número de descendientes o ascendientes necesarios en dirección norte, oeste, este y sur, respectivamente para completar la expansión de una o más ramas. El valor es 0 si no se necesitan más módulos, negativo si sobran módulos en una determinada dirección y positivo si se necesitan más módulos.

Una vez se han creado los primeros módulos líder, estos a su vez siguen transmitiendo el estado de *líder* según los parámetros del árbol generador final.

4.3.2. Expansión a una posición ocupada conexa

Análogamente al caso de la raíz, los módulos, en estado *LIDER* o *Expnd*, que quieren expandir este estado a un vecino al que están conectados, solo tienen que realizar los mismos pasos que en el caso de la expansión de la raíz por descendientes ya existentes. Una vez se emite el mensaje *LIDER*, se actualiza el valor del registro *02* del emisor, para que registre la dirección en la que ha expandido un hijo, y el valor de su registro *04*, también del emisor, para que elimine la dirección por la que ha expandido un hijo. Además, al actualizar dichos registros, si el módulo se encontraba en estado *LIDER*, pasa a tener el estado *Expnd*.

Cada vez que un módulo expande uno de sus hijos, este suma 1 al valor de su registro *C08*, que es un contador que indica el número de hijos que ha expandido un módulo y que permite saber.

4.3.3. Expansión a una posición vacía

Como en el caso de la expansión de la raíz, es común encontrar que un módulo, en estado *LIDER* o *Expnd*, considera que debe expandirse a una posición vacía de la cuadrícula. En estos casos, se realizan los mismos pasos descritos para el caso de la expansión de la raíz a una posición vacía. Además, una vez se emite el mensaje *LIDER*, se actualiza el valor del registro *02* del emisor, para que registre la dirección en la que ha expandido un hijo, y el valor del registro *04*, también del emisor, para que elimine la dirección por la que ha expandido un hijo. Al actualizar dichos registros el módulo pasa a tener el estado *PExpn* durante una iteración para, a la iteración siguiente, pasar a estado *Expnd*.

Cada vez que un módulo expande uno de sus hijos, este suma 1 al valor de su registro *C08*.

4.3.4. Expansión a una posición ocupada no conexa

En ocasiones, un módulo, en estado *LIDER* o *Expnd*, debe expandir una rama en una posición en la que se encuentra un módulo en fase de compresión

al que no está conectado. En estos casos se realiza una operación de cambio de rama. La operación del cambio de rama consta de los pasos siguientes:

1. El módulo que desea expandir una de sus ramas envía el mensaje *DISAL* a su vecino y, si no se encontraba ya en ese estado, pasa a estado *Expnd*.

2. El vecino recibe el mensaje y, como le es posible, realiza la operación de cambio de rama, recibiendo el estado *LIDER* y actualizando sus datos.

3. El vecino, ahora líder de la rama, envía al emisor de la señal *DISAL* un mensaje *EXPDL* para confirmar que se ha realizado la operación con éxito.

4. El líder de la rama emite una señal de recuento, actualizando los contadores de descendientes/ascendientes de todos los módulos de las rama que ha abandonado y a la que se ha unido.

Por supuesto, este es solo el caso en que la operación se ejecuta con éxito. Si el vecino que recibe la señal *DISAL* es una hoja, dependiendo de su estado es posible que la operación acabe de una forma distinta:

1. El módulo que desea expandir una de sus ramas envía el mensaje *DISAL* a su vecino y, si no se encontraba ya en ese estado, pasa a estado *Expnd*.

2. El vecino recibe el mensaje y considera que no le es posible realizar el cambio de rama.

3. El vecino envía al emisor de la señal *DISAL* el mensaje *NDISA*, indicando que no se ha realizado la operación de cambio de rama.

4. El emisor de la señal *DISAL* recupera su estado de líder y actualiza sus registros.

La razón por la que la operación puede no realizarse es la siguiente: si el módulo que recibe el mensaje de cambio de rama es una hoja esperando la respuesta de su padre para comprimirse, en estado *ASKC1* o *ASKC2*, no le es posible confirmar la operación de cambio de rama hasta saber si la respuesta de su padre llegará en las iteraciones previstas o no. En estos casos, para indicar que se ha recibido una señal *DISAL*, el ciclo de estados del módulo hoja pasa a ser *CmD*1* si se encontraba en estado *ASKC1* o *CmD*2* si se encontraba en estado *ASKC2* (el asterisco cambia según la dirección por la que se recibe la señal *DISAL*). Si durante las iteraciones en las que el módulo hoja se encuentra en estado *CmD*1* o *CmD*2* recibe el mensaje

CAN_Z de su padre, el módulo hoja entra en estado *ZIPNW*, para indicar que debe comprimirse en la siguiente iteración, y envía el mensaje *NDISAL* al emisor de la señal *DISAL*. Si no recibe el mensaje *CAN_Z* durante ese tiempo, la hoja realiza el cambio de rama.

En el caso en que el módulo que debe realizar el cambio de rama no es una hoja, la operación no puede generar un mensaje *NDISA*. Aún así, en este caso el ciclo de estados del módulo que debe realizar el cambio de rama se ve alterado de una forma parecida. Como en el caso anterior, el ciclo de estados del módulo pasa a ser *CmD*1* si se encontraba en estado *ASKC1* o *CmD*2* si se encontraba en estado *ASKC2*. Tanto si se recibe un mensaje *CANSZ* de su padre mientras se encuentra en estos dos estados como si no, el módulo pasa del estado *CmD*2* (o de *CmD*1* si recibe un mensaje *CANSZ*) a estado *DISAN*. La finalidad de este ciclo es la de evitar una repetición de estados infinita, en la que nunca se ejecute el cambio de rama y en su lugar se ejecuten siempre operaciones de paso de módulos comprimidos.

Cada vez que un módulo expande uno de sus hijos, suma 1 al valor de su registro *C08*.

4.3.5. Actualización de los registros contadores de módulos

Como consecuencia de una operación de cambio de rama, todos los módulos que pertenecen a las dos ramas implicadas en la operación actualizan una serie de registros con valores incorrectos. Estos registros son los que llevan la cuenta del número de descendientes de cada módulo y del número de módulos necesarios en cada dirección para completar la reconfiguración. Para volver a actualizar estos valores, el módulo que realiza la operación de cambio de rama emite cuatro cadenas de señales, todas ellas en dirección a la raíz del árbol generador actual.

Por la rama que ha dejado envía, a través del canal *08*, un mensaje numérico con valor *9999* y, por el canal *07*, el número de módulos que ha perdido la rama. El valor *9999* indica a los demás módulos que reciben el mensaje que, por la dirección por la que se ha recibido el mensaje, han perdido el número de módulos que se indica por el canal *07*. Una vez recibidos ambos mensajes, el valor indicado por el canal *07* se guarda en el registro *C06* para poder, una vez actualizados todos los registros, mandar el mismo valor al antecesor, siguiendo la cadena, hasta llegar a la raíz.

A través de la rama a la que se ha conectado, el módulo que ha realizado el cambio de rama envía, a través del canal *08*, un mensaje numérico con valor *9998* y, por el canal *07*, el número de módulos que ha ganado la rama. De esta forma, siguiendo la cadena del mismo modo descrito en el párrafo anterior, se actualizan todos los módulos de la rama hasta llegar a la raíz. La única diferencia es que el mensaje con valor *9998* indica que se han añadido módulos a la rama en lugar de haberlos perdido.

Si dos mensajes de este tipo, tanto de perdida de módulos como de

adición, coinciden a la vez en un mismo módulo, este calcula la diferencia, o la suma, y emite los mensajes correspondientes hacia la raíz. Si el valor de la diferencia es 0, no emite ningún mensaje hacia la raíz. Si la diferencia o la suma, dependiendo de si se han recibido dos mensajes de adición, dos de pérdida o uno de cada tipo, es negativa, el módulo envía el valor entero del resultado junto con la señal de pérdida, *9999*, hacia la raíz. Si el resultado es positivo, envía hacia la raíz la señal de adición, *9998*, junto con el resultado hacia.

4.3.6. Paso de módulos comprimidos en fase de expansión

Como en el caso de los módulos comprimidos en fase de compresión, los módulos comprimidos en fase de expansión también se mueven por las ramas del árbol generador actual. Aunque los dos casos son parecidos, existen varias diferencias.

Antes de continuar es necesaria una aclaración: distinguimos un módulo comprimido en fase de compresión de un módulo comprimido en fase de expansión simplemente por el estado en la que se encuentren los módulos que los contienen. Una vez dejado esto claro, podemos describir el procedimiento de paso de módulos comprimidos en fase de expansión.

1. El módulo que desea pasar el módulo comprimido envía a su padre un mensaje *EXPND*.

2. El padre, si no contiene ningún módulo comprimido, contesta a su hijo con la señal *CANEX*.

3. Cuando el hijo recibe el mensaje *CANEX*, envía el módulo comprimido a su padre en la iteración siguiente. Si no recibe el mensaje *CANEX* en dos iteraciones, vuelve al primer paso.

Si un módulo recibe más de un mensaje *EXPND* a la vez, decide que mensaje tratar según el orden de prioridad visto anteriormente. De mayor a menor prioridad: norte, oeste, este y sur.

Al enviar una señal *EXPND*, los módulos entran en un ciclo de estados que protege el paso de módulos comprimidos de interferencias de otras señales: *Expnd* $\rightarrow$ *ASKE1* $\rightarrow$ *ASKE2* $\rightarrow$ *Expnd*. Si un módulo recibe la señal *EXPND*, entra en un ciclo de estados distinto: *Expnd* $\rightarrow$ *ASKE2* $\rightarrow$ *Expnd*. A diferencia del caso en fase de compresión, el número de iteraciones entre movimientos de un mismo módulo comprimido es de tres iteraciones en lugar de cuatro. En la fase de expansión, al no existir la amenaza de las operaciones de cambio de rama, ya que esta operación siempre la realiza un módulo en fase de compresión, el algoritmo puede permitirse mover módulos comprimidos por las ramas del árbol generador actual de forma más rápida.

Por eso el módulo que recibe la señal *EXPND* entra en estado *ASKE2* en lugar de pasar por *ASKE1*.

En el caso en que se trata de pasar un módulo en fase de compresión a otro en fase de expansión, el módulo en fase de compresión realiza las mismas acciones que su equivalente en la sección 4.2.2, mientras que el módulo en fase de expansión entra, al recibir la señal *ASKSZ*, en el ciclo de estados *Expnd* → *ASKE1* → *ASKE2* → *Expnd*, haciendo que el paso de este módulo tarde en ejecutarse cuatro iteraciones en lugar de tres.

Como se ha mencionado anteriormente, al obtener un módulo comprimido, el módulo que lo recibe da valor 1 a sus registros *C15* y *C25* mientras que el módulo que envía el módulo comprimido da, a los mismos registros, el valor 0.

4.3.7. Dirección de viaje de un módulo comprimido

A diferencia de la fase de compresión, en la fase de expansión los módulos comprimidos viajan por las ramas no en dirección a la raíz o a un líder concreto, sino que viajan hacia donde se les necesita. Para saber dónde hacen falta estos módulos comprimidos, se utilizan los registros *C16*, *C17*, *C18* y *C19*. Esencialmente, estos registros guardan la diferencia entre el número de descendientes o ascendientes de un módulo en fase de expansión respecto a los que debería tener según los datos del árbol generador final.

La dirección a la que se envían los módulos comprimidos que recibe un módulo en fase de expansión esta indicada por el registro *C05*. Este registro señala la dirección del registro con mayor valor de los cuatro mencionados anteriormente. La dirección se indica con los valores vistos anteriormente para estos casos: 1000, 100, 10 y 1 para el norte, oeste, este y sur respectivamente.

En caso de empate en el valor de los registros, se sigue la prioridad de expansión. De mayor a menor prioridad: norte, oeste, este y sur.

Por supuesto, tanto el paso de módulos comprimidos como los cambios de rama alteran el valor de estos registros.

4.3.8. Retorno del líder

Una vez alcanzada la hoja de una rama del árbol generador final, la señal de *líder* vuelve a la raíz. Sin embargo, una vez pasada de hijo a padre, el módulo padre jamás enviara la señal hacia la raíz hasta no estar seguro de haber recibido las señales *líder* de todos sus hijos. Esto se controla gracias a que por cada módulo expandido hemos sumado 1 al valor del registro *C08* del padre del módulo expandido. De esta forma, por cada señal de *líder* que vuelva de un hijo, se resta 1 del registro *C08* del padre. Una vez que este registro alcanza el valor 0, el módulo envía la señal de *líder* a su padre, en dirección a la raíz.

El retorno de una señal de *líder* se indica mediante una señal con valor 1 enviada por el canal *C01*.

4.4. Fin de la reconfiguración [End]

Una vez un módulo ha enviado de vuelta a la raíz la señal de *líder* y ya no debe enviar más módulos comprimidos, es decir, el valor de sus registros *C15*, *C16*, *C17*, *C18*, *C19*, *C08* es 0, el módulo pasa a estado *DONEW*. En este estado, como el módulo ya sabe que no debe realizar ningún otro trabajo, busca entre sus vecinos inmediatos, aquellos que están a una posición de distancia en cualquiera de sus cuatro direcciones, aquellos en estado *DONEW* y, si no esta conectado a ellos, se conecta.

De esta forma, la reconfiguración acaba con una estructura con la misma forma que el árbol generador final pero con todos los módulos conectados entre sí.

4.4.1. Reglas de reparación

En este algoritmo, las únicas reglas consideradas como reglas de reparación son las que deciden el valor del registro *C05* (Apartado 4.3.7) ya que dependen del valor de otros registros y no de una fase concreta de la reconfiguración.

Capítulo 5

Complejidad de los algoritmos y análisis experimental

l objetivo principal de este proyecto es el de hacer más eficiente el algoritmo de reconfiguración de robots cristalinos, y para ello se han presentado diferentes modificaciones y versiones del algoritmo de reconfiguración original. Para ser capaces de estimar si una modificación ha mejorado o no el algoritmo original se ha realizado un estudio teórico de las modificaciones y otro experimental.

A continuación presentamos ambos estudios seguidos de las conclusiones a las que estos nos han permitido llegar.

5.1. Complejidad de los algoritmos

El estudio teórico del algoritmo se ha realizado comparando el coste o la mejora que supone una modificación respecto al algoritmo original. Gracias a este estudio hemos podido estimar el coste adicional en número de mensajes enviados que genera cada modificación así como el número de movimientos innecesarios ahorrados.

5.1.1. Algoritmo con señal de parada hasta intersección

Esta modificación extiende una señal de *pausa* desde una hoja de una rama que ha alcanzado su forma final hasta el primer módulo intersección que aún necesite expandirse. De esta manera busca detener cuanto antes los módulos comprimidos que viajan en dirección a la hoja y que ya no son necesarios para expandir la rama por la que viajan. Esta señal de *pausa* es mucho más rápida que el paso del estado *líder*, el encargado de actualizar los datos de cada módulo para decidir hacia dónde debe expandirse el árbol

generador actual, y por tanto se espera que sea capaz de evitar que estos módulos innecesarios se muevan hasta que el módulo que los contiene reciba el estado de *líder* y pueda decidir la nueva dirección hacia la que viajar.

Proposición 5.1 *El algoritmo con señal de parada hasta la intersección aplicado a un árbol generador inicial de N módulos y a un árbol generador final de H hojas envía $N - H - 1$ mensajes de pausa: $O(1)$ mensajes de pausa por módulo y $O(N)$ mensajes de pausa en total.*

Demostración:

El coste adicional que causa la modificación en una misma reconfiguración depende del número de mensajes de pausa enviados. Cada módulo, en una misma reconfiguración, extenderá una sola vez la señal de pausa, justo durante el momento en que la rama de la que forma parte ha alcanzado su forma final. En concreto, como no es el módulo hoja el que emite la señal de *pausa* sino su padre y teniendo en cuenta que el módulo raíz tampoco emite la señal, el número total de señales de *pausa* emitidas en una sola reconfiguración es:

$$Total_{pausa} = N - H - 1$$

Por tanto, el coste adicional por cada módulo que no sea la raíz del árbol generador actual ni sea una de las hojas del árbol generador actual una vez acabada la reconfiguración es de $O(1)$ mensajes y el coste total es de $O(N)$ mensajes de *pausa*.

□

El número de movimientos ahorrados por módulo comprimido que viaje por una rama que acaba de alcanzar su forma final es algo más complicado de deducir. Para llegar a cuantificar el ahorro, primero hay que entender lo que tarda el algoritmo original a reaccionar ante una rama que ha llegado a su forma final.

