IMPLEMENTACIÓN DE ALGORITMOS DISTRIBUIDOS EN ERLANG PARA COMUNICACIONES EN REDES DE PROCESOS

XU HAN

MÁSTER EN INGENIERÍA INFORMÁTICA, FACULTAD DE INFORMÁTICA,
UNIVERSIDAD COMPLUTENSE DE MADRID

Trabajo de Fin de Máster en Ingeniería Informática

07 de Septiembre de 2015

Directores:
JAIME SÁNCHEZ HERNÁNDEZ
MANUEL MONTENEGRO MONTES

Autorización de Difusión

XU HAN

07 de Septiembre de 2015

La abajo firmante, matriculada en el Máster en Ingeniería Informática de la Facultad de Informática, autoriza a la Universidad Complutense de Madrid (UCM) a difundir y utilizar con fines académicos, no comerciales y mencionando expresamente a su autor el presente Trabajo Fin de Máster: "IMPLEMENTACIÓN DE ALGORITMOS DISTRIBUIDOS EN ERLANG PARA COMUNICACIONES EN REDES DE PROCESOS", realizado durante el curso académico 2014-2015 bajo la dirección de JAIME SÁNCHEZ HERNÁNDEZ y MANUEL MONTENEGRO MONTES en el Departamento de SISTEMAS INFORMÁTICOS Y COMPUTACIÓN, y a la Biblioteca de la UCM a depositarlo en el Archivo Institucional E-Prints Complutense con el objeto de incrementar la difusión, uso e impacto del trabajo en Internet y garantizar su preservación y acceso a largo plazo.

Resumen en castellano

El objetivo de este trabajo es implementar en Erlang algunos algoritmos distribuidos conocidos sobre grafos. Además implementamos un servidor genérico incorporando estos algoritmos.

En concreto, hemos elegido dos algoritmos distribuidos existentes para el cómputo del árbol de recubrimiento, y otro para el árbol de recubrimiento mínimo.

Después hemos desarrollado herramientas para la visualización de dichos árboles, y algoritmos de comunicación (`broadcast` y `convergecast`) entre los procesos del grafo, utilizando los árboles calculados para mejorar el rendimiento.

Hemos desarrollado además una extensión de un comportamiento (*behaviour*) de Erlang para encapsular estos algoritmos. Como resultado obtenemos una implementación útil para resolver problemas reales en este área.

Palabras clave

Algoritmo distribuido, broadcast, convergecast, Erlang, servidor genérico, árbol de recubrimiento, red de procesos

Resumen en inglés

Implementation of distributed algorithms in Erlang for process network communication.

The aim of this work is to implement in Erlang some well-known distributed algorithms on graphs. In addition, we implement a generic server by including these algorithms.

In particular, we have chosen two distributed algorithms for computing an spanning tree of a graph, and another one for computing its minimum spanning tree. Afterwards we have developed tools for displaying these trees, and two algorithms (convergecast and broadcast) for communicating the nodes of a graph, by using the previously computed trees, in order to improve performance.

Besides this, We have developed an extension of an Erlang behaviour in order to encapsulate these algorithms. As a result, we have obtained an useful implementation that addresses actual problems in this area.

Keywords

Distributed algorithm, broadcast, convergecast, Erlang, generic server, spanning tree, process network

Índice de contenidos

Índice de figuras

Índice de tablas

Capítulo 1 - Introducción

En este trabajo se profundiza en algunos aspectos de la computación distribuida, utilizando la concurrencia para permitir coordinación y sincronización individual de los nodos de una red de procesos.

En la asignatura de PDAP hemos obtenido un conocimiento básico del lenguaje Erlang y este trabajo da continuidad a este aprendizaje. Utilizamos este lenguaje para resolver problemas reales sobre redes de procesos y permitir la ejecución de las peticiones de los mismos de manera asíncrona y eficiente.

En concreto, abordamos la construcción de árboles de recubrimiento y árboles de recubrimiento mínimo[1]. A continuación hemos desarrollado algoritmos de comunicación entre los vértices del grafo, utilizando dichos árboles para mejorar el rendimiento. Finalmente se ha implementado una librería Erlang utilizando las tecnologías OTP para encapsular esta funcionalidad mediante el mecanismo estándar de este lenguaje[2]. El código de los algoritmos descritos en este memoria está disponible en:

http://gpd.sip.ucm.es/jaime/xu_han_TFM.rar

En este capítulo se presenta una breve introducción a la computación distribuida, el lenguaje Erlang y la librería OTP. A continuación se resumen los objetivos de este trabajo y su planificación.

1.1 Algoritmos distribuidos

Un *sistema distribuido* consiste en una serie de nodos computacionales conectados en una red de comunicaciones que cooperan para llevar a cabo una tarea común[3][4]. En esta red los nodos trabajan de manera autónoma[5][6]. Dos nodos cualesquiera de la red pueden estar comunicados directa o indirectamente (mediante otros nodos intermedios). Las ventajas más destacables de este tipo de sistemas son la posibilidad de compartir recursos y la tolerancia a fallos (si un nodo se cae los demás pueden asumir su tarea). La computación distribuida es un modelo para resolver problemas de computación masiva utilizando un gran número de ordenadores organizados en *clusters* incrustados en una infraestructura de telecomunicaciones distribuida[7]. Es decir, cuando algunos problemas o proyectos necesitan resolver un

problema computacionalmente complejo, la computación distribuida divide el problema o proyecto en partes más pequeñas (que a su vez se pueden dividir sucesivamente) y las asigna a distintos nodos para su procesamiento. De este modo se ahorra tiempo de cálculo, al poder realizar los distintos nodos operaciones de manera paralela. La computación distribuida ha experimentado un desarrollo sin precedentes en las últimas décadas, sobre todo en la computación en Grid, computacion en la nube, redes *ad hoc* móviles y redes de sensores inalámbricos[3].

Un *algoritmo distribuido* es un algoritmo diseñado para ejecutarse en los distintos nodos de un sistema distribuido, mediante la cooperación y sincronización de dichos nodos, con el fin de lograr un objetivo común[8]. Los algoritmos distribuidos se están convirtiendo en una rama de investigación importante en los campos tales como ciencias de la computación, ingeniería, matemática aplicada y en muchos contextos donde la distribución resulta la forma natural de abordar un problema, como algoritmos *divide y vencerás*, servidores, etc[3].

Los algoritmos distribuidos[9] para el cálculo de árboles de recubrimiento que se van a implementar en este trabajo han sido tomados del libro *Distributed Graph Algorithms for Computer Networks* escrito por *K. Erciyes*[3]. Con respecto a los algoritmos de propagación de mensajes a través del grafo, si bien son clásicos en programación, su implementación concreta en Erlang es original de este trabajo.