Proposición 5.2 *En el algoritmo original, si tenemos una rama de un módulo intersección de tamaño N_r y un módulo comprimido parte de la intersección en dirección a la hoja de la rama en el mismo instante en que dicha hoja emite una señal de líder, ambos elementos, módulo y señal, se encontrarán cuando el módulo comprimido haya recorrido aproximadamente el 40 % de N_r.*

Demostración: Imaginemos el caso en que un módulo comprimido está contenido en un módulo intersección que aún necesita expandir una de sus ramas mientras que otra de sus ramas acaba de alcanzar su forma final. Como la intersección aún no sabe que ya no hay que enviar módulos comprimidos por la rama que acaba de completarse, envía el módulo comprimido en dirección a la hoja de la rama a una velocidad de un módulo cada tres iteraciones. En

ese mismo instante la hoja envía la señal de *líder* hacia la intersección a una velocidad de un módulo cada dos iteraciones. Si la distancia entre la intersección y la hoja es de N_r módulos podemos ver la distancia que recorren el módulo ($D_{módulo}$) y la señal ($D_{líder}$) antes de encontrarse y el número de iteraciones (T) que tardarán en hacerlo:

$$Tiempo = Distancia/Velocidad \Rightarrow$$

$$N_r = D_{módulo} + D_{líder} = T\frac{1}{3} + T\frac{1}{2} = T\frac{5}{6}$$

$$T = \frac{5}{6}N_r = 1{,}2N_r$$

$$D_{módulo} = \frac{1}{3}T = \frac{2}{5}N_r; D_{líder} = \frac{1}{2}T = \frac{3}{5}N_r$$

Así podemos ver fácilmente que cuando la señal de *líder* y el módulo comprimido se cruzan, momento en el cual el módulo comprimido descubre que debe dar media vuelta, la señal ha recorrido el 60 % de módulos de la rama mientras que el módulo comprimido ha recorrido el 40 % restante. Dependiendo del tamaño de la rama el número de movimientos del módulo comprimido o el número de módulos que avanza la señal puede variar en un movimiento más o menos, ya que un módulo comprimido solo puede realizar un número entero de movimientos. □

Proposición 5.3 *En el algoritmo con señal de parada hasta intersección, si tenemos una rama de un módulo intersección de tamaño N_r y un módulo comprimido parte de la intersección en dirección a la hoja de la rama en el mismo instante en que dicha hoja emite una señal de pausa, ambos elementos, módulo y señal, se encontrarán cuando el módulo comprimido haya recorrido aproximadamente el 25 % de N_r.*

Demostración: Si aplicamos el mismo razonamiento que en la demostración anterior fijándonos en la señal de *pausa* en lugar de en la señal de *líder* podremos calcular la mejora de nuestra modificación. En este caso la señal avanza a una velocidad de un módulo por iteración:

$$N_r = T(\frac{1}{3} + 1) = T\frac{4}{3}; T = \frac{3}{4}N_r$$

$$D_{módulo} = \frac{1}{3}T = \frac{1}{4}N_r; D_{pausa} = T = \frac{3}{4}N_r$$

Podemos ver que esta vez el módulo comprimido solo recorre el 25 % de la rama. □

Así, pues, el ahorro de movimientos entre el algoritmo original y el que tiene señal de parada hasta intersección puede ser de hasta 15 puntos porcentuales por rama.

Puesto que el número de movimientos ahorrados depende de la distancia entre el módulo comprimido y la señal de pausa, podemos deducir el número máximo de movimientos ahorrados por un módulo comprimido dentro de una rama.

Proposición 5.4 *Mediante el algoritmo con señal de parada hasta intersección, el número máximo de movimientos* Max_{ahorro} *que puede ahorrar un módulo durante una reconfiguración es* $2 * 0{,}15 * (N-1)$ *movimientos.*

Demostración: El máximo ahorro (Max_{ahorro}) al viajar por una rama se da cuando un módulo comprimido se encuentra en el módulo intersección, o en la raíz si no existe ninguno, y la señal de pausa aún se encuentra en la hoja, es decir, en el momento de mayor distancia entre el módulo comprimido y la señal de pausa, justo en el momento en que el módulo intenta entrar en la rama. Teniendo en cuenta que la rama más grande posible es de tamaño $N-1$, el número de módulos del árbol generador inicial menos la raíz, y que los movimientos que no realiza adentrándose en la rama son movimientos que tampoco tiene que realizar para volver a la intersección, entonces:

$$Max_{ahorro} = 2 * 0{,}15 * (N-1)$$

Con cada rama del árbol generador inicial que se expande el tamaño potencial que pueden alcanzar las siguientes ramas es menor. Por tanto, el ahorro máximo por módulo es $O(N)$:

$$\sum_{hojas} (2 * Distancia_{intersección-hoja} * 0{,}15) \leq 2 * 0{,}15 * (N-1)$$

□

Como hemos podido ver en la demostración, el número máximo de movimientos ahorrados por módulo comprimido puede llegar a ser $O(N)$, sin embargo aún no hemos dicho nada sobre el número de movimientos ahorrado de toda la reconfiguración.

Proposición 5.5 *El total de movimientos ahorrados en una ejecución del algoritmo con señal de parada hasta intersección es* $O(N^2)$.

Demostración: Teniendo en cuenta que existen árboles generadores finales que al ejecutar esta modificación consiguen que un número $O(N)$ de módulos se vean afectados por la señal de *pausa* y que el ahorro de un mismo módulo aunque se vea afectado varias veces por la señal de *pausa* nunca supera Max_{ahorro}, es posible expresar el ahorro total como:

$$Max_{ahorro} * N$$

Es decir, el ahorro de movimientos es acotado superiormente por $O(N^2)$. Cabe destacar que no hemos podido demostrar la exactitud de esta cota. □

5.1.2. Algoritmo con señal de parada hasta raíz

Al introducir una nueva señal, la señal de *reanudado*, y al aumentar la distancia de expansión de la señal de *pausa* es de esperar que el coste de comunicación del algoritmo aumente aunque, como veremos, el número de movimientos ahorrados también aumenta en consecuencia.

Proposición 5.6 *El algoritmo de señal de parada hasta la raíz aplicado a un árbol generador inicial de N módulos y a un árbol generador final de H hojas envía un total de $O(N^2)$ mensajes de pausa y reanudado.*

Demostración: Esta modificación, aunque parecida a la modificación de señal de parada hasta la intersección, se diferencia de ésta en que la señal de *pausa* viaja siempre desde la hoja de una rama que ha llegado a su forma final hasta la raíz del árbol. Eso quiere decir que en vez de emitir la señal de *pausa* una sola vez por módulo ahora existe un conjunto de módulos entre la raíz y el módulo intersección de la rama que acaba de completarse que emiten la señal de *pausa* una vez por cada rama del módulo intersección que se complete. Este coste se suma al calculado en la modificación anterior, ya que el resto de módulos siguen emitiendo la señal de *pausa* una sola vez por ejecución. Por supuesto la distancia entre el módulo intersección y la raíz varía por cada módulo intersección por lo que este incremento en el coste debe expresarse como la suma para todos los módulos intersección, llamemos k al número de ellos, de la distancia de la raíz a cada intersección ($Dist_i$) multiplicado por el número de ramas de la intersección (R_i). Si no existe ninguna intersección en el árbol, ningún módulo con más de un hijo, el valor de esta suma es 0.

Para facilitar el cálculo del número de mensajes de *pausa* enviados, tratamos cada módulo intersección como si fuera la única intersección del árbol, es decir, solo tenemos en cuenta el módulo intersección más la suma de los módulos de sus ramas (N_i), la distancia de la raíz del árbol a la intersección ($Dist_i$) y el número de ramas de la intersección (R_i). Los módulos que unen varios módulos intersección no se consideran parte de una intersección ni rama de ninguna intersección y solo se tienen en cuenta para calcular ($Dist_i$). En definitiva:

$$Mensajes_{pausa} = \sum_{i=1}^{k}(N_i - R_i - 1 + Dist_i * R_i)$$

Además de los mensajes de la señal de *pausa*, esta modificación del algoritmo introduce la señal de *reanudado* que permite reanudar los módulos pausados, los mismos que emiten la señal de pausa desde la intersección hasta la raíz, sin necesidad de que intervenga la señal de *líder*. La señal de *reanudado* se emite antes de expandir una rama, siempre y cuando se haya emitido una señal de *pausa* antes en la reconfiguración. Además esta señal nunca se emite durante la expansión de la primera rama de cada uno de

los 2 subárboles posibles que nacen directamente de la raíz, o lo que es lo mismo, de los 2 posibles hijos de la raíz ($Hijos_{raíz}$). El coste total del envío de las dos señales, *pausa* y *reanudado*, es pues:

$$Mensajes_{Total} = \sum_{i=1}^{k}(N_i - R_1 - 1) + 2 * \sum_{i=1}^{k}(Dist_i * R_i) - Hijos_{raíz}$$

Sabiendo que la señal de *pausa* se emite una vez por cada una de las ramas del árbol generador final, y que la señal de *reanudado* se emite una vez por rama menos $Hijos_{raíz}$, podemos concluir que el coste por módulo de el envío de ambas señales es $O(R)$ en ambos casos, donde R es el número de hojas del árbol final.

Dado que el número total de señales enviadas depende del número de hojas del árbol final, podemos concluir que cuando el árbol final tenga el máximo número de ramas posibles, que es $O(N)$, las señales enviadas serán $O(N^2)$. □

El ahorro de movimientos innecesarios tanto por módulo como en total es igual que el de la modificación anterior del algoritmo, $O(N)$ y $O(N^2)$ respectivamente, ya que para ahorrar movimientos se usa la misma estrategia, la señal de *pausa*, y los módulos comprimidos siguen viajando a la misma velocidad que en la modificación anterior, la señal de *reanudado* no afecta al ahorro sino a la expansión y el máximo ahorro se da cuando todas las ramas tienen el tamaño máximo posible. Sin embargo los resultados prácticos de esta modificación serán aún mejores que los de la anterior debido a que ahora restringimos el acceso a todo un subárbol de la raíz del árbol generador actual en lugar de solo a una rama.

5.1.3. Algoritmo con señal de parada para toda la configuración

El análisis teórico de esta modificación del algoritmo es bastante más sencillo que en las modificaciones anteriores.

Proposición 5.7 *El algoritmo con señal de parada para toda la configuración aplicado a un árbol generador inicial de N módulos y a un árbol generador final de H hojas envía $O(N^2)$ mensajes de pausa y reanudado.*

Demostración: Esta versión del algoritmo se diferencia de la versión con señal de parada hasta la raíz en que sus señales de *pausa* y *reanudado* se expanden por todo el árbol, incluyendo los módulos en fase de compresión y las ramas del árbol generador actual que ya se hayan expandido por lo que, esencialmente, cada módulo emite una señal de *pausa* y una señal de *reanudado* por cada hoja del árbol generador final. Aunque estas señales no siempre se extienden por todos los módulos del árbol generador actual

debido a que algunos de los módulos del árbol están comprimidos en otros módulos, podemos considerar que cada módulo envía $O(H)$ señales con estos mensajes. Por tanto, el número total de mensajes es $N * H + N * H$, esto es $O(H * N)$ que en el peor de los casos es $O(N^2)$. Incluso evitando que las señales se extiendan al subárbol sur de la raíz cuando se expande el subárbol este del árbol, el coste total se mantiene en $O(N^2)$ ya que si no existe ningún módulo conectado a la raíz en dirección este en el árbol generador final esta manera de ahorrar en envío de señales tiene lugar en la reconfiguración. □

Como en las anteriores modificaciones versiones del algoritmo el ahorro de movimientos innecesarios por módulo y total se mantiene en $O(N)$ y $O(N^2)$ debido a que nuestra estimación considera el caso en que todas las ramas son de tamaño $O(N)$ y a que la velocidad de las señales y de los módulos comprimidos no ha cambiado respecto de los algoritmos anteriores.

5.1.4. Algoritmo multilíder

Este nuevo algoritmo es radicalmente diferente al resto de modificaciones del algoritmo original analizadas en este apartado, sin embargo, como veremos a continuación, podemos utilizar algunas ideas de los estudios teóricos anteriores para analizar los costes de este algoritmo.

Para poder determinar el número de señales adicionales que envía este algoritmo primero debemos realizar un análisis por separado del envío de las diferentes señales que introduce este algoritmo y que no existen en la versión original. Para empezar analizamos la señal numérica que acompaña a la señal de *Back_*.

Proposición 5.8 *El algoritmo multilíder aplicado a un árbol generador inicial de N módulos y a un árbol generador final de H hojas envía $O(N^2)$ mensajes de de texto Back_.*

Demostración: La señal de *Back_* es la señal que envían las hojas del árbol generador inicial una vez han recibido la señal de un candidato a raíz del árbol. Esta señal viaja de hijos a padres hasta llegar a la raíz del árbol a no ser que encuentre otra señal de un candidato a raíz mejor que el módulo que envió la señal que recibieron las hojas, en cuyo caso se envía esta nueva señal hasta las hojas del árbol y estas vuelven a emitir la señal de *Back_*. A esta señal de *Back_* se le añade una señal numérica que se envía junto a ella y que indica el número de módulos que cuelgan del módulo que la recibe en la dirección por la que ha recibido el mensaje. Por tanto esta señal se emite tantas veces como la señal de *Back_*. En un caso ideal con un solo candidato a raíz del árbol esta señal se envía $O(N)$ veces, una vez por cada módulo del árbol. Sin embargo, es posible existencia de más de un candidato. En el caso de un árbol con el mayor número posible de candidatos a raíz, como por ejemplo un árbol en forma de escalera que asciende de oeste a este con $N/2$

módulos como candidatos, el número de mensajes numéricos por módulo es $O(N)$, de manera que el total de mensajes enviados por el algoritmo es $O(N^2)$. □

La señal siguiente a tener en cuenta es la de operación de cambio de rama denegada. Esta señal depende exclusivamente del número de operaciones de cambio de rama que se envíen a hojas del árbol generador actual que ya hayan pedido permiso para comprimirse y recibido la confirmación de dicha acción.

Proposición 5.9 *El máximo número de mensajes de operación de cambio de rama denegada que pueden ser emitidos durante una ejecución del algoritmo multilíder es* $O(N)$.

Demostración: Como podemos deducir de las condiciones bajo las que se envía esta señal, predecir el número exacto de veces que esta se envía es imposible, ya que depende de cada configuración. Sin embargo, como sabemos que que esta señal solo puede ser emitida por una hoja en fase de compresión, podemos garantizar que, como máximo, esta señal se envía $O(N)$ veces, ya que a lo largo de la reconfiguración solo puede llegar a existir un total de N hojas en estado de compresión. □

Las señales siguientes a analizar son las dos que actualizan los registros que indican el número de hijos de cada módulo y el número de módulos que hace falta enviar en cada dirección. Estas señales se envían cuando se ha ejecutado una operación de cambio de rama para desconectar un módulo de una rama en estado de compresión y conectarla a una rama en estado de expansión. Una de las dos señales informa a la rama en estado de expansión de que se han añadido nuevos módulos a la rama y la otra informa a la rama en estado de compresión de que ha perdido módulos como resultado del cambio de rama.

Proposición 5.10 *En el algoritmo multilíder la emisión de una pareja de señales de adición y sustracción produce como máximo* $O(N)$ *mensajes.*

Demostración: Normalmente estas señales se expanden hasta la raíz informando a todos los módulos que encuentran a su paso. Sin embargo, existe una excepción: si dos o más señales de este tipo, ya sea de adición o de sustracción, se encuentran en un mismo módulo intersección este emite una única señal de adición o sustracción según el número de módulos sumados o sustraídos a sus ramas. Estas señales se comportan como la señal de *pausa* del algoritmo con señal de parada hasta la raíz, solo que en vez de emitir una sola señal en este caso se emiten dos al mismo tiempo, una por cada rama implicada en la operación. Para calcular el coste de emitir estas señales enumeramos las operaciones de cambio de rama de 1 a k y sumamos las

distancias de la raíz al módulo que ha realiza la operación de cambio de rama pasando por la rama en estado de expansión ($DistE_i$) y por la rama en estado de compresión ($DistC_i$).

$$Mensajes_{adición,substracción} = \sum_{i=1}^{k}(DistE_i + DistC_i)$$

Teniendo en cuenta que para que exista la operación del cambio de rama se necesitan dos ramas, y que la suma de la longitud de las dos ramas no puede ser superior a $N-1$, la emisión de estas dos señales juntas produce a lo sumo $O(N)$ mensajes. □

El total de mensajes en una reconfiguración debido a la emisión de estas señales es difícil de calcular. Para conseguirlo hemos analizado diversas situaciones teóricas que nos permiten poner a prueba el número potencial de veces que pueden ser emitidas estas señales.

Proposición 5.11 *El número de veces que se ejecuta satisfactoriamente una operación de cambio de rama durante una ejecución del algoritmo multilíder puede llegar a ser $N/3$ y genera $O(N^2)$ mensajes de adición y sustracción en total.*

Demostración: Consideremos una rama en estado de compresión que no está congestionada, esto es, cuyos módulos comprimidos aún tienen espacio para viajar por ella, y cuya hoja está constantemente perseguida por un módulo líder que le envía mensajes de cambio de rama. Una hoja en estado de compresión que tenga espacio para comprimirse siempre genera la señal de negación de cambio de rama, por lo que es imposible en este caso que las señales de adición/sustracción se emitan N veces.

Consideremos una rama en estado de compresión y congestionada, esto es, cuyos módulos comprimidos se bloquean unos a otros e impiden que se muevan. Supongamos que está formada por $N/2$ módulos que contienen $N/2$ módulos comprimidos. Consideremos una rama en expansión que envíe señales de cambio de rama a las hojas de la rama en compresión que se vayan generando. Llega un momento en que los módulos pueden moverse, generando una situación como la descrita en el párrafo anterior. Por consiguiente, las señales de adición y substracción no pueden ser emitidas $N/2$ veces.

Finalmente, al imaginar una rama en estado de compresión en línea recta y una rama en estado de expansión que intenta atravesarla varias veces como una costura vemos que es posible emitir las señales de adición y substracción $N/3$ veces. Por tanto podemos decir que el coste total del envío de estas dos señales en una sola ejecución del algoritmo es $N*(N/3)$ o $O(N^2)$ mensajes, $O(N)$ mensajes por módulo. No hemos podido demostrar que $N/3$ sea una

cota superior, pero no hemos podido llegar a plantear ningún otro ejemplo superior a $N/3$.

En todo caso, cada módulo solo puede cambiar de rama una vez, de modo que la cota $O(N^2)$ es inmediata. □

Por último solo nos queda analizar el mensaje que indica el número de módulos encontrados, un mensaje numérico que acompaña a las señales de adición y sustracción y que indica cuántos módulos se han añadido o perdido en la rama. Estas señales se envían a la vez que las señales de adición y sustracción por lo que su coste es el mismo. Cabe destacar que el coste de las señales de adición y substracción sumado al coste de enviar este tipo de mensajes sigue siendo $O(N^2)$.

Proposición 5.12 *El algoritmo multilíder aplicado a un árbol generador inicial de N módulos y a un árbol generador final de H hojas envía $O(N^2)$ mensajes más que el algoritmo original.*

Demostración: Utilizando los cálculos de las señales previamente descritas vemos que en total el número de mensajes adicionales enviados por módulo es de $O(N)$ mensajes y el total de mensajes adicionales enviados por el algoritmo, es $O(N^2)$.

$$Coste_{módulo} = N/2 + 1 + 2 * N/3 = O(N)$$

$$Coste_{total} = N * N/2 + N + N * 2 * N/3 = O(N^2)$$

□

Una vez calculado el número de mensajes adicionales que este algoritmo produce, nos queda calcular el número de movimientos innecesarios que evita. Este algoritmo, como los demás, se centra en evitar los movimientos innecesarios de los módulos que viajan dentro de una rama en estado de expansión y en ese sentido se puede decir que este algoritmo es prácticamente perfecto. En ocasiones el algoritmo multilíder genera movimientos innecesarios al efectuar una operación de cambio de rama y, por tanto, para calcular el número de movimiento innecesarios que evita conviene estudiar primero el número total de movimientos innecesarios que efectúa el algoritmo original, y a ese número debemos restarle el número de movimientos innecesarios que genera nuestro algoritmo.

Proposición 5.13 *El número de movimientos innecesarios en ramas en fase de expansión durante una ejecución del algoritmo original puede llegar a ser $O(N^2)$.*

Demostración: Pongamos como ejemplo un árbol generador final cuyos módulos cuelgan todos del hijo sur de la raíz menos uno que debe convertirse en

el hijo de la raíz en dirección este. Teniendo en cuenta el algoritmo original y su prioridad de expansión, este módulo será el último en ser expandido al reconfigurar el árbol generador inicial. Además, en este caso, todos los módulos del árbol generador inicial cuelgan del subárbol este de la raíz, de forma que todos los módulos, al ejecutar el algoritmo, pasan por la raíz del árbol. En el peor de los casos el árbol generador final tiene una rama en el subárbol sur de la raíz de tal forma que cuando su hoja llega a su forma final el módulo destinado a convertirse en hijo en dirección este de la raíz se encuentra contenido en el módulo padre de la hoja de esta rama. Este es el peor de los casos porque ahora debe volver a la raíz para acabar ocupando su lugar después de recorrer prácticamente los N módulos del árbol dos veces. En concreto este módulo, teniendo en cuenta que acabará siendo la única hoja del sub-árbol este de la raíz, ha realizado $2*(N-R-raíz)$ movimientos innecesarios, que son $O(N^2)$ movimientos. Haciendo una aproximación, asumiendo que todos los módulos que realizan movimientos innecesarios pueden llegar a recorrer casi N módulos dos veces, y sabiendo que solo $N/2$ módulos pueden realizar movimientos innecesarios, el número total de movimientos innecesarios en una ejecución del algoritmo original es de $O(N^2)$ movimientos.

$$Movimientos_{innecesarios} = 2 * (N - R - raíz) * N/2$$

□

Para calcular el número de movimientos innecesarios que pueden generar las operaciones de cambio de rama y las señales de adición y substracción que esta genera hemos analizado en más profundidad el efecto de dichas señales.

Proposición 5.14 *El número de movimientos innecesarios producidos en ramas en fase de expansión durante una ejecución del algoritmo multilíder es de $O(N^2)$ movimientos.*

Demostración: Supongamos un árbol generador actual y final de misma forma que los de la Figura (5.1) pero con diferente número de módulos en sus ramas. En este ejemplo, en el momento en que la rama sur del árbol encuentra la rama en estado de compresión y realiza la operación de cambio de rama, contamos con $N/6$ módulos en estado de expansión ya expandidos, es decir, que no se encuentran comprimidos, con $N/2$ módulos comprimidos viajando, a través de los módulos en fase de expansión, hacia el líder de la rama sur y con $N/3$ que se han añadido a la rama sur con la operación de cambio de rama. En el caso que estamos presentando, los $N/3$ módulos añadidos a la rama sur son suficientes para completar la expansión de la rama y por tanto, a medida que reciben la señal que actualiza el número de módulos de la rama, los módulos comprimidos que viajan hacia el líder de la rama sur cambian su dirección para dirigirse al líder de la rama este.

Teniendo en cuanta que hay 2 módulos vacíos entre cada módulos comprimidos y considerando el número de módulos comprimidos como C y el número de módulos en fase de expansión ya expandidos como E, podemos calcular el número de movimientos innecesarios que han realizado los módulos comprimidos de la siguiente manera:

$$Movimientos_{innecesarios} = 1 + (1+3) + (1+3*2) + ... = \sum_{i=1}^{N/6}(1+i3)$$

$$= \frac{N}{6} + 3\sum_{i=1}^{N/6} i = \frac{N}{6} + 3\frac{(1+\frac{N}{6})\frac{N}{6}}{2}$$

$$= O(N^2)$$

□

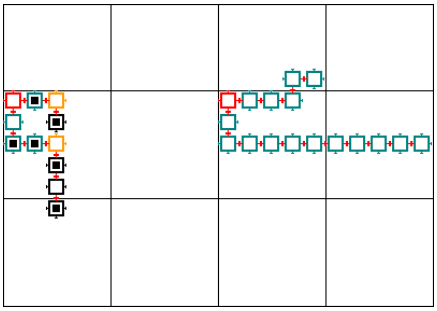


Figura 5.1: Árbol generador actual a la izquierda y árbol generador final a la derecha. El árbol generador actual, más concretamente su *líder* sur, acaba de realizar una operación de cambio de rama. A consecuencia de esta operación, un módulo que viaja hacia el *líder* de la rama sur debe volver al *líder* de la rama este.

Tanto el número de movimientos innecesarios eliminados como el número de movimientos innecesarios añadidos son cuadráticos. Nos ha sido imposible comparar dichos números en un contexto teórico general, ya que ambos dependen fuertemente de las configuraciones que se analicen. De hecho, el caso peor para la Proposición 5.13 no es el mismo que para la Proposición 5.14.

5.2. Análisis experimental de las modificaciones

5.2.1. Introducción a los resultados

A continuación presentamos un estudio experimental de los algoritmos presentados en este proyecto. Esta serie de experimentos no solo nos permi-

ten probar los algoritmos, además podemos analizar las instrucciones ejecutadas y los mensajes enviados para comprobar la certeza de nuestro análisis teórico y obtener datos reales sobre la media de mensajes y movimientos por ejecución.

Para minimizar el tamaño de las diferentes gráficas de esta sección hemos abreviado el nombre de los algoritmos con estas siglas:

- Algoritmo original ⇒ AO
- Algoritmo con señal de parada hasta la intersección ⇒ PI
- Algoritmo con señal de parada hasta la raíz ⇒ PR
- Algoritmo con señal de parada para toda la configuración ⇒ PT
- Algoritmo multilíder ⇒ ML

5.2.2. Herramientas utilizadas

Para realizar al análisis práctico se han utilizado dos herramientas distintas: el simulador de robots cristalinos y un parser de acciones exportadas del simulador.

El simulador es el mismo utilizado por otros proyectos de final de carrera como el de Joan Soler [3], autor del algoritmo original, puede encontrarse más información respecto a esta herramienta en su página web [6].

El parser de acciones es un programa creado específicamente para este proyecto. Más información sobre el parser se presenta en el anexo 6.

5.2.3. Juegos de prueba

Los juegos de prueba creados para analizar el comportamiento de los algoritmos de este proyecto están clasificados en las siguientes categorías:

Minihole-Spiralhole-Square: Ejemplos más complejos que los presentados en la categoría *Densidad* que buscan ver cómo aumentan la emisión de señales y el número de movimientos en función del número de módulos que componen la figura. Estos ejemplos están formados por figuras de forma Minihole que pasan a forma Spiralhole y Square, figuras Spiralhole que pasan a forma Minihole y Square y, por último, figuras Square que pasan a forma Minihole y Spiralhole. Además, todos estos casos se repiten múltiples veces cambiando el número de módulos que los componen: 10, 20, 50, 100, 200, 500 y 1000.

Peines & Rectángulos: Ejemplos diseñados para estudiar el impacto de la orientación de las ramas y el orden de compresión y expansión de los árboles generadores inicial y final en la configuración en figuras poco densas de tipo histograma (Peines) o muy densas (Rectángulos), que se ilustran en la Figura .

Estos juegos de prueba no solo buscan analizar el comportamiento de los algoritmos según el número de módulos, sino comprobar si la composición, orientación o densidad de una figura influye de alguna forma en la reconfiguración. Este análisis nos puede ser útil si en un futuro se deja a los robots, o en nuestro caso al simulador, la tarea de decidir la mejor manera de abordar una reconfiguración. Por ejemplo, si se envía a los robots una forma final, estos podrían cambiar la prioridad de compresión y/o expansión de las diferentes direcciones de un módulo, simulando un cambio de orientación en la figura aunque esta no cambie, si los robots consideran que una reconfiguración con las prioridades originales no es óptima. Hasta entonces, esta experimentación también nos sirve para crear cambios de forma que faciliten la reconfiguración.

Todos estos juegos de prueba así como el análisis de sus resultados mediante el parser de acciones pueden encontrarse en la página web de este proyecto.

5.2.4. Movimientos según el número de módulos

Antes de empezar con esta fase de la experimentación debemos puntualizar que entendemos como movimiento toda compresión, expansión o paso de módulo comprimido. No entendemos como movimientos los cambios de conexión de un módulo.

Para empezar veamos, con los resultados obtenidos de los casos de prueba Minihole-Spiralhole-Square, cómo evolucionan el número de movimientos y de mensajes al incrementar el número de módulos de nuestras configuraciones.

Como puede verse en la Figura 5.2, todos nuestros algoritmos mejoran el original. Aunque cada mejora del algoritmo realiza menos movimientos totales que la anterior, la mejora de una modificación respecto a la anterior es minúscula salvo para el algoritmo multilíder. Si comparamos el algoritmo con señal de parada hasta intersección, con señal de parada hasta raíz y con señal de parada para toda la configuración con el algoritmo original para configuraciones de menos de 20 módulos, la diferencia en el número de movimientos difícilmente llega a los 10 movimientos. Incluso hay ocasiones, como durante la ejecución de los juegos de prueba de 20 módulos, en que el algoritmo con señal de parada para toda la configuración obtiene peores resultados que los algoritmos con señal de parada hasta intersección y con señal de parada hasta raíz. Solo el algoritmo multilíder consigue diferenciarse substancialmente del resto, ya que es capaz de llevar a cabo las mismas reconfiguraciones con poco más de la mitad de movimientos que el resto de los algoritmos.

La escasa diferencia de movimientos entre el algoritmo original y los algoritmos con señal de para se debe a que, al fin y al cabo, su funcionamiento es extremadamente parecido y a que la señal de parada solo evita un pequeño

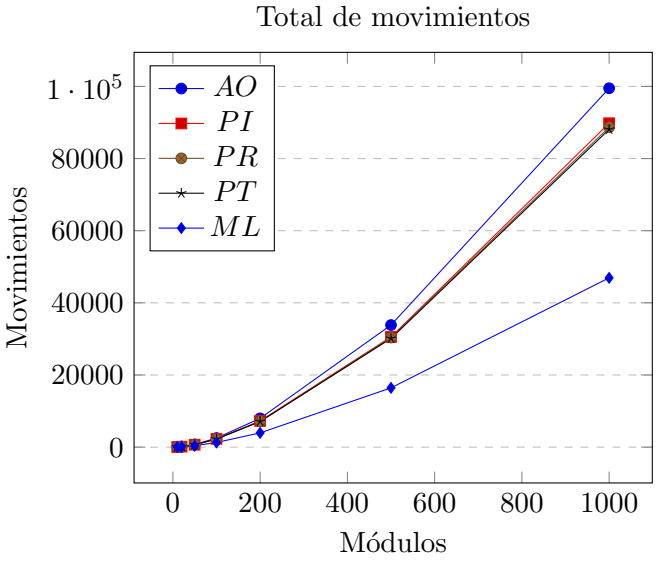


Figura 5.2: Gráfica que muestra, para los algoritmos presentados en este proyecto y el algoritmo original, la media del número total de movimientos realizados en los juegos de prueba Minihole-Spiralhole-Square.

número de movimientos innecesarios. Por otro lado, el algoritmo multilíder solo realiza movimientos innecesarios al realizar una operación de cambio de rama, y, aún así, el número de módulos afectados por esta operación es mucho menor que el número de módulos que realizan movimientos innecesarios en los otros algoritmos.

La misma diferencia en el número de movimientos innecesarios se da tanto en la fase de expansión, Figura 5.4, como en la fase de compresión, Figura 5.3.

Nuestro cálculo teórico sobre la complejidad de los algoritmos estudiados en este proyecto, marca una cota superior de $O(N^2)$ para todos los algoritmos. Aún así, como podemos apreciar en la Figura 5.2, durante la ejecución de los juegos de prueba ninguno de los algoritmos superó la complejidad indicada en el Apartado 5.1 y, concretamente en el caso del algoritmo multilíder, los resultados reales son mucho más bajos que la cota calculada.

5.2.5. Mensajes según el número de módulos

Junto con la comparación de movimientos, comparar el número de mensajes que emite cada modificación del algoritmo es otro de los elementos más interesantes de esta experimentación. Entendemos como mensaje, no solo los mensajes numéricos y de texto emitidos por los módulos, sino también todas las lecturas de estados o registros de los módulos vecinos ya que, en la vida real, estos datos se comunican mediante mensajes.

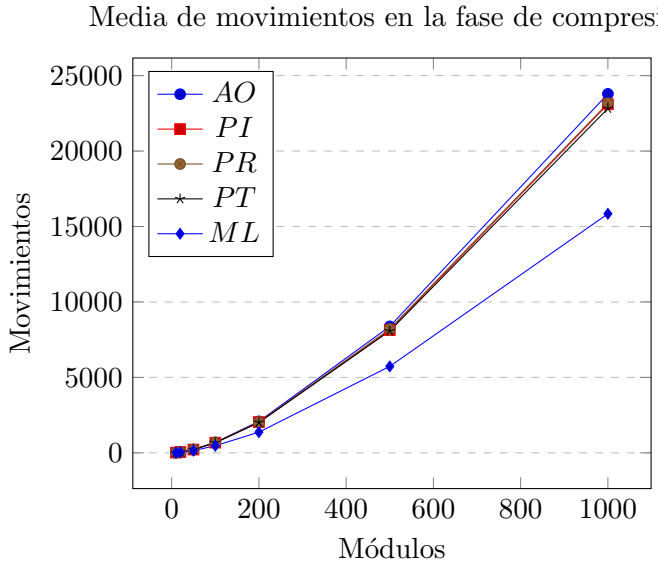


Figura 5.3: Gráfica que muestra, para los algoritmos presentados en este proyecto y el algoritmo original, la media del número de movimientos realizados en los juegos de prueba Minihole-Spiralhole-Square durante la fase de compresión.