1.2 Introducción al lenguaje Erlang

Erlang es un lenguaje de programación funcional orientado a sistemas concurrentes[10]. Originalmente fue desarrollado por la compañía sueca Ericsson. El propósito de este lenguaje era hacer frente a los problemas de escalabilidad que afectaban a las implementaciones de sus sistemas concurrentes[11]. Este lenguaje da soporte a sistemas concurrentes y distribuidos que se comunican mediante paso de mensajes de manera asíncrona. Soporta características típicas de los lenguajes funcionales como el orden superior, funciones anónimas, listas intensionales, etc., pero admite algunas características imperativas (no es puro desde el punto de vista funcional) y no tiene disciplina estática de tipos. En Erlang no existe memoria compartida, sino que cada proceso tiene su propio almacenamiento privado.

En este trabajo se asume un conocimiento básico del lenguaje Erlang. No obstante, a continuación se presentan algunos aspectos del lenguaje a los que se hará referencia en capítulos posteriores.

En Erlang cada proceso se identifica unívocamente dentro de una red de nodos mediante un PID (*Process IDentifier*), que tiene la forma `<X.Y.Z>`, donde `X`, `Y` y `Z` son números. Cuando un proceso quiere enviar un mensaje a otro proceso lo puede hacer mediante el operador `!` (*send*), que recibe el PID del proceso destino y el mensaje a enviar (un término cualquiera de Erlang). Dentro de este término es habitual incluir el PID del remitente, para que el destinatario pueda responder al mismo.

La distribución estándar de las herramientas del lenguaje Erlang incluye las librerías OTP. Estas librerías a su vez incluyen, además de funciones varias sobre manejo de listas, ficheros, etc., módulos que abstraen los aspectos comunes de las implementaciones de arquitecturas distribuidas tales como cliente/servidor. Estas abstracciones se materializan como comportamientos (*behaviour*) OTP, que conceptualmente son como clases abstractas en un lenguaje de programación orientado a objetos. Especifican un conjunto de funciones *callback* que el programador tiene que implementar con la funcionalidad concreta de su aplicación, mientras que el "esqueleto" de la aplicación viene dado por las librerías OTP. Los comportamientos más utilizados sirven para implementar servidores, máquinas de estados finitos, gestores de eventos, etc.

1.3 Objetivos

El objetivo original del trabajo es estudiar algoritmos ya existentes para el cálculo distribuido del árbol de recubrimiento mínimo de un grafo e implementarlos en Erlang[12]. En concreto se han abordado los siguientes aspectos:

- Algoritmos distribuidos para calcular el árbol de recubrimiento (no necesariamente mínimo) de un grafo: `term_ST` y `tarry_ST`. Estos dos algoritmos se han tomado del libro de K.Erciyes[3].
- Algoritmo distribuido para calcular el árbol de recubrimiento mínimo de un grafo: `AGHS`, también tomado del libro de K.Erciyes[3].

- Visualización gráfica de los árboles de recubrimiento generados. Esta presentación ha sido muy útil para depurar los algoritmos y es además una contribución relevante desde el punto de vista didáctico.
- Implementación de los algoritmos `broadcast` y `convergecast` para realizar cómputos distribuidos a través de una red utilizando el árbol de recubrimiento generado.
- Comparativa de rendimiento entre los algoritmos de `term_ST` y `tarry_ST`.
- Abstracción de los algoritmos `term_ST` y `AGHS` en un comportamiento de Erlang, de modo que el usuario (programador) de este comportamiento puede abstraerse de los detalles de construcción de la red y cálculo del árbol de recubrimiento mínimo y puede únicamente centrarse en las funciones `broadcast` y `convergecast`.

1.4 Plan de trabajo

El resto del trabajo está dividido en los siguientes capítulos:

- Capítulo 2

Se presentará una descripción de los algoritmos distribuidos `term_ST` y `tarry_ST`, y su implementación en Erlang con las explicaciones de cada módulo y mensaje utilizados. Al final del capítulo se incluye un ejemplo para ilustrar el funcionamiento y se hace una comparativa de rendimiento entre los dos.

- Capítulo 3

Se presentará el algoritmo distribuido `AGHS` y su implementación en Erlang con las explicaciones de cada módulo y mensaje utilizados[13]. También al final del capítulo se muestra un ejemplo de cómputo.

- Capítulo 4

Se presentarán los algoritmos `broadcast` y `convergecast`, algunos ejemplos para ilustrar su funcionamiento y su implementación en Erlang con las explicaciones de cada módulo y mensaje utilizados.

- Capítulo 5

Se implementa el comportamiento OTP de Erlang que encapsula algunos de estos algoritmos y un servidor genérico.

- Capítulo 6

  Se presentarán las conclusiones y el trabajo futuro.

Chapter 1 - Introduction

This paper elaborates on some aspects of distributed computing by using concurrence to allow individual coordination and synchronization of the nodes of a network of processes.

In the PDAP course (*Programación Declarativa Aplicada*) we have obtained a basic knowledge of the Erlang language and this work is a continuation of this learning. We use this language to solve real problems on process networks and allow the execution of their requests asynchronously and efficiently.

Specifically, we address the construction of the spanning tree and the minimum spanning tree[1] of a graph. We have developed algorithms for communication between the vertices of the graph by using these trees to improve performance. Finally, we have implemented an Erlang library by using the OTP technology in order to encapsulate this functionality through the standard mechanism of this language[2]. The code of the algorithms described in this work is available at:

http://gpd.sip.ucm.es/jaime/xu_han_TFM.rar

This chapter gives a brief introduction to distributed computing, the Erlang language, and the OTP library. Afterwards, we summarize the objectives of this work and its work plan.

1.1 Distributed algorithm

A *distributed system* consists of a number of computational nodes connected by a communication network that cooperates to accomplish a common task[3][4]. In this process network each process works in an independent way[5][6]. Any two nodes of the network can be communicated directly or indirectly (i.e. through other intermediate nodes). The main advantages of this kind of systems are the ability to share resources and their fault tolerance (if one node fails, the remaining ones can assume its task). Distributed computing is a model meant to solve problems involving massive computations by using a large number of computers organized into *clusters* embedded in an infrastructure of distributed telecommunications[7]. That is, when some problems or projects need to solve a computationally complex problem, a distributed computing-based approach divides the problem or project into smaller parts (which, in turn, can be subsequently divided) and

assigns them to different nodes for processing. Therefore, this approach allows better computation times, since the nodes can perform their operations at the same time. Distributed computing has experienced an unprecedented development in recent decades, especially in the areas of grid computing, cloud computing, mobile *ad hoc* networks and wireless sensor networks[3].