Al incrementar el número de módulos no solo aumenta el número de movimientos, el número de mensajes también se ve afectado por este incremento. A continuación presentamos la media de los resultados obtenidos por cada algoritmo según el número de módulos de los casos de prueba Minihole-Spiralhole-Square.

Como puede verse en la Figura 5.5, todos los algoritmos con señal de parada emiten, durante su fase inicial, el mismo número de mensajes que el algoritmo original. Este hecho era de esperar ya que la fase de búsqueda realiza las mismas acciones en el algoritmo original que en los algoritmos con señal de parada. El algoritmo multilíder, sin embargo, emite N mensajes más que los demás algoritmos, siendo N el número de módulos de la configuración. Estos mensajes adicionales son los los mensajes numéricos que acompañan a la señal de *Back_* y que informan a los módulos del número de descendientes en cada una de sus direcciones. Aún así, como veremos, estos mensajes adicionales no suponen una gran carga en el número total de mensajes emitidos por el algoritmo multilíder.

Es en las Figuras 5.6 y 5.7 donde vemos el efecto especifico sobre los mensajes de los algoritmos con señales de parada y del algoritmo multilíder. Tanto el algoritmo con señal de parada como el algoritmo multilíder emiten menos mensajes que el algoritmo original. Aunque originalmente esperábamos reducir el número de movimientos a costa de un aumento en el número

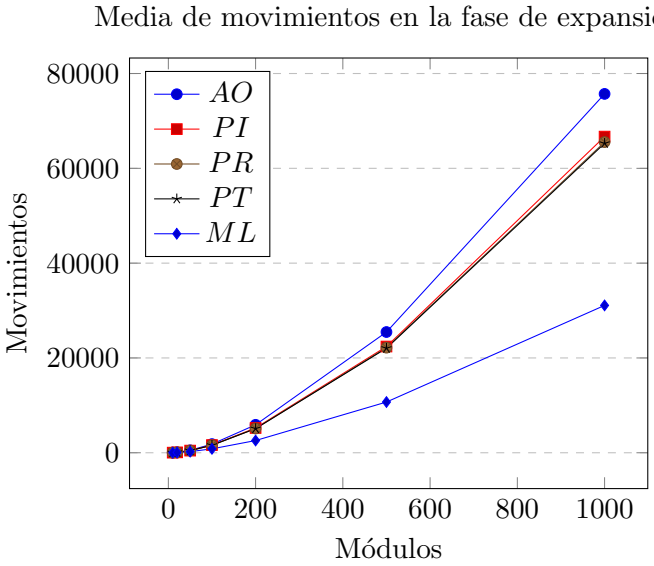


Figura 5.4: Gráfica que muestra, para los algoritmos presentados en este proyecto y el algoritmo original, la media del número de movimientos realizados en los juegos de prueba Minihole-Spiralhole-Square durante la fase de expansión.

de señales emitidas, podemos ver que no ha sido el caso.

La mayor fuente de mensajes emitidos, tanto del algoritmo original como de los algoritmos con señal de parada, es el envío continuo, a cada iteración, de mensajes para iniciar una operación de paso de módulo comprimido. Siempre que un módulo contiene un módulo comprimido intenta, a cada iteración, iniciar una operación de paso de módulo comprimido y no cesa de intentarlo hasta conseguir una respuesta de su padre. Los algoritmos con señal de parada, al pausar los módulos, evitando así que los módulos pausados puedan realizar acción alguna, consiguen evitar también el envío de los mensajes de inicio de operación de paso de módulo comprimido, y mientas más módulos pause cada algoritmo, menos mensajes envía. Por eso los algoritmos con señal de parada que emiten menos mensajes son, en orden de mayor a menor número de mensajes emitidos, el algoritmo con señal de parada hasta intersección, el algoritmo con señal de parada hasta la raíz y el algoritmo de señal de parada para toda la configuración.

El algoritmo multilíder, por otro lado, no genera apenas colapsos de módulos comprimidos durante su fase de expansión, envía los mensajes de inicio de operación de paso de módulo comprimido una vez cada tres iteraciones en lugar de una vez por iteración, y, al haber menos colapsos en las ramas en fase de expansión, reduce también el tiempo de colapsos de módulos comprimidos en la fase de compresión. Todo ello reduce en gran

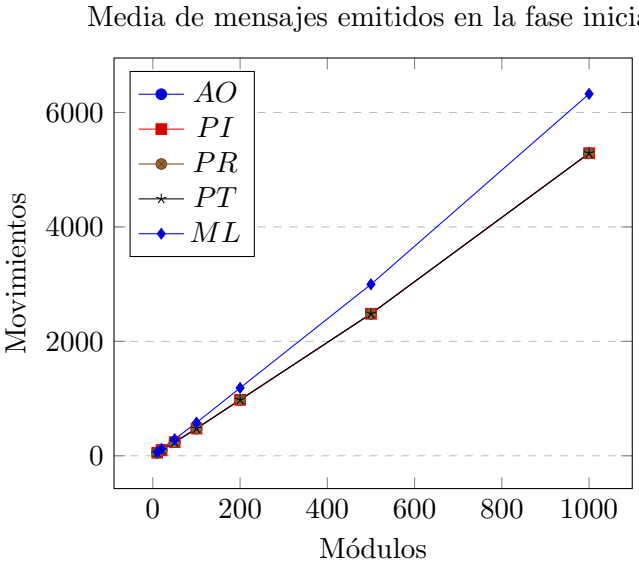


Figura 5.5: Gráfica que muestra, para los algoritmos presentados en este proyecto y el algoritmo original, la media del número de mensajes emitidos en los juegos de prueba Minihole-Spiralhole-Square durante la fase de búsqueda de la raíz.

medida el número de mensajes emitidos por este algoritmo.

Otra de las razones por la que los algoritmos presentados en este proyecto emiten menos mensajes que el algoritmo original es que estos mensajes se emiten para iniciar el movimiento de un módulo. Por tanto, cuantos menos movimientos realiza el algoritmo, menos mensajes totales necesita enviar.

En definitiva, de entre todos los algoritmos estudiados y propuestos en este proyecto, el algoritmo multilíder es el que menos señales emite con diferencia.

Si comparamos los resultas obtenidos en la experimentación con los previstos en el cálculo de la complejidad de los algoritmos (Apartado 5.1) podemos ver que no nos equivocamos en el cálculo de su cota superior. Es más, los resultados reales son menores que la cota calculada dado que el número de mensajes esta directamente relacionado con el número de movimientos, por lo que menos movimientos implica menos mensajes emitidos.

Podemos ver en la Figura 5.5 que, tal como se indica en la Proposición 5.8 y en su demostración, el número de mensajes emitidos por el algoritmo multilíder durante su fase inicial supera siempre en poco más de N, siendo N el número de módulos de la configuración, al número de mensajes emitidos en la misma fase por el resto de los algoritmos y, en todo caso, es lineal.

Otro dato que podemos comprobar es, tal como muestra la Figura 5.9, que el número de operaciones de cambio de rama que se realizan, y por

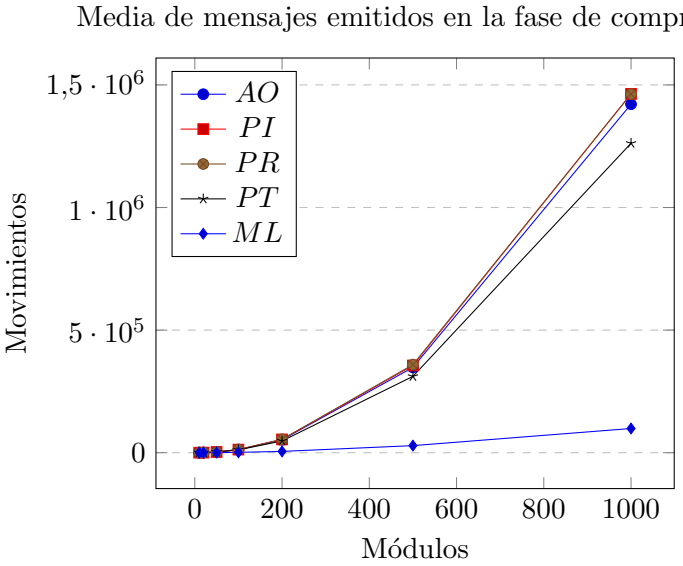


Figura 5.6: Gráfica que muestra, para los algoritmos presentados en este proyecto y el algoritmo original, la media del número de mensajes emitidos en los juegos de prueba Minihole-Spiralhole-Square durante la fase de compresión.

tanto el número de mensajes de cambio de rama que se emiten, nunca supera $1/3{*}N$ (Proposición 5.11). Curiosamente, la Figura 5.9 también nos muestra que mientras más módulos hay en una configuración más numerosas son las operaciones de cambio de rama.

5.2.6. Orden de compresión en los algoritmos

Uno de los elementos que más llaman la atención durante la ejecución del algoritmo multilíder es que, comparado con el algoritmo original, su orden de compresión resulta aparentemente caótico y aleatorio para el usuario. Este desorden en la compresión no es el comportamiento que esperábamos en el algoritmo multilíder, por lo que, durante la la experimentación con el algoritmo multilíder, hemos buscado la causa de la falta de orden en la compresión de ramas del árbol generador actual.

En el algoritmo original existe un orden de expansión definido: recibir un módulo del oeste tiene más prioridad que recibir un módulo del norte, que a su vez tiene más prioridad que recibir un módulo del este que, finalmente, tiene más prioridad que recibir un módulo del sur. Esta prioridad consigue mantenerse gracias a una serie de mensajes, que se repiten a cada iteración, en que un módulo que contiene un módulo comprimido le pide a un vecino, al que esta conectado, que acepte el paso de dicho módulo. Este mensaje se envía siempre, incluso durante la iteración en la que sabemos con seguridad

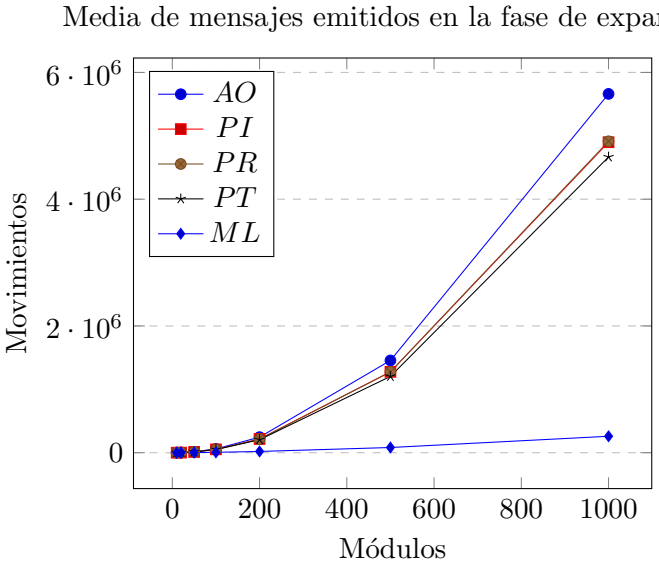


Figura 5.7: Gráfica que muestra, para los algoritmos presentados en este proyecto y el algoritmo original, la media del número de mensajes emitidos en los juegos de prueba Minihole-Spiralhole-Square durante la fase de expansión.

que el módulo comprimido va a enviarse pues ya se ha confirmado la operación. Además la velocidad a la que viajan los módulos comprimidos, un módulo cada tres iteraciones, permite que un módulo que ha empezado a recibir módulos comprimidos envíe, casi a cada iteración, una señal de paso de módulo comprimido. Toda esta cantidad de mensajes, aunque innecesaria, asegura que si un módulo intersección con dos ramas en fase de compresión, por ejemplo su rama este y su rama sur, empieza a recibir módulos de su rama este, entonces la intersección no puede recibir módulos de su rama sur hasta que la rama este no ha sido absorbida en su totalidad. Esto ocurre debido a que las señales continuadas que envía la rama este siempre dan prioridad a dicha rama.

Por otra parte, en el algoritmo multilíder, aunque comparte la misma prioridad de compresión que el algoritmo original, no ocurre lo mismo. A diferencia del algoritmo original, el algoritmo multilíder no envía estas señales de paso de módulo comprimido a cada iteración, sino que las emite cuando el módulo que ha enviado la señal sabe que es imposible que se reciba una confirmación, una vez pasada la tercera iteración desde que se envió la petición. Otra diferencia es que los módulos comprimidos en ramas en fase de compresión del algoritmo multilíder viajan a un módulo cada cuatro iteraciones en lugar de tres, como en el algoritmo original. Todo esto crea una distancia entre módulos comprimidos lo suficientemente grande como

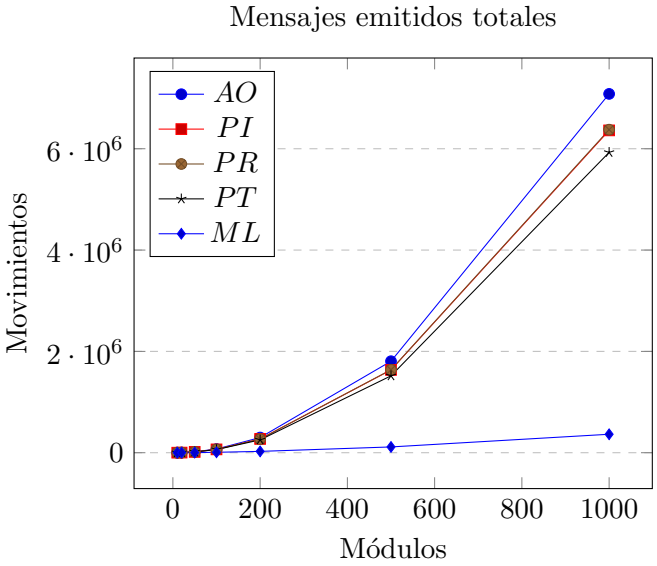


Figura 5.8: Gráfica que muestra, para los algoritmos presentados en este proyecto y el algoritmo original, la media del número total de mensajes emitidos en los juegos de prueba Minihole-Spiralhole-Square durante las fases inicial, de expansión y de compresión.

para que algunos módulos comprimidos de otras ramas con menor prioridad puedan ser recibidos por el módulo intersección. Incluso cuando los módulos comprimidos entran en ramas en fase de expansión, en donde viajan a la misma velocidad que en el algoritmo original, se sigue manteniendo esta distancia entre módulos.

En ocasiones, en el caso del algoritmo multilíder, podemos observar ramas con mayor prioridad que dejan de comprimirse hasta que otra rama, con menor prioridad, no se ha comprimido por completo. Esto es debido a que un módulo, de una rama con menos prioridad, ha aprovechado la mencionada distancia entre módulos comprimidos para entrar en el módulo intersección justo en el momento en que la rama con mayor prioridad envía su mensaje de paso de módulo comprimido. Como en ese momento el módulo intersección ya contiene un módulo comprimido, no puede confirmar la operación. De esta forma, se crea un ciclo en la rama con mayor prioridad en que, siempre que esta envía el mensaje de paso de módulo comprimido al módulo intersección, la intersección siempre está ocupada por otro módulo.

Como resultado de estas intrusiones en la compresión, las ramas en fase de compresión se comprimen, en apariencia, de forma caótica. Si bien este detalle no afecta negativamente a la reconfiguración, la velocidad a la que los módulos en fase de compresión se incorporan a una rama en fase de expansión se mantiene siempre estable, sí que puede hacer que la fase de

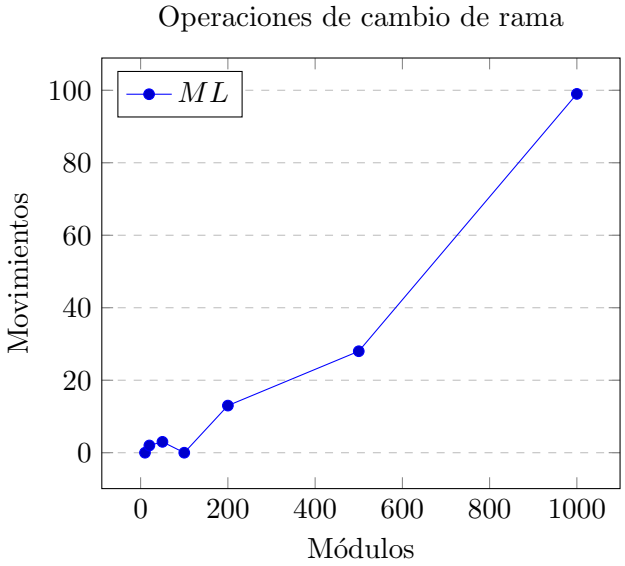


Figura 5.9: Gráfica que muestra, para el algoritmo multilíder, el número total de operaciones de cambio de rama realizadas durante los juegos de prueba Minihole-Spiralhole-Square.

compresión resulte más confusa para el usuario del simulador.