A distributed algorithm is an algorithm meant to be run in different nodes of a distributed system by means of cooperation and synchronization between them, in order to achieve a common goal[8]. Distributed algorithms are also becoming an important research field in some fields such as computing science, engineering, applied mathematics and in many contexts in which distribution is the natural way to address a problem, such as divide-and-conquer algorithms, servers, etc[3].

The distributed algorithms[9] for the spanning tree computation that will be implemented in this work have been taken from the book *Distributed Graph Algorithms for Computer Networks* written by *K. Erciyes*[3]. With regard to the algorithms of message propagation through the graph, although they are conventional in programming, their concrete implementation in Erlang is a novel part of this work.

1.2 Introduction to Erlang language

Erlang is a functional programming language oriented to the development of concurrent systems[10]. It was developed originally by the Swedish company Ericsson. The purpose of this language was to address the scalability problems which were arising in the implementations of their concurrent systems[11]. This language supports concurrent and distributed systems that communicate with each other by asynchronous message passing. It provides the usual properties of functional languages such as higher-order functions, anonymous functions, list comprehensions, etc., but allows some imperative features (that is, it is not pure from the functional point of view) and has no static type discipline. In Erlang there is no shared memory, but each process has its own private storage.

In this work a basic knowledge of Erlang is assumed. However, we introduce some aspects of the language which will be mentioned in later chapters.

In Erlang each process is uniquely identified within a process network by a PID (*Process IDentifier*), which has the form `<X.Y.Z>`, where `X`, `Y` and `Z` are numbers. When a process wants to send a message to another one, it can do it by applying the operator `!` (*send*), which receives the PID of a given process and the message being sent (any Erlang term). Within this message it is common to include the PID of the sender in order to allow an answer from the recipient.

The standard distribution tools of the Erlang language includes the OTP libraries. These libraries include, in addition to several functions for handling lists, files, etc., modules that abstract the common aspects of the implementations of distributed architectures such as client/server. These abstractions are materialized as OTP *behaviours*, that are like abstract classes in a object-oriented programming language. Behaviours specify a set of callback functions that the programmer has to implement with the concrete functionality of her application, while the "skeleton" of the application is already given by the OTP libraries. Behaviours are commonly used to implement servers, finite state machines, event handlers, etc.

1.3 Objectives

The original aim of this work is to study existing algorithms for the distributed computation of a minimum spanning tree within a graph and to implement them in Erlang[12]. In particular, we have addressed the following goals:

- Two distributed algorithms for computing the (not necessarily minimum) spanning tree of a graph: `term_ST` and `tarry_ST`. These algorithms have been taken from the book of K. Erciyes[3].
- A distributed algorithm for computing the minimum spanning tree of a graph: AGHS, which has been taken from the book of K. Erciyes[3].
- Graphical display of the generated spanning trees. This presentation has been very useful for debugging algorithms and is also a relevant contribution from the educational point of view.
- Implementation of the `broadcast` and `convergecast` algorithms in order to perform distributed computing over a network by using the generated spanning tree.

- Comparison of performance between the algorithms `term_ST` and `tarry_ST`.
- Abstraction of the algorithms `term_ST` and `AGHS` in an Erlang behaviour, so that the user (programmer) of this behaviour can be abstracted from the details of construction of the network and the computation of the minimum spanning tree, so she can only focus on the `broadcast` and `convergecast` operations.

1.4 Work plan

The rest of this work is divided into the following chapters:

- Chapter 2

It introduces a description of the distributed algorithms `term_ST` and `tarry_ST`, and its implementation in Erlang with explanations of each module and the messages they use. The end of this chapter includes an example illustrating their operation, and a performance comparison between them.

- Chapter 3

In this chapter the AGHS distributed algorithm and its implementation in Erlang will be presented, with the explanations of each module and message involved[13]. An example of computation is also given at the end of the chapter.

- Chapter 4

In this chapter the algorithms `broadcast` and `convergecast` will be introduced, with some examples illustrating their operation and their implementation in Erlang with explanations of each module and message.

- Chapter 5

In this chapter we have implemented the Erlang OTP behaviour that encapsulates some of these algorithms into a generic server.

- Chapter 6

It introduces the conclusions and describes future work.

Capítulo 2 - Algoritmos para el cálculo del árbol de recubrimiento para grafos no dirigidos

En este capítulo se introducen dos algoritmos para generar el árbol de recubrimiento de un grafo dado. A partir del grafo de entrada, se generan unos procesos, donde cada proceso representa un nodo del grafo y el cálculo del árbol de recubrimiento se lleva a cabo mediante el envío de mensajes entre los distintos nodos. Los dos algoritmos que se presentarán son el algoritmo distribuido asíncrono con detección de terminación y el algoritmo de recorrido de Tarry[3].

Comenzamos introduciendo algunas nociones básicas sobre grafos. Una *componente conexa* de un grafo G(V, E) es un subgrafo G' de G donde existe un camino entre cualquier par de vértices en G'. Si, dentro de un grafo, existe un camino entre dos vértices cualesquiera a través de una o varias aristas, decimos que el grafo es *conexo* y sólo tiene una componente conexa. En el caso contrario, si existen dos vértices que no se pueden conectar mediante un camino, decimos que estos dos vértices están en dos componentes diferentes de un grafo. Un grafo conexo G solamente tiene una componente que es sí mismo[3]. Si un grafo G = (V, E) es acíclico y tiene más de una componente, G es un *bosque*[3]. Dado un grafo G(V, E), donde V es el conjunto de vértices y E es el conjunto de aristas, se define un *bosque de recubrimiento* como un subgrafo acíclico H(V', E') del grafo G donde V' = V. Si H tiene una componente conexa, entonces H(V', E') es un *árbol de recubrimiento* de G[3].

Los algoritmos de cálculo del árbol de recubrimiento han sido ampliamente utilizados en contextos industriales, por ejemplo en el diseño de redes. Además tienen aplicaciones importantes en diseño de redes de computación y comunicaciones, enlaces de redes de transporte, conexiones de cableado, etc. También tienen aplicaciones menos directas tales como pruebas de homogeneidad superficial, problemas de clustering y clasificación, etc.[14].

El primer algoritmo que se va a presentar es un algoritmo distribuido asíncrono con detección de terminación que llamamos `term_ST`. En un algoritmo distribuido asíncrono no existe ninguna restricción sobre el orden de ejecución de los procesos, ni sobre los intervalos temporales en los que cada proceso puede enviar mensajes. Esto

contrasta con los algoritmos síncronos, en el que los procesos se comunican a lo largo de una serie de ciclos; en cada ciclo reciben los mensajes enviados en el ciclo anterior, los procesan, y envían los mensajes necesarios para el siguiente ciclo. En comparación con los algoritmos síncronos, los algoritmos asíncronos pueden reducir el tiempo de respuesta del sistema. Sin embargo, los algoritmos asíncronos son más difíciles de analizar y comprobar que son correctos. En un algoritmo asíncrono, los nodos envían distintos tipos de mensajes, según el destinatario sea su hijo o su padre dentro del árbol de recubrimiento que se está construyendo. A su vez, un hijo envía un mensaje a su padre una vez que todos los sus hijos le han contestado. De este modo, cualquier nodo puede detectar la terminación del algoritmo para el subárbol del que es raíz. Se detalla este algoritmo en la sección 2.1 de este capítulo.

El segundo algoritmo presentado en este capítulo es el de recorrido de Tarry. En este caso se realiza un recorrido de los nodos del grafo mediante una ficha (*token*). Esta ficha es como un símbolo que indica qué nodos ya están visitados. Los nodos almacenan el estado de sus vecinos para identificar si ya les ha pasado la ficha previamente. Cuando un nodo recibe la ficha por primera vez, considera que el emisor del nodo es su padre en el árbol de recubrimiento, la envía a uno cualquiera de sus restantes vecinos, y espera a que le sea devuelta por éste. Después, se la envía a otro vecino que no la haya recibido anteriormente. Un nodo devuelve la ficha a su padre solo en el caso de que ya haya enviado la ficha a todos los restantes vecinos (hijos en el árbol de recubrimiento). Se detalla este algoritmo en la sección 2.2 de este capítulo.

2.1 Algoritmo asíncrono con detección de la terminación (`Term_ST`)

Este algoritmo utiliza una máquina de estados finitos, un modelo matemático utilizado en sistemas cuya salida depende de su entrada y su estado actual (el cual, a su vez, depende de la historia de su entrada). La máquina dispone de un conjunto de estados, de los cuales uno de ellos es el estado inicial. Cuando la máquina recibe una cadena de símbolos (entrada), se cambia su estado dependiendo de la función de transición, que determina la salida y el estado siguiente en función de la entrada y el estado actual. La diferencia entre una máquina de estados y un sistema puramente funcional es que la

salida del sistema funcional sólo depende de la entrada, mientras que en una máquina de estados depende también del estado actual. También se puede describir gráficamente la máquina de estados como un grafo dirigido, en el que cada nodo es un estado, y las aristas representan la función de transición. La denominación 'máquina de estados finitos' significa que existe un número limitado de estados en la máquina. Pero, en un instante dado sólo tiene un estado actual.

Existen dos tipos de máquina de estados: Moore y Mealy. En ambos casos el estado actual y la entrada deciden el estado siguiente. La diferencia es el modo en que la entrada afecta a la salida. En una máquina de estados de Moore, la entrada cambia el estado actual, y no produce ninguna salida. Es decir, en este tipo de máquinas de estados la entrada solo provoca que la máquina entre en el siguiente estado y no especifica el valor de salida. Los valores de salida dependen exclusivamente del estado actual. Sin embargo, en una máquina de estado Mealy, la entrada no solo cambia el estado actual, sino que también su valor especifica la salida.

En el algoritmo que presentamos, cada nodo es una máquina de estados finitos que tiene tres estados: `idle`, `xplord` y `term`. Todos los nodos empiezan con el estado `idle` y terminan con el estado `term`. A continuación describimos estos estados:

- `idle`

  El estado inicial. Existen dos posibles situaciones para cambiar desde este estado al estado `xplord`: o bien la raíz recibe el mensaje `root`, o bien un nodo que no es la raíz recibe el mensaje `probe` por la primera vez.

- `xplord`

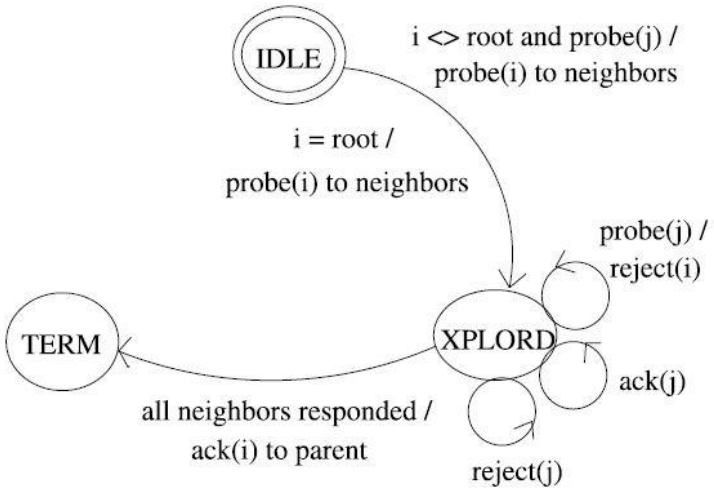
  En este estado el nodo puede enviar mensaje `probe` o `reject` a sus vecinos, o enviar mensaje `ack` a su padre. Cuando un nodo entra en este estado es debido a la recepción, por primera vez, del mensaje `probe` desde otro nodo. En este caso, el nodo receptor identifica al remitente como su padre dentro del árbol de recubrimiento y reenvía el mensaje `probe` al resto de vecinos. Si un nodo, estando ya en el estado `xplord`, recibe otro mensaje `probe`, le responderá al remitente de éste último con un mensaje `reject`, indicando que ya tiene un padre. Por último, cuando un nodo en estado `xplord` ha recibido las respuestas `ack` o `reject` de todos los vecinos a los

que les ha enviado el mensaje `probe`, pasa al estado `term`. Por tanto, sólo hay una situación para cambiar desde este estado al estado `term`: el nodo no tiene ningún vecino más que no haya enviado respuesta.

- `term`

Es el estado final. Si el nodo que entra en este estado es la raíz del árbol de recubrimiento, termina este algoritmo. Si no, envía mensaje `ack` a su padre.

Figura 2.1 Diagrama de máquina de estado del algoritmo `term_ST`[3]



En la figura 2.1 se muestra la máquina de estado del algoritmo `term_ST`. El resumen puede ver en la tabla 2.1.