Un ejemplo claro de la diferencia en el orden de compresión es el del juego de prueba de *Rectángulos* en el que se mide el impacto de la orientación de una figura densa. Tal como se puede apreciar en la Figura 5.10, el orden de compresión del algoritmo multilíder es más impredecible que el del algoritmo original. En la imagen (b) de la figura, se observa un orden de compresión en el que ninguna ninguna rama puede comprimirse hasta que otra de mayor prioridad haya acabado de hacerlo. En el caso del algoritmo multilíder, imagen (c), la rama con mayor coordenada X representa la rama con mayor prioridad. Aquí se observa lo poco predecible que puede llegar a ser la compresión de sus ramas en fase de compresión. En este caso tanto las ramas con menor coordenada X, menor prioridad, como la rama con mayor coordenada X, mayor prioridad, consiguen comprimir módulos hacia la raíz de forma paralela.

5.2.7. Impacto de la orientación en figuras densas

Para estudiar el impacto de la orientación en figuras densas usamos los resultados de experimentar con los juegos de prueba *agents_rectangles_5x20x5* y *agents_rectangles_20x5x20* de la categoría *Rectangulos* ejecutando el algoritmo multilíder. Estos dos juegos de prueba son como los mostrados en la Figura 5.10 del apartado 5.2.6, un rectángulo en horizontal que pasa a formal un rectángulo en vertical y viceversa.

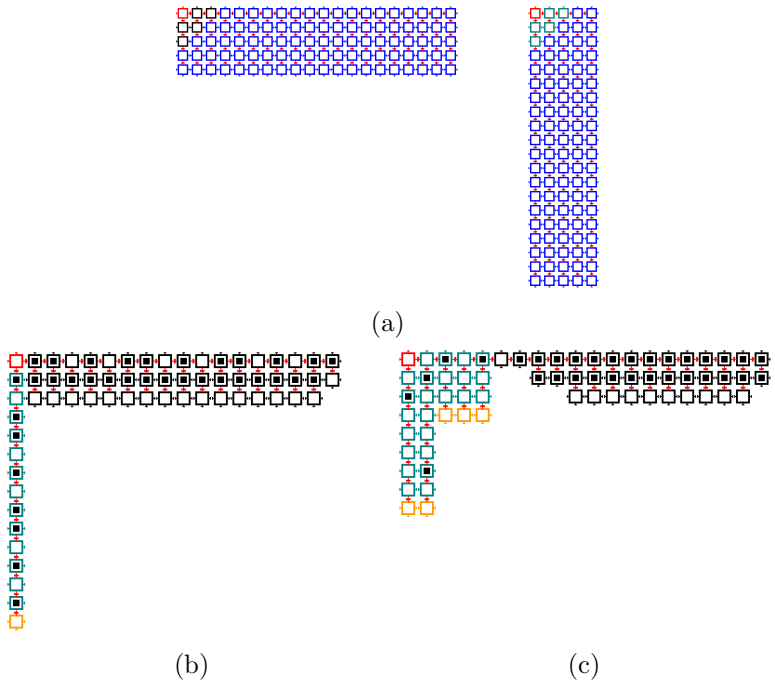


Figura 5.10: Las imágenes (b) y (c) muestran un árbol generador actual de forma rectangular que pasa de horizontal a vertical, tal como muestra la imagen (a). La imagen de la izquierda, (b), muestra una reconfiguración mediante el algoritmo original. La imagen de la derecha, (c), muestra una reconfiguración mediante el algoritmo multilíder.

Como podemos ver en la Figura 5.11, el paso de vertical a horizontal realiza menos movimientos, y por tanto emite menos mensajes, que el paso de horizontal a vertical. Esto se debe a dos motivos. El primer motivo es que el algoritmo multilíder, y todos los demás algoritmos estudiados en este proyecto, tiende a formar largas ramas hacia el sur. El segundo es la altura, la distancia de la raíz al módulo con menor coordenada Y, de la figura.

El motivo más obvio es la altura. En el caso de vertical a horizontal se realizan menos movimientos porque a la hora de expandir módulos en dirección sur encontramos suficientes módulos no comprimidos y, por tanto, no necesitamos realizar ningún movimiento para realizar la expansión. En el caso de horizontal a vertical la altura es menor, por lo que no existen suficientes módulos al sur de la raíz para expandir las primeras ramas, realizando así más movimientos que en el caso anterior.

La razón por la que la dirección de las ramas ayuda a reducir movimientos es menos evidente. Estas ramas se forman durante la fase inicial al resolver los ciclos del grafo de adyacencia. Gracias a la prioridad de dirección a la hora de resolver este problema, siempre que un módulo puede recibir un

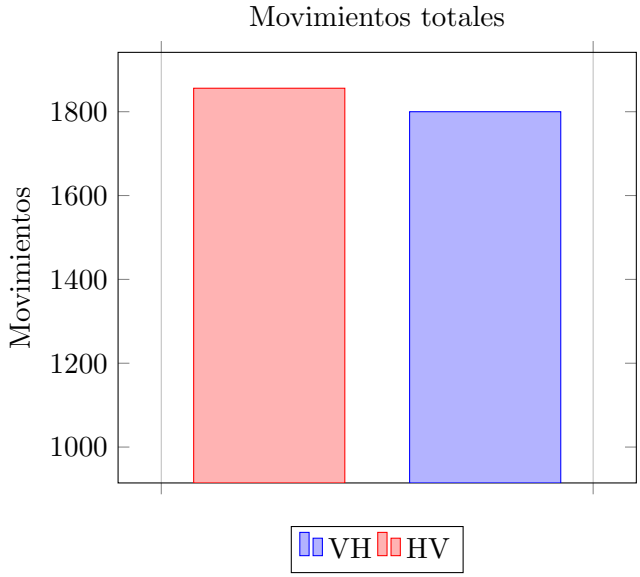


Figura 5.11: Gráfico de barras que representa el número de movimientos realizados por el algoritmo multilíder al ejecutar los juegos de prueba *agents_rectangles_5x20x5* (VH, de vertical a horizontal) y *agents_rectangles_20x5x20* (HV, de horizontal a vertical).

mensaje de un mismo candidato a raíz por más de una dirección, entre las que se encuentre la dirección norte, el módulo permanece unido a su vecino del norte una vez acabada la fase inicial. De esta forma, una vez se inicia la fase de expansión, en el caso de vertical a horizontal, ya existen suficientes módulos hacia el sur como para expandir las primeras ramas del árbol generador actual sin tener que realizar movimiento alguno. En el caso de horizontal a vertical, al tener menos altura, se han comprimido demasiados módulos y, a la hora de realizar la expansión de las primeras ramas, se realizan movimientos que el caso anterior evita. Si el algoritmo no generara las ramas en linea recta y de norte a sur, no podría realizarse de forma tan rápida esta expansión de las ramas más cercanas a la raíz, el árbol generador actual no tendría módulos al sur de la raíz como para expandir estas primeras ramas, aumentando así el número de movimientos del algoritmo. Puede existir el caso en que al pasar de vertical a horizontal se necesite realizar algún movimiento para expandir las ramas más cercanas a la raíz, sin embargo nunca serán tantos como cuando el movimiento de una figura se realiza de horizontal a vertical.

5.2.8. Impacto de la orientación en figuras poco densas

Para experimentar con la orientación de las figuras poco densas se han usado los casos de prueba de la categoría *Peines* ejecutando el algoritmo

multilíder. Cada juego de prueba representa un histograma, orientado en una de las cuatro direcciones de los puntos cardinales, que debe cambiar su orientación. Cada barra del histograma, así como su base, es de un solo módulo de ancho. Además, estos juegos de prueba también contemplan el paso de una orientación a la misma.

Para analizar los resultados de esta experimentación, hemos agrupado los resultados según su orientación de origen. Un mismo grupo de resultados contiene tres cambios de orientación y una reconfiguración de una orientación a si misma.

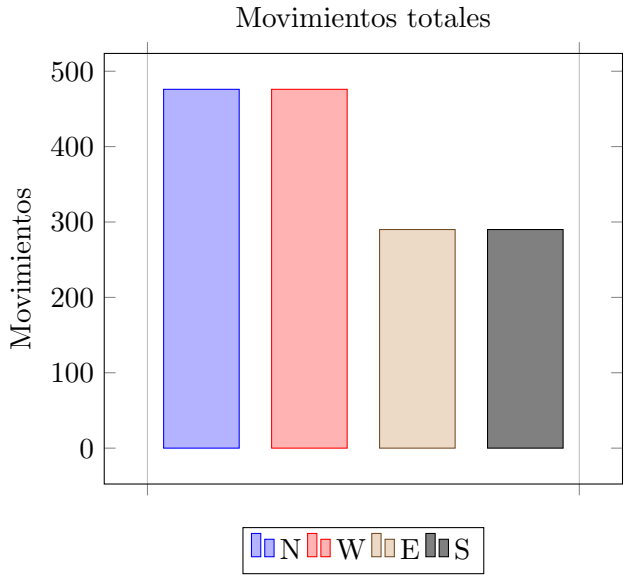


Figura 5.12: Gráfico de barras que representa el número de movimientos realizados por el algoritmo multilíder al ejecutar los juegos de prueba de la categoría *Peines* con misma forma inicial y final. Cada sigla de la leyenda representa la orientación de su forma inicial.

Empezamos observando los resultados de los movimientos realizados al cambiar de una forma a si misma. Como es lógico, en estos casos, se realizan los mismos movimientos durante la fase de compresión que durante la fase de expansión (todos los módulos comprimidos deben volver a su posición inicial). Sin embargo, podemos apreciar un dato algo extraño: los juegos de prueba con dirección de origen norte y oeste no realizan los mismos movimientos que los de dirección de origen este y sur (Figura 5.12). Antes de explicar el porqué de este suceso primero veamos el resultado de los otros juegos de prueba.

El resto de juegos de prueba muestran resultados igual de extraños (Figura 5.13). Si nos fijamos bien en los juegos de prueba inversos vemos que nos encontramos en la misma situación que la descrita en el párrafo anterior.

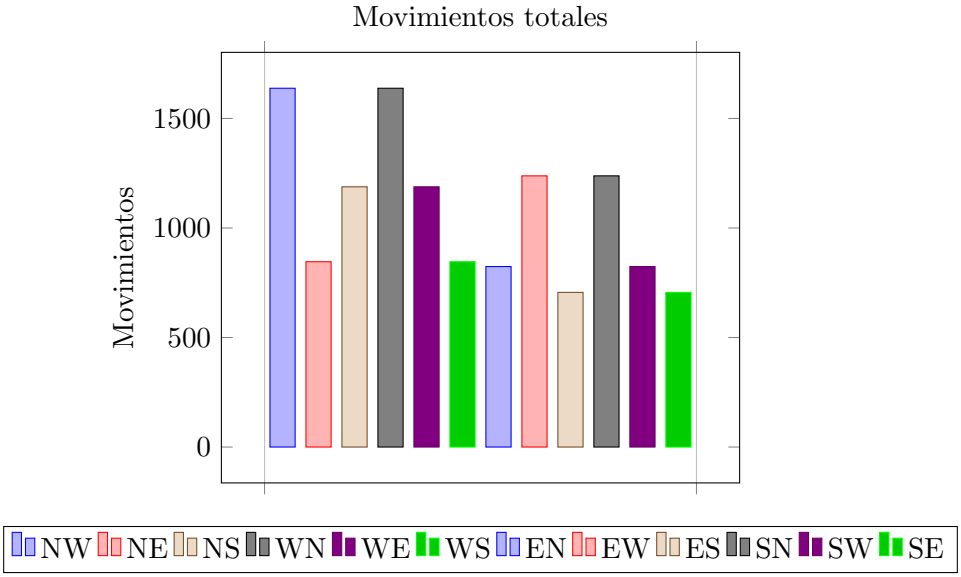


Figura 5.13: Gráfico de barras que representa el número de movimientos realizados por el algoritmo multilíder al ejecutar los juegos de prueba de la categoría *Peines* con forma inicial y final diferentes. La primera sigla de la leyenda representa la orientación de su forma inicial y la segunda la orientación de su forma final.

Entendemos como juegos de prueba inversos dos juegos de prueba diferentes cuya forma inicial en uno es la forma final del otro y donde la forma final del primero es igual a la forma inicial del segundo. Los resultados de los juegos de prueba de norte a oeste y de oeste a norte son idénticos, como lo son los de los juegos de prueba de este a sur y de sur a este. Sin embargo, los resultados de los juegos de prueba del resto de casos inversos no coinciden. Es más, los resultados de algunos juegos de prueba que, a simple vista, no tienen nada que ver coinciden, como los juegos de prueba de norte a este y de oeste a sur.

Tras analizar los datos obtenidos de las gráficas de las Figuras 5.12 y 5.13 podemos llegar a una conclusión. La orientación de una figura poco densa solo importa porque puede llegar a determinar si la raíz del árbol generador, tanto inicial como final, tiene un hijo o dos. Todos los juegos de prueba invertidos que coinciden, solo coinciden porque ambos pasan de un árbol generador inicial a un árbol generador final con el mismo número de hijos, no por ser uno el caso inverso del otro. Es más, los juegos de prueba no invertidos cuyos resultados coinciden, son los que pasan de un árbol generador inicial a un árbol generador final cuyas raíces tienen un número diferente de hijos y cuya carga de módulos se se distribuyen de la misma manera durante la reconfiguración.

En definitiva, la orientación en una figura poco densa solo influye en la reconfiguración si esta determina el número de hijos de la raíz.

5.2.9. Reconfiguración de figuras sin ciclos

Todos los juegos de prueba presentados hasta ahora están formados, antes de iniciar la ejecución de un algoritmo, por módulos conectados a todos sus vecinos. Sin embargo, puede interesarnos abordar casos en que el grafo de adyacencias inicial esta formado por módulos que no están conectados a todos sus vecinos, tal como en los ejemplos de la Figura 5.14. Por supues-

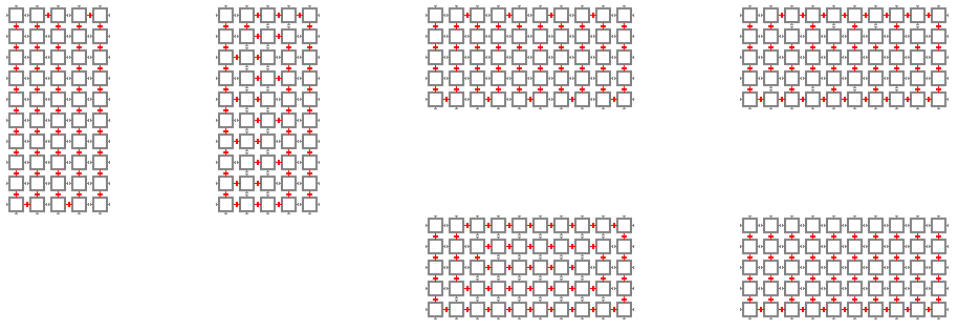


Figura 5.14: Figura que muestra un juego de pruebas con tres rectángulos sin ciclos que deben tomar la misma forma de la que parten.

to todos de los algoritmos, tanto el original como los presentados en este proyecto, son capaces de llevar a cabo la reconfiguración de una figura sin ciclos, ya que, durante la fase de inicial de la reconfiguración, el grafo de adyacencias se rompe en forma de árbol generador inicial, que es un grafo de adyacencia sin ciclos.

Capítulo 6

Analizador sintáctico de acciones

El analizador sintáctico de acciones, o parser de acciones, es una herramienta diseñada para analizar las diferentes acciones realizadas durante la ejecución de un conjunto de reglas por el simulador de robots cristalinos.

6.1. ¿Para qué necesitamos un parser de acciones?

Si queremos analizar en profundidad el impacto de un conjunto de reglas para el simulador de robots cristalino sobre un grupo de robots necesitamos saber exactamente el número de mensajes o movimientos realizados durante la ejecución del conjunto de reglas.

Aunque el simulador de robots cristalino dispone de una serie de herramientas que registran y permiten visualizar qué reglas se han aplicado en cada iteración nos es imposible saber el número de movimientos o de mensajes que han producido o emitido dichas reglas a no ser que el usuario busque y anote, iteración tras iteración, los cambios en el sistema de robots. Realizar estas anotaciones a mano puede ser factible, aunque tedioso, en sistemas con un número de módulos reducido, sin embargo, para sistemas con decenas o cientos de módulos esta tarea consumiría demasiado tiempo y esfuerzo. Es por esta razón por la que decidimos crear el parser de acciones de robots cristalinos.