Tabla 2.1 Resumen de máquina de estado

ESTADO	ENTRADA	SALIDA	ESTADO OBJETIVO
idle	es root	envía mensaje probe a todos sus vecinos	xplord
	no es root y recibe mensaje probe desde otro nodo	envía mensaje probe a todos sus vecinos	xplord
xplord	recibe mensaje probe desde otro nodo	envía mensaje reject a eso nodo	xplord
	recibe mensaje ack desde otro nodo	-	xplord

	recibe mensaje reject desde otro nodo	-	xplord
	recibe respuesta de todos sus vecinos	envía mensaje ack a su padre	term
term	-	-	term

2.2 El algoritmo de recorrido de Tarry (`tarry_ST`)

El algoritmo de recorrido de Tarry construye también un árbol de recubrimiento, pero mediante el uso de una ficha (*token*). Este algoritmo empieza a transmitir la ficha desde la raíz y al final se propaga por todos los nodos del grafo. Cuando un nodo recibe la ficha por primera vez, pone el remitente de ésta como su padre y pasa la ficha a alguno de sus vecinos que no hayan sido visitados (excepto a su padre). En otro caso, cuando el nodo ha recibido ya la ficha previamente, comprueba si tiene otros vecinos no visitados: si no los tiene, devuelve esta ficha a su padre; si los tiene le pasa esta ficha a alguno de ellos. Hay que tener la cuenta que durante todo el recorrido sólo existe una única ficha que se transmite desde un nodo a otro. La ficha no se puede pasar dos veces por el mismo canal en la misma dirección, es decir, si la ficha ya ha pasado desde nodo A a nodo B, no se puede pasar desde el nodo A al nodo B de nuevo. Pero la ficha sí podría pasar desde el nodo B al nodo A, por ejemplo cuando el nodo B recibe la ficha de sus hijos.

Para elegir un vecino para pasar la ficha, el nodo tiene que monitorizar los estados de sus vecinos. Cuando pasa la ficha a uno de sus vecinos, marca el estado de este vecino como visitado. Una vez que todos los vecinos han sido visitados, devuelve la ficha a su padre para que éste la pase a su siguiente vecino no visitado. Cuando la ficha regresa al nodo raíz (que es el que recibe y propaga la ficha por primera vez), termina este algoritmo.

Además para ambos algoritmos en el árbol de recubrimiento el nodo no sólo conoce a sus hijos sino también a su padre.

2.3 Implementación en Erlang

Una vez presentada la descripción teórica de los dos algoritmos, ahora se explica cómo se implementan en Erlang. Probaremos ambos algoritmos sobre un grafo generado

aleatoriamente, y analizaremos su eficiencia experimentalmente, comparando los tiempos de cómputo.

Para implementar estos algoritmos en Erlang, desarrollamos un módulo `connectedGraph` para generar un grafo de modo aleatorio. Cada algoritmo se implementa en su propio módulo, que utiliza el grafo generado por `connectedGraph` y calcula el árbol de recubrimiento según se ha descrito en las secciones anteriores. En las siguientes subsecciones se detalla la implementación de estos módulos.

2.3.1. Módulo `connectedGraph`

El objetivo de este módulo es generar un grafo aleatorio no dirigido (valorado) a partir de parámetros: número de vértices y número de aristas, `NumNodes` y `NumEdges`, respectivamente.

Cuando se introducen estos dos parámetros, hay que comprobar que son válidos. Además, en un grafo conexo, hay que comprobar el número de aristas está entre `NumNodes -1` y $\lfloor$`(NumNodes*(NumNodes-1))/2`$\rfloor$.

Por otro lado, como este grafo se genera aleatoriamente, hay que garantizar que sólo tiene una componente conexa. Es decir, que entre dos nodos cualesquiera que estén contenidos en este grafo existe un camino que los comunica mutuamente. Si el grafo tiene varias componentes conexas, se pueden obtener varios árboles de recubrimiento, uno por cada componente. Al utilizar algoritmo `term_ST` hay que elegir un nodo como `root` para cada componente, de modo que en este grafo tendrámos varias raíces. En esta misma situación, cuando se utilice algoritmo `tarry_ST` hay que crear una ficha para cada componente.

Para hacer estas comprobaciones, en este módulo utilizamos el módulo `digraph` de la librería estándar de Erlang. Con la ayuda de estos módulos es mucho más sencillo comprobar la correcta construcción del grafo generado aleatoriamente.

El módulo `digraph` implementa las funciones básicas de un grafo dirigido tales como crear un grafo nuevo, añadir vértices y arcos a un grafo ya existente, eliminar vértices y arcos de un grafo, consultar la cantidad de vértices y arcos, obtener la ruta entre dos vértices dados, etc. Un grafo dirigido es un par (V, E) donde V es un conjunto finito de vértices y E es un conjunto finito de aristas dirigidas. En un grafo dirigido, las aristas

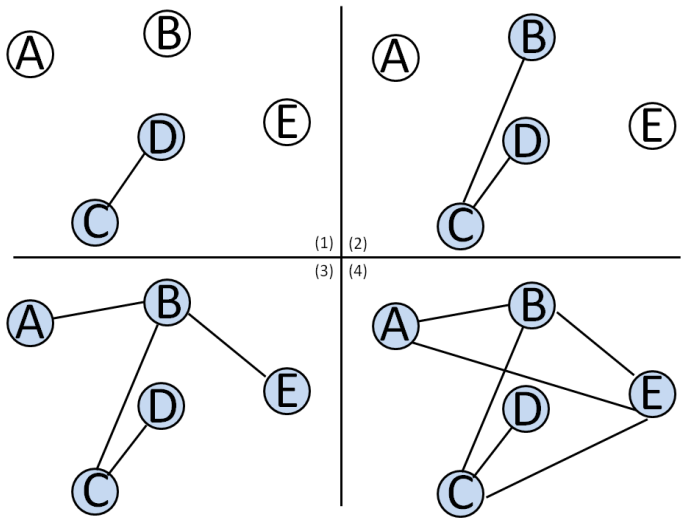
tienen un sentido. Por ejemplo, la arista desde el vértice A hasta el vértice B es diferente de la arista desde el vértice B hasta el vértice A. Como los dos algoritmos expuestos en este capítulo se basan en grafos no dirigidos, hay que tener esto en cuenta a la hora de crear las aristas.

En nuestro caso se genera un grafo vacío usando la función `new/0` del módulo `digraph`. Luego se añaden `NumNodes` nodos. Se seleccionan arbitrariamente dos nodos y se combinan para añadir una arista entre ellos. Como el grafo generado por el módulo `digraph` es dirigido, mientras que el grafo que queremos generar no lo es, cuando se añade una arista nueva hay que tener en cuenta que no se añada por duplicado. Por ejemplo, si se añade un arco entre A y B, se tiene que comprobar si este arco (bien desde A a B, o bien desde B a A) ya existe en la lista de las aristas que se han añadido en el grafo.

En el segundo paso usamos algunas funciones del módulo `digraph_utils` para optimizar el grafo que se ha generado en el paso anterior. En el módulo `digraph_utils` existen varias funciones implementadas que se basan en el recorrido en profundidad de un grafo dirigido. Por ejemplo, comprueban si el grafo es un árbol, implementan algoritmos de ordenamiento topológico, buscan componentes fuertemente conexas, etc. Al final de este algoritmo se devuelve la lista de aristas en forma de pares {X, Y}, donde X e Y son números que identifican cada nodo.

En nuestro caso, como las aristas se añaden aleatoriamente, hay que usar un algoritmo para garantizar que el grafo generado es conexo. El algoritmo es el siguiente: en primer lugar se seleccionan dos nodos cualesquiera para añadir la primera arista entre ellos, y luego se elige uno de estos dos nodos y se conecta con uno de los restantes, etc. De este modo, se van conectando vértices seleccionados previamente con vértices aún sin seleccionar. Tras `NumNodes-1` iteraciones se obtendrá un grafo conexo construido con un número mínimo de aristas. A partir de este momento, se pueden añadir las aristas restantes de manera aleatoria en el caso en que el usuario requiera más aristas en el grafo.

Figura 2.2 Ejemplo de construcción del grafo



Por ejemplo en la figura 2.2, supongamos que el cliente quiere construir un grafo conexo con cinco nodos y 6 aristas. Al principio tenemos 5 vértices individuales que son A, B, C, D y E. Se añade la primera arista aleatoriamente (por ejemplo, entre C y D). El siguiente paso es elegir uno de estos dos nodos y otro del conjunto {A, B, E}. En nuestro ejemplo elegimos C y B para añadir una arista entre ellos. Las siguientes dos iteraciones añaden las arista {A, B} y {B, E}. Hasta ahora el grafo conexo ya está construido pero aún faltan dos aristas más que ha pedido el usuario. Entonces se seleccionan aleatoriamente dos nodos cualesquiera del grafo para cada arista. En este caso tenemos una arista entre A y E, y otra arista entre C y E.

El módulo `connectedGraph` se utiliza en todos los algoritmos descritos en este trabajo. Antes de ejecutar los dos algoritmos, es necesario transformar el grafo obtenido aleatoriamente por este módulo en una red de procesos interconectados. Es decir, cada nodo será un proceso que contiene una lista de procesos vecinos. A continuación se describe el módulo `process_generator` que se encarga de generar los procesos y añadir las conexiones entre ellos según las aristas del grafo de entrada.

2.3.2. Módulo `process_generator`

Este módulo implementa como media entre el usuario y el algoritmo. El usuario solo debe introducir el número de nodos y arcos del grafo que quiere generar y se obtiene el árbol de recubrimiento para el grafo generado.

Este módulo llama a la función `graphstart/2` del módulo `connectedGraph` para crear un grafo conexo y a la función `start/2` del módulo `term_ST` (o `tarry_ST`) para generar la red de procesos. En primer lugar se genera un proceso por cada nodo y después se envía un mensaje a cada uno de ellos para que añada sus vecinos según la lista de los aristas del grafo. Se implementa la red de los procesos utilizando el mensaje `addEdge`. Este mensaje se usa para que el nodo añada los vecinos. Cuando un nodo lo recibe, añade el remitente a la lista de sus vecinos, y actualiza su estado.

Luego se selecciona un nodo (proceso) como `root`, por ejemplo el primero de la lista de los procesos, y se le envía un mensaje para arrancar el algoritmo. Cuando recibe el mensaje `terminate` desde `root`, el algoritmo termina y `process_generator` genera una salida para visualizar gráficamente el grafo y el árbol de recubrimiento obtenidos. Además se muestra el tiempo de ejecución del algoritmo para comparar con otros.

Tras la ejecución de los algoritmos de cálculo del árbol de recubrimiento, se obtiene la presentación gráfica del mismo en formato `.dot`. El usuario puede visualizar la imagen mediante *Graphviz* que es una potente herramienta para generar los diagramas o grafos en general.

En la siguiente sección se presentan los módulos correspondientes a los dos algoritmos que tratamos.

2.3.3. Módulo `term_ST`

En este módulo se implementa la parte lógica del proceso de este algoritmo. Como se ha explicado antes, se utiliza una máquina de estados asociados a cada nodo. El estado de un nodo cambia a través de mensajes. El estado de cada nodo contiene el *PID* de sí mismo, la lista de *PID*s de los procesos vecinos, el padre, la lista de hijos, la lista de los vecinos que no son sus hijos ni su padre, el estado actual (que puede ser `idle`, `xplord` o `term`), la lista de aristas del árbol, un recorrido jerarquizado por niveles del subárbol asociado (ajeno a la lógica del algoritmo, pero que facilita la salida para la representación gráfica), la lista de los hijos que se han encontrado y valor que tiene. Los mensajes son los siguientes:

- **Mensaje `root(Who)`**

Este mensaje se usa para decidir el nodo raíz. Antes de empezar el algoritmo se selecciona un nodo como raíz y se envía este mensaje a dicho nodo. Cuando un nodo recibe este mensaje, envía el mensaje `probe` a todos sus vecinos. Como un nodo raíz no tiene padre, marca a su padre como "no" dentro del estado del nodo. Su estado actual se cambia como `xplord`.

- **Mensaje `probe`(Who)**

Este mensaje sirve para preguntar al destinatario si el remitente puede ser su padre. Si puede serlo, este nodo marca al remitente como padre; en el caso contrario, envía mensaje `reject` para rechazar la petición.

Cuando un nodo recibe este mensaje, su estado actual puede ser `idle` o `xplord`. Si el nodo aún no ha recibido un mensaje `probe`, es decir que su estado actual es `idle`, no tiene padre. En este caso tiene que marcar al remitente como su padre y cambiar su estado actual como `xplord`. Como este nodo necesita seguir buscando sus hijos, envía el mensaje `probe` a todos sus vecinos (excepto a su padre) para solicitar ser su padre. En el caso de que este nodo no tenga más vecinos, aparte de su padre, envía un mensaje `ack` a este último para informar de que ya ha terminado su subárbol y le envía además la lista de aristas de su subárbol.