6.2. Menu principal

Al ejecutar el simulador se abre la ventana del menú principal con cuatro opciones:

6.2.1. Repair Rules File

Esta opción prepara el conjunto de reglas para poder ser analizado correctamente por el parser.

Una vez pulsado este botón, se abre una ventana en donde podemos elegir un fichero de reglas del simulador. Escogido el conjunto de reglas a reparar, el parser genera un nuevo fichero de reglas con dos correcciones:

- Se eliminan los espacios a principio de linea.
- Se asigna una fase inventada, *[WW]*, a las reglas que no tienen indicada una fase a principio de su nombre.

Como estas reparaciones puede que no hagan falta si las reglas se han redactado correctamente, esta acción es opcional.

6.2.2. Numerate and Parse Rules

Al seleccionar esta opción se abre una ventana que permite elegir un fichero de reglas a analizar.

Si las reglas no tienen ninguno de los fallos mencionados en el Apartado 6.2.1 esta opción genera dos nuevos ficheros.

Uno de ellos, terminado en *_numbered*, contiene el conjunto de reglas numeradas a partir de 0. Para que el parser sea capaz de interpretar las acciones que ha realizado cada regla en una reconfiguración es importante que en el simulador se haya abierto este conjunto de reglas numeradas en lugar del conjunto de reglas original. Esta opción no numera ninguna regla que se aplique al árbol generador final ya que este árbol no es más que un artificio de la simulación, en el caso de aplicar las reglas a un robot real, este árbol no existiría.

El segundo fichero, terminado en *_actions*, contiene una lista que consta de cada número que identifica una regla seguido del número de acciones de cada tipo que realiza dicha regla. El orden en que se representan las acciones es:

1. Cambios de estado.
2. Mensajes numéricos emitidos.
3. Mensajes de texto emitidos.
4. Movimientos por la superficie.
5. Valores de registros alterados.
6. Acoplamientos o desacoplamientos.
7. Paso de módulo comprimido.

8. Compresión de módulo.
9. Descompresión de módulo (expansión de un módulo comprimido en un espacio vacío).
10. Mensajes de acción de compresión.
11. Mensajes de confirmación de acción de compresión.
12. Mensajes de negación de acción de compresión.
13. Mensajes de acción de paso de módulo comprimido.
14. Mensajes de confirmación de paso de módulo comprimido.
15. Mensajes de operación de cambio de rama.
16. Mensajes de confirmación de operación de cambio de rama.
17. Mensajes de negación de operación de cambio de rama.
18. Mensajes de paso del estado *líder*.
19. Mensajes de *pausa*.
20. Mensajes de *reanudación*.
21. Mensajes de recuento de módulos.

Si el parser encuentra una acción que no comprende en una regla, escribe *Error* en lugar del número de veces que la regla realiza cada acción.

6.2.3. Parse log File

Esta opción es la responsable de analizar las acciones realizadas durante una o más ejecuciones de un conjunto de reglas. Al seleccionarla abre la ventana de análisis estadístico.

6.2.4. Exit

Esta opción cierra el parser de acciones.

6.3. Ventana de análisis estadístico

Esta ventana consta de varios botones:

- El primer botón, en el que puede leerse *Open*, permite elegir el fichero *_actions* del conjunto de reglas que se ha usado para ejecutar la reconfiguración a analizar.

- El botón de *Add Log* añade a la lista de logs de acciones a analizar el fichero que seleccionemos. Es importante que estas acciones correspondan a las realizadas por un conjunto de reglas numeradas, el mismo conjunto que genera la opción del Apartado anterior 6.2.2 a partir del conjunto de reglas original.

- El botón *Remove Log* elimina el log de la lista de logs a analizar que esté seleccionado.

- El botón *Remove All* elimina todos los logs de la lista de logs a analizar.

- El botón *Parse* analiza los logs de los ficheros de acciones importados del simulador y genera un único fichero común con el análisis de las diferentes acciones generadas por fase de ejecución. Al final del fichero podemos encontrar una media y un total del número de mensajes y movimientos totales del conjunto de logs analizados.

- El botón *Cancel* cierra esta ventana y vuelve a la ventana del menú principal del parser.

6.4. ¿Cómo funciona?

Para poder analizar las acciones realizadas durante una reconfiguración primero necesitamos ejecutar todas las opciones del simulador. Antes de empezar hay que tener en cuenta que todos los archivos generados por el parser serán creados en el mismo directorio en que se encuentra el parser.

Primero, al ejecutar el parser, si el fichero de reglas contiene alguno de los dos errores mencionados en el Aparatado 6.2.1, debemos ejecutar la primera opción, *Repair Rules File*, y una vez generado el nuevo conjunto de reglas substituimos la fase *[WW]* por la fase pertinente.

Como segundo paso debemos ejecutar la opción de *Numerate and Parse Rules*. Esta opción genera los dos ficheros clave para el análisis de las acciones, *Nombre_numbered* y *Nombre_actions*.

El tercer paso es el de ejecutar tantas reconfiguraciones como se desee usando como conjunto de reglas el fichero *Nombre_numbered* generado en el paso anterior. No hay que olvidar que después de cada ejecución hay que guardar en un fichero las acciones que el simulador muestra en la pestaña *actions.log* usando la opción *save* de dicha pestaña.

Por último abrimos la ventana de análisis estadístico y cargamos, mediante el botón *Open*, el fichero *Nombre_actions* generado en el segundo paso. Luego añadimos a la lista de logs a analizar los ficheros de *actions.log* que hemos guardado después de cada iteración. Una vez hemos añadido todos los logs que queremos analizar podemos pulsar *Parse* para generar el fichero con los resultados del análisis.

6.5. Requisitos

Nuestro parser solo necesita los siguientes elementos:

- Una versión de Java compatible con el simulador de robots cristalinos (superior a la versión 6.0).
- El Simulador de robots cristalinos.
- Un sistema operativo de Microsoft Windows superior a Windows 95.

Capítulo 7

Gestión del proyecto

7.1. Planificación

Para realizar la planificación inicial, el proyecto se dividió en 4 bloques:

1. **Programación:** Esta fase incluye el estudio del simulador, diseño e implementación de los algoritmos de mejora e implementación y ejecución de los juegos de prueba.

    a) **Estudio del simulador:** Estudio del funcionamiento del simulador y del lenguaje de sus reglas de actuación.

    b) **Primera versión del algoritmo:** Diseño e implementación de un algoritmo simple con señal de parada hasta intersección. Incluye la creación y ejecución de los juegos de prueba correspondientes.

    c) **Segunda versión del algoritmo:** Diseño e implementación de un algoritmo con señal de parada hasta la raíz del árbol e implementación de una segunda cadena de señales de reanudado. Incluye la ejecución de los juegos de prueba correspondientes.

    d) **Tercera versión del algoritmo:** Diseño e implementación de un algoritmo con señal de parada para toda la configuración. Incluye la ejecución de los juegos de prueba correspondientes.

    e) **Versión final del algoritmo:** Diseño e implementación del algoritmo multilíder. Incluye la ejecución de los juegos de prueba correspondientes.

2. **Experimentación:** Esta fase esta formada por el estudio de la complejidad de los algoritmos y la experimentación de juegos de prueba más avanzados para obtener resultados reales sobre el rendimiento de los algoritmos.

   a) **Estudio de complejidad:** Estudio de la complejidad de los 5 algoritmos del proyecto.
   b) **Experimentación:** Ejecución de una serie de jugos de prueba específicos para medir el rendimiento de los algoritmos estudiados en el proyecto así como el análisis de dichos resultados.

3. **Página web:** En esta fase se planificó la creación o modificación de la página web del proyecto sobre algoritmos distribuidos de robots cristalinos.

4. **Memoria:** Esta fase abarca la redacción de la memoria del proyecto.

5. **Presentación:** Por último, en esta fase se produce la preparación de la defensa del proyecto.

El cálculo de horas inicial fue complejo debido a que, al tratarse de un proyecto de investigación, es difícil llegar a hacerse una idea de todos los posibles problemas que pueden surgir.

La metodología seguida en cada fase es la siguiente:

- Se planifica la fase y se anotan los posibles imprevistos y dudas que hayan surgido durante la planificación.
- Se presenta la planificación a la directora del proyecto, se discute dicha planificación y se decide la mejor manera de continuar el proyecto.
- Implementación de la fase y obtención de resultados.
- Se presentan los resultados de la fase a la directora para su validación.
- Redacción en la memoria del tema relacionado con la fase que se ha implementado.

Podemos ver el diagrama de gantt inicial en la Figura 7.3. Durante el proyecto se ha modificado el calendario a medida que nos hemos encontrado con diferentes problemas e incidentes. La figura 7.4 muestra el diagrama de gantt final.

Como se puede apreciar en el diagrama final, las fechas de la implementación del algoritmo multilíder, experimentación y redacción de la memoria se han alejado bastante de las fechas originales debido a los problemas encontrados durante la implementación del algoritmo y la experimentación. En estas fases han intervenido factores no previstos como la excesiva aparición de errores durante la implementación del algoritmo, el fallo del algoritmo de Joan Soler durante la experimentación y el tiempo de ejecución de los juegos de prueba del simulador. Estos problemas o no se previeron durante

la planificación o bien, en el caso de la implementación del algoritmo multilíder o el tiempo de ejecución de los juegos de prueba, han necesitado una mayor inversión de tiempo de la prevista.

Además, en el diagrama final se ha añadido una nueva sub-fase a la experimentación dedicada a la modificación del algoritmo de Joan Soler.

7.2. Presupuesto

En la valoración económica de este proyecto no se ha tenido en cuenta el coste del simulador ya que forma parte de otros proyectos anteriores a este.

Todos los recursos utilizados son humanos. El material usado es digital y libre, por lo que su coste no tiene relevancia en el presupuesto.

En el proyecto han intervenido dos personas: la directora del proyecto y el analista y programador. La Figura 7.1 muestra el precio por hora de cada una de las personas implicadas en el proyecto.

Trabajo	Precio/hora
Director de proyecto	55,00 €
Analista	40,00 €
Programador	25,00 €

Figura 7.1: Tabla de precio por hora según rol.

En la Figura 7.2 se muestran los recursos utilizados, las horas invertidas y los costes de cada fase.

Trabajo	Horas	Recursos	Precio
Estudio del simulador	30	Analista	1.200,00 €
Primera versión del algoritmo	37	Programador	925,00 €
Segunda versión del algoritmo	36	Programador	900,00 €
Tercera versión del algoritmo	39	Programador	975,00 €
Última versión del algoritmo	80	Programador	2.000,00 €
Modificación del algoritmo original	20	Programador	500,00 €
Estudio de la complejidad	75	Analista	3.000,00 €
Experimentación	160	Analista	6.400,00 €
Página web	4	Programador	100,00 €
Redactar la memoria	270	Analista	10.800,00 €
Presentación	15	Analista	600,00 €
Reuniones de seguimiento	40	Director y Analista	3.800,00 €
TOTAL	806		31.200,00 €

Figura 7.2: Desglose económico del proyecto.

Figura 7.3: Diagrama de gantt inicial.

Figura 7.4: Diagrama de gantt final.

Capítulo 8

Conclusiones

8.1. Resultados obtenidos

El objetivo principal del trabajo era conseguir mejorar el algoritmo distribuido original para conseguir una reducción en el número de movimientos. No solo hemos conseguido reducir el número de movimientos de forma espectacular, sino que, además, hemos conseguido reducir la comunicación entre módulos a una pequeña fracción de lo que se generaba en el algoritmo original.

En un primer momento se consideró el estudio de un algoritmo distribuido 3D pero al final, al ver que el algoritmo era idéntico al algoritmo 2D pero añadiendo reglas para las 2 nuevas direcciones posibles (encima y debajo), vimos que el estudio carecía de interés y optamos por abandonar este objetivo.

Además, hemos analizado la complejidad de todos los algoritmos de mejora presentados y hemos experimentado con ellos para comprobar, no solo su eficiencia real, sino también que la complejidad real se ajusta a la teorizada.

También hemos creado un analizador de acciones que nos permite extraer, a partir de un log del simulador, todos los movimientos y mensajes realizados por la simulación así como su tipo.

Por último hemos conseguido encontrar patrones que en un futuro ayudarán a mejorar la eficiencia del algoritmo como la importancia de la densidad de módulos en una figura y la de su orientación respecto el universo del simulador.

8.2. Dificultades encontradas

La primera dificultad que hemos encontrado ha sido el lenguaje del simulador. Comprender en profundidad el lenguaje del simulador ha sido difícil ya que nos hemos encontrado con algunas situaciones como el caso en que el

simulador acepta un asterisco a la hora de indicar que puedes recibir un mensaje de cualquier dirección, por ejemplo *M*Expnd*, pero no lo acepta cuando quieres comprobar la negación de la expresión, por ejemplo *!M*Expnd*, obligando a alargar las precondiciones de las reglas.

La manera en que el simulador interpreta las reglas también ha sido un problema. El simulador comprueba, para cada módulo, qué reglas pueden ejecutarse y, después de validar todas las reglas posibles, las ejecuta sin comprobar nada más. De esta forma, si una de las reglas que el simulador da como válida altera el estado de un módulo, haciendo así que deje de cumplir la precondición del resto de reglas válidas, el simulador sigue ejecutando el resto de reglas aunque no se cumpla su precondición. Para solucionarlo hemos tenido que utilizar toda una serie de estados y contadores que, en muchas ocasiones, evitan que se ejecute más de una regla por iteración, entorpeciendo la reconfiguración y haciendo que dure más ejecuciones de las que debería, y obligando a crear muchas reglas adicionales.

La gran cantidad de reglas existentes también han sido una gran fuente de problemas. Asegurar que más de 700 reglas no van a interferir unas con otras es prácticamente imposible. Cada vez que encontrábamos un problema y creábamos un nuevo conjunto de reglas para solucionarlo, surgían problemas nuevos causados por la interferencia de las reglas nuevas con las antiguas en situaciones extremadamente variadas.

Cuando no había problema con las reglas nuevas, aparecían circunstancias imprevistas. En ejemplos pequeños, de menos de 100 módulos, ninguno de los algoritmos estudiados en el proyecto daba problemas, sin embargo, al incrementar el número de módulos, empezaban a surgir casos extremos en donde incluso el conjunto de reglas del algoritmo de Joan Soler [3] dejaba de funcionar correctamente e interfería con otras reglas. En concreto, en el caso de las reglas de Joan Soler, se han llegado a encontrar problemas incluso durante las últimas semanas del proyecto.

Por último cabe resaltar la estabilidad del simulador. Ya sea por el cambio a las nuevas versiones de java, por el hecho de estar programado en java o por el gran número de módulos de algunos ejemplos, el simulador en ocasiones deja de funcionar al llegar a la última ejecución de la reconfiguración. Esto era un problema a la hora de realizar la experimentación porque dejaba de funcionar antes de ejecutar la opción, introducida por consola, que obliga al simulador a guardar el log de acciones una vez acabada la configuración e imposibilita que pueda volver a ejecutar otro caso de prueba. Para solucionarlo, aprovechando que el simulador guarda las acciones realizadas en un archivo temporal, creamos un script para powershell de windows que abría el simulador, ejecutaba el caso de prueba, al cabo de un tiempo guardaba el log de acciones y luego mataba el proceso del simulador en el sistema para volverlo a abrir con un nuevo juego de prueba.

8.3. Futuro del proyecto

Algunas de las tareas que podrían llevarse a cabo en un futuro podrían ser:

- *Ampliar el algoritmo multilíder:* Aunque el algoritmo multilíder evita la gran mayoría de los movimientos innecesarios durante la fase de expansión, podríamos intentar reducirlo aún más desde la fase de compresión o incluso desde el momento en que se busca la raíz. Si se conseguimos que un módulo sepa si ya ocupa una posición que existe en la forma final y que existe un camino de módulos ya en posición desde el módulo a la raíz, entonces podríamos dejar ramas enteras del árbol generador inicial en fase de expansión incluso antes de iniciar la fase de compresión.

- *Integración del analizador de acciones:* En un futuro podríamos integrar el analizador de acciones en el simulador siempre y cuando este permita al usuario añadir nuevos patrones a reconocer como si de un conjunto de reglas se tratara.

- *Implementar un sistema threads:* En ocasiones, sobretodo para casos de prueba con muchos módulos, el simulador tarda demasiado en ejecutar la reconfiguración y no llega a aprovechar toda la potencia del ordenador que lo ejecuta. Para evitarlo podríamos estudiar si creando una serie de threads durante la ejecución se puede llegar a reducir el tiempo que lleva el terminar una reconfiguración.