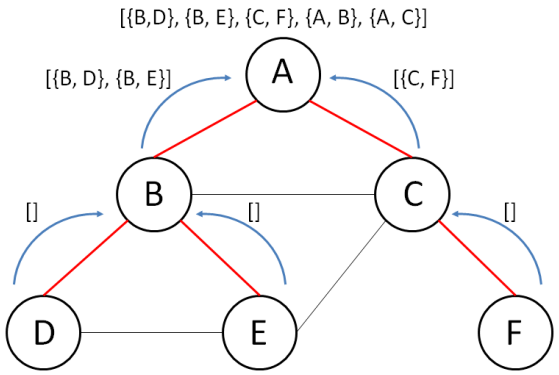
Si un nodo ya ha recibido este mensaje antes, es decir que su estado actual es `xplord`, ya tiene padre. En este caso envía mensaje `reject` al remitente para rechazar la petición.

- **Mensaje `ack`(Who, recorrido en anchura del hijo, lista de aristas de árbol del hijo)**

Una vez un nodo recibe este mensaje, anota el remitente como uno de sus hijos. Luego verifica que todos sus vecinos (excepto el padre) ya están clasificados como hijos u otros (`others`). La clasificación `Others` es para aquellos nodos que han rechazado la petición de paternidad del nodo actual (`probe`). Por ejemplo, si un nodo A envía un mensaje `ack` a un nodo B para preguntar si A puede ser el padre de B, en el caso en que B rechace la petición a nodo A decimos que el nodo B se clasifica como `other` para el

nodo A. Si todavía queda algún vecino que no ha enviado mensaje `ack` o el mensaje `reject`, el proceso de este nodo no ha terminado. En este caso sigue esperando la respuesta de los nodos que quedan por responder. Cuando no queda ningún vecino que sea hijo u `other`, el proceso de este nodo termina y envía mensaje `ack` a su padre. En este caso, si este nodo no tiene padre (`root`), termina el algoritmo. Este mensaje también lleva el subárbol correspondiente al remitente para que su padre lo incorpore al árbol que está construyendo. Cuando todos sus vecinos (excepto el padre) ya han enviado las respuestas, añade las aristas que conectan los hijos con este nodo para construir el subárbol. En la figura 2.3 se ilustra este procedimiento con un ejemplo.

Figura 2.3 Ejemplo de construir árbol de recubrimiento en gráfica



Por ejemplo, dados seis nodos que son: [A, B, C, D, E, F]; B es padre de E y D, C es el padre de F y A es el padre de B y C. Como D, E y F son hojas, éstas envían el mensaje `ack` a su padre junto con una lista vacía para decirle que sus subárboles correspondientes son vacíos. Cuando B recibe la mensaje `ack` desde E y D, fusiona las aristas recibidas por sus subárboles y envía las aristas que parten de él (esto es, las tuplas con su identificador y el de cada uno de sus hijos) en forma de lista.

- **Mensaje `reject(Who)`**

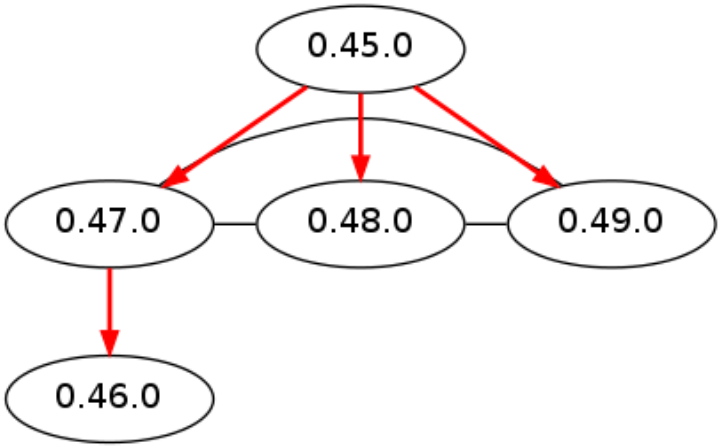
Cuando un nodo recibe este mensaje, añade el remitente a la lista de `others`. Luego verifica que todos sus vecinos (excepto su padre) ya están clasificados como hijo u `others`. Si todavía hay algún vecino sin clasificar, no se envía respuesta, ya que el

proceso de este nodo no ha terminado y sigue esperando hasta que todos los vecinos envíen las respuestas. En caso contrario (todos los vecinos ya han enviado la respuestas a este nodo), el nodo termina y envía mensaje `ack` a su padre. Una vez que todos los vecinos de la raíz han devuelto las respuestas, termina el algoritmo.

En la figura 2.4 se muestra un ejemplo para ilustrar el funcionamiento del algoritmo `term_ST`. Suponemos que se han generado 5 procesos y el grafo aleatorio conexo ya está calculado siendo los nodos:

[0.45.0, 0.46.0, 0.47.0, 0.48.0, 0.49.0], y los arcos [{0.45.0, 0.47.0}, {0.45.0, 0.48.0}, {0.45.0, 0.49.0}, {0.47.0, 0.48.0}, {0.47.0, 0.46.0}, {0.47.0, 0.49.0}, {0.48.0, 0.49.0}].

Figura 2.4 Ejemplo del algoritmo `term_ST`



En el ejemplo, `root` es el proceso 0.45.0. Envía el mensaje `probe` a los procesos 0.47.0, 0.48.0 y 0.49.0. El proceso 0.47.0 recibe el mensaje `probe` y envía este mensaje a los procesos 0.48.0, 0.49.0 y 0.46.0. Como los procesos 0.48.0 y 0.49.0 ya tienen padre (0.45.0) entonces rechazan al proceso 0.47.0 que sea su padre. El proceso 0.47.0 recibe el mensaje `reject` y marca ambos procesos como `others`. Del mismo modo, el proceso 0.48.0 es rechazado como padre por los procesos 0.47.0 y 0.49.0, y el proceso 0.49.0 es rechazado por los procesos 0.47.0 y 0.48.0. Como los procesos 0.48.0 y 0.49.0 no tienen más vecinos de los que esperar respuesta, entonces envían el mensaje `ack` al proceso 0.45.0 El proceso 0.45.0 recibe los mensajes `ack` desde los dos procesos 0.48.0 y 0.49.0, marca los dos como hijos y sigue esperando la respuesta del otro vecino 0.47.0. El proceso 0.46.0 recibe el mensaje `probe` y, como no tiene más vecinos excepto

el proceso 0.47.0, envía la respuesta (mensaje `ack`) al proceso 0.47.0. El proceso 0.47.0 recibe el mensaje `ack` del proceso 0.46.0 y los mensajes `reject` de otros dos procesos 0.48.0 y 0.49.0 y envía la respuesta (mensaje `ack`) a su padre 0.45.0. El proceso 0.45.0 recibe el mensaje del proceso 0.47.0 y como no hay más vecinos para esperar, termina este algoritmo. El árbol de recubrimiento generado se muestra en la figura 2.4 con flechas rojas.