- *Modo consola:* Aunque la gran mayoría de las veces nos interesa ver paso a paso la ejecución de una reconfiguración, en ocasiones no es necesario. Para intentar ahorrar tiempo en la ejecución de la reconfiguración y facilitar la experimentación podría ser interesante crear un modo de ejecución solo por consola, que no mostrara ningún tipo de ventana ni imagen.

8.4. Valoración personal

La verdad es que durante la planificación e implementación de los diferentes algoritmos he disfrutado muchísimo. Horas y horas implementando nuevas versiones, solucionando errores, mirando paso por paso la ejecución, preguntándome qué pieza del rompecabezas se me escapaba para, finalmente, verlo funcionar a la perfección. Es por estos momentos por lo que más he disfrutado este trabajo. Me hubiera gustado ampliar aún más el último algoritmo, llevarlo al extremo de generar reconfiguraciones perfectas sin movimientos innecesarios, pero mi tiempo con este proyecto ha llegado a su fin.

Una vez acabado, incluso con los resultados de la experimentación en la mano y viendo lo que ha mejorado el algoritmo con su versión multilíder, soy consciente que solo he realizado una pequeña parte de un proyecto mucho más grande, con mucho historia y con mucho futuro y potencial.

Además, durante el tiempo en que he estado trabajando en este proyecto he aprendido mucho más sobre Java[7][8], html5[9][10], powershell[11][12][13][14][15][16][17] y LaTeX[18][19][20][21][22][23][24][25][26][27][28][29][30].

Por último me gustaría dar las gracias a Vera Sacristán y a su dedicación en este proyecto. Con su ayuda, y sobretodo su paciencia, me ha ayudado en mis momentos más bajos y de menor motivación y me ha animado a continuar con el proyecto cuando ya me flaqueaban las fuerzas. Ayuda mucho saber que hay alguien interesado en tu trabajo y que te ayuda y plantea nuevas ideas que en su momento no habías sido capaz de ver.

Gracias.

Bibliografía

[1] Daniela Rus, Marsette Vona. A Physical Implementation of the Self-reconfiguring Cyristalline Robot. Proceedings of the IEEE Intl. Conference on Robotics and Automation pp. 1726–1733, 2000.

[2] G. Aloupis, S. Collette, M. Damian, E. D. Demaine, R. Flatland, S. Langerman, J. O'Rourke, V. Pinciu, S. Ramaswami, V. Sacristán, S Wuhrer. Efficient Constant-Velocity Reconfiguration of Crystalline Robots. Robotica, Vol. 29, N. 1, pp. 59-71, 2011.

[3] J. Soler. Reconfiguració de robots cristal·lins (in Catalan). Degree thesis under the supervision of V. Sacristán, Facultat de Matemàtiques i Estadística, Universitat Politècnica de Catalunya, 2013.

[4] J. W. Suh, S. B. Homans and M. Yim, "Telecubes: Mechanical Design of a Module for Self-Reconfigurable Robotics," Proceedings of the IEEE International Conference on Robotics and Automation, Washington, DC (May 11–15, 2002) pp. 4095–4101.

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

[6] http://www-ma2.upc.edu/vera/CrystalSimulation/.

[7] http://stackoverflow.com/questions/7442310/adding-elements-to-jlist-in-swing-java.

[8] http://stackoverflow.com/questions/4005378/console-writeline-and-system-out-println.

[9] http://www.w3schools.com/html/html5_video.asp.

[10] https://developer.mozilla.org/en-US/docs/Web/Guide/HTML/Using_HTML5_audio_and_video.

[11] https://technet.microsoft.com/en-us/magazine/2008.12.heyscriptingguy.aspx.

[12] https://www.youtube.com/watch?v=2IC-YbzUZAc.

[13] http://stackoverflow.com/questions/17849522/how-to-perform-keystroke-inside-powershell.

[14] https://technet.microsoft.com/en-us/library/ee156818.aspx.

[15] http://stackoverflow.com/questions/19824799/how-to-send-ctrl-or-alt-any-other-key.

[16] http://stackoverflow.com/questions/9788492/powershell-extract-file-name-and-extension.

[17] http://stackoverflow.com/questions/18847145/loop-through-files-in-a-directory-using-powershell.

[18] http://elclubdelautodidacta.es/wp/2013/03/latex-sumatorio-y-productorio/.

[19] http://tex.stackexchange.com/questions/69373/how-to-write-properly-math-accent-for-letter-i.

[20] http://en.wikibooks.org/wiki/LaTeX/List_Structures.

[21] https://es.sharelatex.com/learn/Sections_and_chapters.

[22] http://en.wikibooks.org/wiki/LaTeX/Mathematics.

[23] http://en.wikibooks.org/wiki/LaTeX/List_Structures#Itemize.

[24] http://www.latex-community.org/forum/viewtopic.php?f=46&t=21257.

[25] http://texblog.net/help/latex/markboth.html.

[26] http://pgfplots.sourceforge.net/.

[27] https://es.sharelatex.com/learn/Pgfplots_package.

[28] http://tex.stackexchange.com/questions/45529/pgfplot-axis-mark.

[29] https://www.sharelatex.com/learn/Pgfplots_package.

[30] http://tex.stackexchange.com/questions/31276/number-format-in-pgfplots-axis.

Anexo: Definiciones

En este anexo describimos no solo la estructura de nuestros conjuntos de reglas y la función de todos y cada uno de los estados, contadores y mensajes que utilizan.

Cada apartado describe los estados, contadores y mensajes que usa cada conjunto de reglas y que difieren de los descritos en los apartados anteriores de este mismo anexo.

Clasificación en las tres fases principales de los algoritmos algoritmo

Todas las reglas tanto del algoritmo original como de las modificaciones están clasificadas en tres fases según cuando y para qué se utilizan. Esto se expresa el principio de cada regla con una sigla entre corchetes que, aunque no tiene efecto en el algoritmo, ayuda al usuario a comprender mejor las reglas. A continuación describimos dichas fases:

[S] Fase inicial: Esta fase consta de las reglas que buscan la raíz del árbol generador inicial y que dan a cada módulo el valor inicial de sus registros y su estado inicial. Cada módulo que no esta conectado a ningún otro módulo ni por el norte ni por el oeste se considera a si mismo como candidato a raíz y envía un mensaje a sus vecinos con sus coordenadas relativas. Estos vecinos a su vez envían el mismo mensaje a sus vecinos con las coordenadas actualizadas. Si un módulo recibe dos mensajes de diferente origen (coordenadas) elije el mensaje del módulo que esta más al noroeste y lo transmite. Para evitar ciclos, un módulo que recibe el mismo mensaje de más de una dirección reconoce que hay un ciclo en el grafo de conexiones y rompe una de estas. Cuando un mensaje llega a un módulo hoja, este vuelve a ser enviado en dirección a su padre. Finalmente, si un módulo candidato a raíz recibe su propio mensaje, se reconoce raíz y empieza la fase de compresión.

[C] Fase de compresión: La fase de compresión consta de reglas que se aplican una vez se ha encontrado una raíz en el árbol generador inicial. Esta raíz envía una señal que viaja por todo el árbol y una vez ha llegado a las hojas provoca que estas se compriman en sus padres y viajen a través de las ramas del árbol hacia la raíz. Este proceso de compresión y movimiento

se repite siempre que sea posible. Esta fase se ejecuta al mismo tiempo que la fase de expansión.

[E] Fase de expansión: Consta de las reglas que se aplican al llegar a la raíz el primer módulo comprimido, siempre que la raíz tenga toda la información disponible para empezar la reconfiguración (una vez la raíz del árbol generador ha encontrado la raíz del árbol final). Esta fase expande una señal de líder que viaja desde la raíz del árbol siguiendo el camino indicado por la información del árbol generador final para acabar formando, rama a rama, la figura deseada. Para conseguirlo, se transmite la señal de un módulo en fase de expansión a otro o, en caso de encontrar una posición vacía, expandiendo módulos comprimidos para llenar esos huecos. La dirección en que se expande la señal tiene un sistema de prioridades: primero se intenta expandir hacia el sur, luego hacia el este, hacia el oeste y por último hacia el norte. Para facilitar la expansión, los módulos comprimidos siguen al líder a través de las ramas. Esta fase se ejecuta al mismo tiempo que la fase de compresión.

Fases o grupos secundarios

[F] Fase inicial de la copia de la forma final: La representación de la forma final y todas las reglas que se aplican sobre ella es la manera que tenemos en nuestro simulador de simular que la raíz tiene la información de la forma a la que ha de llegar. Por eso cuando analizamos y estudiamos las reglas aplicadas a los diferentes juegos de prueba nunca tenemos en cuenta las reglas que se le han aplicado. Las reglas de este grupo son las mismas que se aplican a la configuración inicial.

[R] Reglas de reparación: Reglas auxiliares que se aplican en todos los módulos independientemente de la fase en la que se encuentren, no pertenecen a una fase específica. Normalmente son reglas que mantienen contadores o que expanden señales que han de viajar por todo el árbol independientemente de en qué fase se encuentre cada módulo.

[End] Fin de la reconfiguración: En la última modificación del algoritmo existe esta última fase o grupo de reglas compuesto por exactamente cuatro reglas. Su única función es la de conectar entre sí módulos vecinos que ya hayan acabado su trabajo en el árbol actual y que ya no deban realizar ninguna otra acción para alcanzar la forma final.

Algoritmo original

Descripción de estados

Start: Estado inicial de todos los módulos de la configuración inicial. Es necesario asignar a los módulos este estado antes de empezar a aplicar el

algoritmo.

Final: Estado inicial de todos los módulos de la configuración final. Es necesario asignar a los módulos este estado antes de empezar a aplicar el algoritmo.

CanbS: Estado que adquieren los módulos de la configuración inicial que no tienen ningún vecino ni al norte ni al oeste al iniciar la reconfiguración. Estos módulos son candidatos a ser raíz del árbol generador.

CanbF: Estado que adquieren los módulos de la representación de la forma final que no tienen ningún vecino ni al norte ni al oeste al iniciar la reconfiguración. Estos módulos son candidatos a ser raíz del árbol generador.

WaitS: Estado que adquieren los módulos de la configuración inicial que han recibido el mensaje enviado por un módulo candidato a raíz del árbol. Incluso si el módulo que recibe el mensaje es un candidato a raíz del árbol este también cambia a este estado ya que recibirlo indica la existencia de un candidato mejor.

WaitF: Estado que adquieren los módulos de la representación de la forma final que han recibido el mensaje enviado por un módulo candidato a raíz del árbol. Incluso si el módulo que recibe el mensaje es un candidato a raíz del árbol este también cambia a este estado ya que recibirlo indica la existencia de un candidato mejor.

ForwS: Cuando un módulo en estado *WaitS* difunde el mensaje que ha recibido de un candidato a raíz del árbol generador inicial, pasa a estado *ForwS*. Permanece en este estado hasta recibir un mensaje que proviene de una hoja del árbol o hasta recibir otro mensaje de un candidato a raíz mejor que el recibido anteriormente.

ForwC: Cuando un módulo en estado *WaitC* difunde el mensaje que ha recibido de un candidato a raíz del árbol generador final, pasa a estado *ForwC*. Permanece en este estado hasta recibir un mensaje que proviene de una hoja del árbol o hasta recibir otro mensaje de un candidato a raíz mejor que el recibido anteriormente.

BackS: Una vez el mensaje enviado por un candidato a raíz ha llegado hasta una hoja del árbol generador inicial esta entra en estado *BackS* y envía un mensaje hacia la raíz. Todos los módulos que reciben este mensaje adquieren este estado. Si por el camino un módulo en este estado recibe el mensaje de un candidato a raíz mejor que el actual vuelve a estado *WaitS*.

BackF: Una vez el mensaje enviado por un candidato a raíz ha llegado hasta una hoja del árbol generador final esta entra en estado *BackF* y envía un mensaje hacia la raíz. Todos los módulos que reciben este mensaje adquieren este estado. Si por el camino un módulo en este estado recibe el mensaje de un candidato a raíz mejor que el actual vuelve a estado *WaitF*.

RootS: Estado al que pasa un módulo candidato a raíz del árbol generador inicial cuando recibe su propio mensaje de vuelta de todos sus hijos. Una vez alcanzado este estado el módulo envía a sus hijos un mensaje para comenzar la fase de compresión.

RootF: Estado al que pasa un módulo candidato a raíz del árbol generador final cuando recibe su propio mensaje de vuelta de todos sus hijos.

RootL: Cuando el estado de líder pretende pasar de una rama del árbol generador actual a la otra (del hijo sur de la raíz del árbol al hijo del este) este tiene que pasar por la raíz del árbol. Para indicar que la raíz es el líder actual del árbol esta pasa al estado *RootL*.

RootP: Cuando una raíz del árbol generador actual en estado *RootL* expande un módulo comprimido a una posición vacía, pasa a estado *RootP*. Este estado se mantiene por una iteración. Pasado este tiempo el módulo vuelve a estado *RootL*.

LIDER: Este estado otorga al módulo que lo obtiene la potestad de decidir la dirección por la que debe expandirse el árbol. Según pasa de módulo a módulo esta señal deja una marca que indica a los módulos comprimidos que llegan la dirección a seguir. Este estado surge de la raíz del árbol generador actual. La reconfiguración acaba cuando el estado vuelve a la raíz una vez alcanzada la forma final.

PLIDR: Estado que tiene un módulo que ha sido expandido a una posición vacía. Este estado se conserva durante una iteración, la misma en la que el módulo ha sido descomprimido. Pasado este tiempo el módulo obtiene el estado *LIDER*.

PExpn: Estado utilizado para proteger a un módulo que acaba de entregar su estado de líder de cualquier otra regla que se encuentre entre la que le ha obligado a entregar su estado y la regla que le hace pasar del estado *PExpn* a *Expnd*. Este estado solo se mantiene durante una fracción de iteración ya que se obtiene y se pierde durante la misma iteración.

Expnd: Cuando un módulo entrega su estado de líder, pasa a estado *Expnd* (después de pasar por el estado *PExpn* por una fracción de iteración). Este estado indica que un módulo ya ha llegado a la posición que le corresponde en la configuración final.

Cmprs: Estado de los módulos que reciben el mensaje de la raíz del árbol generador inicial una vez esta ha sido escogida. Este estado indica a las hojas del árbol que deben comprimirse en dirección a la raíz y a los módulos ya comprimidos que deben moverse por las ramas del árbol en dirección a la raíz. Un módulo pierde este estado al entrar en estado de líder.

Slave: Estado de los módulos que reciben el mensaje de la raíz del árbol generador final una vez esta ha sido escogida.

Descripción de registros

C00: Este registro guarda la posición relativa del de cada módulo tanto del árbol generador inicial como del árbol generador final respecto de la raíz de cada árbol. Ambas raíces dan a este registro el valor de 5050 (50 de coordenada x y 50 de coordenada y).

C01: Cada módulo guarda en este registro la dirección que hay que seguir

desde su posición para llegar viajando a través de las ramas del árbol a su raíz, ya sea del árbol final o inicial. Esta dirección se expresa mediante un número de cuatro cifras. El valor 1000 indica la dirección norte, 100 indica el oeste, 10 el este y 1 el sur.

C02: El valor de este registro es la suma de los valores 1000, 100, 10 y 1 dependiendo de si en ese momento el módulo tiene algún hijo en la dirección norte, oeste, este o sur respectivamente.

C04: Dirección en la que el módulo debe tener hijos y todavía no lo tiene. Por ejemplo, considerando las cuatro direcciones como 1000, 100, 10 y 1 (norte, oeste, este y sur respectivamente) si un módulo lee del árbol generador final que debería tener un hijo al oeste, otro al este y otro al sur pero actualmente solo tiene un hijo al este el valor de C04 sería de 0101 para registrar la falta de hijos al oeste y al sur.

C05: Cuando un módulo entrega su estado de líder, guarda en este registro la dirección por la que se puede encontrar el nuevo líder. De esta forma los módulos comprimidos pueden encontrar siempre el camino a seguir hacia el líder.

C16: Registro que indica que un módulo hoja que ha pedido confirmación para comprimirse además ha recibido una señal de cambio de rama. El valor de este registro es la dirección por la que ha recibido la señal de cambio de rama, es decir 1000, 100, 10 o 1 (norte, oeste, este o sur respectivamente). Si el valor de este registro es diferente de 0 y el módulo recibe una señal de confirmación de compresión, utiliza la información del registro para enviar un mensaje negando la operación de cambio de rama y luego da al registro el valor de 3000. El valor 3000 en este registro indica que el módulo esta listo para comprimirse.

C17: Señal que cuenta el número de iteraciones que ha pasado desde que un módulo hoja en compresión pidió permiso para comprimirse. Si el módulo no recibe ninguna confirmación para comprimirse en 2 iteraciones pero sí que ha recibido una señal de cambio de rama, ejecuta el cambio de rama en vez de volver a pedir permiso para comprimirse.

C18: Booleano que indica que un módulo hoja ha enviado un mensaje pidiendo permiso para comprimirse. Su valor es 1 cuando ha pedido permiso y 0 si aún no lo ha pedido, si han pasado 2 iteraciones desde que lo pidió y si esta ejecutando la acción de cambio de rama.

C20: Este registro tiene dos funciones. La primera es la de contador para retrasar la ejecución de otras reglas sobre el módulo raíz del árbol generador inicial cuando este ha empieza a expandir su hijo del sur o del oeste. La segunda es la de señalar en la dirección en la que se ha enviado un mensaje que pueda cambiar la fase de un módulo de compresión a expansión (es decir, un mensaje de cambio de rama o de paso de líder a un módulo en fase de compresión). Aunque a primera vista pueda parecer que tiene la misma función que el registro C005 en realidad son muy diferentes, mientras que C005 solo toma valor una vez confirmado el paso del estado *líder* y

permanece con ese valor hasta la vuelta de dicho estado, C020 solo tiene valor diferente de 0 desde el momento en que se envía el mensaje de cambio de líder (las señales descritas antes en este mismo párrafo) hasta que se confirma el cambio.

C22: Booleano que evita que se cree más de un líder cuando el módulo raíz adquiere dicho estado. Solo tiene valor 1 en el módulo raíz.

C23: Es la distancia a la que un módulo del árbol generador inicial sabe que encontrará el módulo del árbol final que tiene los datos que necesita para completar la configuración. Es la manera que tenemos de simular que la raíz del árbol generador inicial conoce los datos necesarios para completar la configuración. El valor de este registro lo calcula la raíz del árbol generador inicial y lo transmite al pasar el estado de *líder*.

C24: Al comprimir un módulo, el módulo que acoge al comprimido actualiza el valor de este registro para marcar que ahora es un módulo comprimido. Su valor es siempre 1 si el módulo esta comprimido y 0 si no lo está.

C25: Al comprimir un módulo, el módulo que se comprime actualiza el valor de este registro para marcar que ahora contiene un módulo comprimido. Durante algunas operaciones de compresión o de envío de módulo comprimido también hace a la vez de contador, por lo que a veces su valor es 2.

Algoritmo con señal de parada hasta intersección

Descripción de estados

Pause: Este es el estado al que pasan los módulos que reciben una señal numérica de pausa. Esto mantiene a los módulos pausados y sin realizar acción alguna hasta recibir la señal de líder, momento en el cual vuelven a estado *LIDER*.

También existen en este algoritmo todos los estados descritos para el algoritmo original en el Apartado 8.4.

Descripción de registros

C08: Registro que marca los módulos que han sido pausados alguna vez. Se usa para evitar enviar más de un mensaje de pausa por rama. Aunque hubiera sido más complicado, esta modificación del algoritmo original se podría haber levado a cabo sin este registro (aunque con muchas más reglas). Sin embargo, como veremos en el Apartado 8.4, este registro nos permite evitar la repetición de envío de otro tipo de señal.

Esta modificación conserva además la función de todos los registros descritos para el algoritmo original en el Apartado 8.4.

Algoritmo con señal de parada hasta raíz

Descripción de estados

Los estados que usa esta modificación son los descritos en el Apartado 8.4.

Descripción de registros

C08: Además de la función especificada en el Apartado 8.4, ahora este registro también evita que se envíe más de una señal de reanudación al expandir una nueva rama. Como todos los módulos que no han sido pausados (esto incluye los módulos recién expandidos) tienen el valor de este registro a 0, el algoritmo solo envía la señal de reanudación si un módulo líder intenta expandirse mientras el valor de su registro C008 es 1, es decir, no envía nunca el mensaje de reanudación al expandir ni la primera rama de la reconfiguración ni, una vez enviado el mensaje de reanudación y con el estado *líder* perteneciendo a un módulo recién expandido, al continuar expandiendo ninguna otra rama.

Esta modificación conserva además la función de todos los registros descritos en el Apartado 8.4.

Algoritmo con señal de parada para toda la configuración

Descripción de estados

Los estados que usa esta modificación son los descritos en el Apartado 8.4.

Descripción de registros

Las funciones de todos los registros en esta modificación son los descritos en el Apartado 8.4.

Versión multilíder del algoritmo

Descripción de estados

ASKL1: Estado al que pasa un módulo líder cuando un vecino al que está conectado le pide permiso para enviarle un módulo comprimido. El primero de un grupo de dos estados que controlan y protegen el proceso de paso de módulo comprimido en módulos líder.

ASKL2: Estado que sigue a *ASKL1* para proteger de interferencias de otras reglas al módulo líder mientras recibe un módulo comprimido. En

caso de no recibir en dos iteraciones el módulo comprimido se restauran los registros del módulo que esperaba recibirlo y se cambia su estado a *LIDER* de nuevo.

ASKE1: Estado al que pasa un módulo en fase de expansión cuando un vecino al que está conectado le pide permiso para enviarle un módulo comprimido o cuando un módulo en fase de expansión pide permiso para enviar un módulo comprimido. El primero de un grupo de dos estados que controlan y protegen el proceso de paso de módulo comprimido en módulos en fase de expansión.

ASKE2: Estado que sigue a *ASKE1* para proteger de interferencias de otras reglas a un módulo en fase de expansión mientras recibe o envía un módulo comprimido. En caso de no recibir en dos iteraciones el módulo comprimido o el permiso de envío del módulo comprimido, se restauran los registros del módulo que esperaba recibirlo y se cambia su estado a *Expnd*.

ASKC1: Estado al que pasa un módulo en fase de compresión cuando un vecino al que está conectado le pide permiso para enviarle un módulo comprimido o bien cuando un módulo en fase de compresión pide permiso para enviar un módulo comprimido. El primero de un grupo de dos estados que controlan y protegen el proceso de paso de módulo comprimido en módulos en fase de compresión.

ASKC2: Estado que sigue a *ASKC1* para proteger de interferencias de otras reglas al módulo en fase de compresión mientras recibe o envía un módulo comprimido. En caso de no recibir en dos iteraciones el módulo comprimido o el permiso de envío de módulo comprimido, se restauran los registros del módulo que esperaba recibirlo y se cambia su estado a *Cmprs*.

CmD*1: Para evitar la ejecución cíclica de los estados *ASKC1*, *ASKC2*, y *Cmprs* se creó este estado que asume un módulo que se encontraba en estado *ASKC1* cuando ha recibido una señal de cambio de rama. El asterisco indica la dirección por la que ha recibido la señal de cambio de rama (N para norte, W para oeste, E para este y S para sur).

CmD*2: Estado que sigue a *CmD*1*. Si en la iteración siguiente a obtener este estado no se ha recibido el módulo comprimido o no se ha recibido confirmación para comprimirse, en vez de volver al estado *Cmprs* se pasa al estado *DISA** el cual obliga a realizar un cambio de rama. El asterisco indica la dirección por la que ha recibido la señal de cambio de rama (N para norte, W para oeste, E para este y S para sur).

DISA*: Estado que fuerza al módulo a realizar una operación de cambio de rama para conectarse a la rama indicada por el asterisco (N para la rama al norte, W para la rama al oeste, E para la rama al este y S para la rama al sur). Se obtiene cuando un módulo en compresión ha pasado dos iteraciones esperando a recibir un módulo o una señal de otro módulo vecino sin recibir nada y, mientras esperaba, otro vecino le ha enviado una señal de cambio de rama.

ZIPNW: Para solucionar problemas como los descritos en el Apartado

3.1.4 se alargó a dos iteraciones el tiempo necesario para realizar la compresión de una hoja tras recibir permiso para comprimirla. Durante la primera iteración se adquiere este estado y en la segunda se restaura el estado del módulo hoja a *Cmprs* y se comprime en dirección al padre. Si durante este estado el módulo hoja recibe un mensaje de cambio de rama responde con un aviso al emisor del mensaje negando que se vaya a realizar el cambio.

El resto de estados que usa esta modificación son los descritos en el Apartado 8.4.

Descripción de registros

C05: En esta modificación este registro ha pasado a indicar la dirección por la cual un módulo debe enviar o expandir cualquier módulo comprimido que reciba. Su valor depende del número de módulos comprimidos que hagan falta en cada una de las cuatro direcciones del módulo para llegar a la forma final. La dirección que necesita más módulos tiene prioridad sobre las demás. En caso de empate se utiliza la prioridad de expansión del algoritmo original (S, W, E y N). El valor 1000 indica el norte, 100 el oeste, 10 el este y 1 el sur.

C06: Registro auxiliar en el que se guarda durante una iteración el número de módulos que la rama en la que se encuentra el módulo ha ganado o perdido a causa de una operación de cambio de rama.

C07: Booleano que protege a los módulos que acaban de expandir un módulo comprimido en una posición vacía para evitar que ejecuten otras reglas durante esa iteración. Esta función puede llevarse a cabo con un cambio de estado como los que tiene este algoritmo para otras operaciones como el paso de módulos comprimidos, pero para simplificar el código y no aumentar aún más el número de reglas, se decidió asignar la función a un registro.

C08: Para conseguir que cuando el estado *líder* vuelva a la raíz se espera a recibir tantas señales de asignación de líder como líderes ha expandido un módulo, se utiliza este registro como contador. Por cada líder expandido aumenta, y con cada señal de asignación de líder recibida disminuye. Cuando el valor del registro es 0, el módulo tiene estado *líder* y además ya no debe expandirse más, se envía el estado *líder* en dirección a la raíz.

C10: Número de descendientes que cuelgan en cada momento de un módulo en dirección norte. Si se puede llegar a la raíz del árbol generador inicial por esta dirección su valor es 0.

C11: Número de descendientes que cuelgan en cada momento de un módulo en dirección oeste. Si se puede llegar a la raíz del árbol generador inicial por esta dirección su valor es 0.

C12: Número de descendientes que cuelgan en cada momento de un módulo en dirección este. Si se puede llegar a la raíz del árbol generador inicial por esta dirección su valor es 0.

C13: Número de descendientes que cuelgan en cada momento de un

módulo en dirección sur. Si se puede llegar a la raíz del árbol generador inicial por esta dirección su valor es 0.

C14: Total del número de descendientes que cuelgan en cada momento de un módulo en todas sus direcciones.

C15: Booleano que indica si un módulo contiene otro módulo comprimido o no. Su valor es 0 en caso positivo y 1 en caso negativo.

C16: Número de módulos que hacen falta en dirección norte para alcanzar el número de descendientes en esa dirección que se necesiten en la forma final. El valor es negativo si deben llegar módulos comprimidos por esa dirección y positivo si todavía es necesario mandar módulos en esa dirección.

C17: Número de módulos que hacen falta en dirección oeste para alcanzar el número de descendientes en esa dirección que se necesiten en la forma final. El valor es negativo si deben llegar módulos comprimidos por esa dirección y positivo si todavía es necesario mandar módulos en esa dirección.

C18: Número de módulos que hacen falta en dirección este para alcanzar el número de descendientes en esa dirección que se necesiten en la forma final. El valor es negativo si deben llegar módulos comprimidos por esa dirección y positivo si todavía es necesario mandar módulos en esa dirección.

C19: Número de módulos que hacen falta en dirección sur para alcanzar el número de descendientes en esa dirección que se necesiten en la forma final. El valor es negativo si deben llegar módulos comprimidos por esa dirección y positivo si todavía es necesario mandar módulos en esa dirección.

C20: Al tener información de sobra sobre el estado del árbol generador actual en cada momento este registro conserva solo una de sus funciones originales, la de indicador de que se ha enviado un mensaje de algún tipo.

C21: Booleano que indica si el estado *líder* de un módulo es por expansión o porque esta esperando a que lleguen otras señales de asignación de líder para volver a la raíz. Su valor es 0 si el estado es de expansión y 1 si se trata de una espera.

Esta modificación conserva además la función de todos los registros descritos en el Apartado 8.4, siempre que no estén descritos aquí.

Descripción de las señales de los algoritmos

Aquí se enumeran y describen de las señales tanto numéricas como de texto que emiten los algoritmos presentados en el proyecto. Las señales numéricas se presentan según el canal por el que se emiten ya que, en nuestros algoritmos, es el canal el que da sentido al número emitido. Los módulos de nuestro algoritmo disponen de 8 canales distintos de emisión/recepción de señales.

Señales de texto

Detac: Mensaje enviado durante la fase de búsqueda de la raíz. Este mensaje indica al módulo que la recibe que debe desconectarse del vecino que le ha enviado la señal.

Back_: Mensaje enviado durante la fase de búsqueda de la raíz. Este mensaje se envía una vez que la señal de un candidato a raíz ha alcanzado una hoja del árbol. Este mensaje se emite hacia la raíz para comprobar que el árbol generador inicial es correcto. Todos los módulos que emiten esta señal pasan a estado *BackS* o *BackF* dependiendo de si se trata de un módulo del árbol generador inicial o final respectivamente.

Slave: Mensaje enviado durante la fase de búsqueda de la raíz. Este mensaje se envía una vez que se ha encontrado la raíz del árbol e indica que todos los módulos deben pasar a estado *Slave* o *Cmprs* dependiendo de si forman parte del árbol generador final o inicial respectivamente.

LIDER: Mensaje enviado durante la fase de expansión. Todo módulo que recibe este mensaje pasa a estado *LIDER*.

EXPND: Mensaje enviado durante la fase de expansión. Este mensaje indica al módulo que el emisor de la señal pide permiso para enviarle un módulo comprimido.

CANEX: Mensaje enviado durante la fase de expansión. Este mensaje indica al módulo que el emisor de la señal da permiso para enviarle un módulo comprimido.

Disal: Mensaje enviado durante la fase de expansión. Este mensaje indica al módulo que lo recibe que debe, si le es posible, abandonar su rama para incorporarse a la rama del emisor de la señal.

NDISA: Mensaje enviado durante la fase de expansión. Este mensaje indica al módulo que lo recibe que la operación de cambio de rama que se ha pedido no va a realizarse. Este mensaje solo se envía en respuesta a la señal *Disal*.

EXPDL: Mensaje enviado durante la fase de expansión. Este mensaje indica al módulo que lo recibe que la operación de cambio de rama se ha realizado satisfactoriamente. Este mensaje solo se envía en respuesta a la señal *Disal*.

ASK_Z: Mensaje enviado durante la fase de compresión. Indica al módulo que lo recibe que el emisor de la señal desea comprimirse en su interior.

CAN_Z: Mensaje enviado durante la fase de compresión. Indica al módulo que lo recibe que el emisor de la señal le da permiso para se comprima en el emisor. Este mensaje solo se emite en respuesta a la señal*ASK_Z*.

ASKSZ: Mensaje enviado durante la fase de compresión. Este mensaje indica al módulo que lo recibe que el emisor de la señal le pide permiso para enviarle un módulo comprimido.

CANSZ: Mensaje enviado durante la fase de compresión. Este mensaje indica al módulo que lo recibe que el emisor de la señal le da permiso para

enviarle un módulo comprimido. Este mensaje solo se emite en respuesta a la señal*ASKSZ*.

NO_SZ: Mensaje enviado durante la fase de compresión. Este mensaje indica al módulo que lo recibe que la operación de paso de módulo comprimido ya confirmada, mediante una señal *CANSZ*, no va a poder realizarse. Este mensaje solo se emite en respuesta a la señal*CANSZ*.

Señales numéricas

01: Este canal se utiliza en dos fases: en la fase de búsqueda de la raíz (solo en el caso del algoritmo multilíder) y en la fase de expansión. En la fase de búsqueda de la raíz se utiliza este canal para que un hijo pueda informar a un padre del número de descendientes del padre. En la fase de expansión este canal se utiliza para notificar que la señal de líder viaja en dirección a la raíz, una vez completada la expansión de una rama y para pausar o reanudar la actividad de los módulos (este último caso solo tiene lugar en los algoritmos con señal de parada). Para notificar el paso de la señal de *líder* el valor que viaja por el canal es 1, para pausar o reanudar la actividad de los módulos el valor es 9999 y 9998 respectivamente.

02: Este canal se utiliza durante la fase de expansión. Su única utilidad es la pasar el valor 1 a un módulo que acaba de ser expandido en una posición vacía para indicarle que ya no está comprimido dentro de ningún módulo.

07: Este canal se utiliza durante la fase de expansión para informar a un módulo del número de módulo que se han añadido o retirado de su rama a consecuencia de una operación de cambio de rama. El valor puede ser cualquier número natural hasta 32767.

08: Este canal se utiliza durante la fase de compresión para informar a un módulo que se ha realizado una operación de cambio de módulo en su rama y que, por tanto, se han perdido o añadido módulos en esta. Su valor es 9999 en caso de perder módulos y a 9998 en caso de añadir módulos.