2.3.4. Modulo `tarry_ST`

Al igual que en el algoritmo `term_ST`, en este módulo `tarry_ST` se implementa la parte lógica del algoritmo. Los estados que tiene cada nodo se cambian a través de mensajes. El estado de cada nodo contiene el PID del mismo, la lista de los vecinos, el padre, la lista de los vecinos a los que les ha pasado la ficha (`used`) y la lista de aristas del árbol. Los mensajes usados son las siguientes:

- **Mensaje `root(Who)`**

Este mensaje se usa para decidir la raíz del árbol. Antes de empezar el algoritmo envía este mensaje al nodo que hará de raíz del árbol de recubrimiento. Cuando un nodo recibe este mensaje, pasa la ficha al primer vecino de su lista de vecinos. Se selecciona el primer vecino para pasar el mensaje `token`[3], e inserta este vecino en la lista de vecinos visitados para cambiar el estado de este nodo. Como el nodo raíz no tiene padre, marca a su padre como “no” y mientras actualiza la lista `used` de su estado.

- **Mensaje `token(Who)`**

Este mensaje es para pasar la ficha desde un nodo al otro. Cuando un nodo recibe este mensaje, primero verifica si es la primera vez que se ha recibido la ficha. Esto lo lleva a cabo examinando su padre: si no tiene padre todavía, significa que es la primera vez llega la ficha; si ya tiene el padre, significa que la ficha ya le ha llegado previamente a través de algún otro nodo.

En el caso de recibir por primera vez la ficha, el nodo reconoce al emisor de la ficha como su padre, reenvía la ficha a su primer vecino de la lista de los vecinos que aún

no están visitados y añade este vecino en la lista de los visitados. Si no quedan vecinos por visitar, se devuelve la ficha a su padre.

En el caso de que ya hubiese recibido la ficha, también envía la ficha a su primer vecino de la lista de los vecinos no visitados, y lo añade en la lista visitados. Si no quedan vecinos sin visitar examina si este nodo es `root`. Si no es `root`, envía la ficha a su padre junto con su subárbol. Si es `root`, termina el algoritmo.

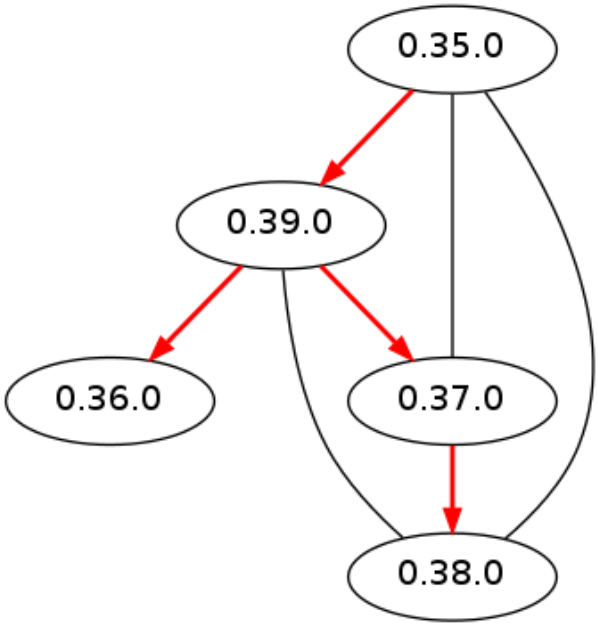
- **Mensaje `token(Who, lista de arcos del árbol)`**

Este mensaje es para añadir un subárbol al árbol correspondiente a un nodo. Cuando un nodo recibe este mensaje, añade esta lista de arcos del árbol en su propio árbol y actualiza su estado.

En la figura 2.5 se muestra un ejemplo para entender mejor el proceso del algoritmo `tarry_ST`. Supongamos que se han generado 5 procesos y el grafo conexo ya ha sido generado. Supongamos que los nodos son

[0.35.0, 0.36.0, 0.37.0, 0.38.0, 0.39.0], y los arcos son: [{0.35.0, 0.37.0}, {0.35.0, 0.38.0}, {0.35.0, 0.39.0}, {0.37.0, 0.38.0}, {0.37.0, 0.39.0}, {0.36.0, 0.39.0}, {0.38.0, 0.39.0}].

Figura 2.5 Ejemplo del algoritmo `tarry_ST`



El nodo 0.35.0 se selecciona como raíz. Envía la ficha a su primer vecino (0.39.0) de la lista de vecinos que son 0.39.0, 0.37.0, y 0.38.0.

El nodo 0.39.0 recibe la ficha por la primera vez. Por tanto, acepta a 0.35.0 como su padre. Sus vecinos son 0.36.0, 0.37.0 y 0.38.0. Se envía la ficha a su primer vecino 0.36.0. Como 0.36.0 no tiene ningún vecino, devuelve la ficha a su padre 0.39.0. El nodo 0.39.0 recibe la ficha de nuevo, envía la ficha a su primer vecino que no ha sido visitado (0.37.0). El nodo 0.37.0 recibe la ficha por primera vez y luego la reenvía al nodo 0.38.0. El nodo 0.38.0 recibe la ficha por primera vez, luego tiene a 0.37.0 como padre. El nodo 0.38.0 envía ficha al nodo 0.39.0. El nodo 0.39.0 tiene un vecino aún no visitado. Por tanto, se lo envía la ficha al nodo 0.38.0. Hasta este momento al nodo 0.38.0 le queda un vecino por visitar, el nodo 0.35.0, por lo tanto le envía la ficha a éste. El nodo 0.35.0 tiene dos vecinos sin visitar que son nodo 0.37.0 y el nodo 0.38.0. Envía la ficha al nodo 0.37.0. El nodo 0.37.0 tiene un vecino sin visitar que es 0.35.0, entonces se lo envía. Ahora al nodo 0.35.0 le queda un vecino por visitar que es el nodo 0.38.0. Se lo envía a éste último. Como el nodo 0.38.0 ya no tiene más vecinos a los que visitar. Devuelve la ficha a su padre (0.37.0). De igual modo el nodo 0.37.0 envía la ficha a su padre (0.39.0), y el nodo 0.39.0 la envía a su padre (0.35.0). El nodo raíz (0.35.0) ya no tiene más vecino a los que visitar, por lo que termina el algoritmo. El árbol de recubrimiento generado se muestra en la figura 2.5 con flechas rojas.

2.4 Comparativa de rendimiento entre los dos algoritmos

A continuación hemos realizado una comparativa de rendimiento entre los algoritmos `term_ST` y `tarry_ST`. Utilizamos dos grafos conexos diferentes para la comparativa y ejecutamos cada algoritmo 50 veces para cada uno para obtener una media del tiempo de ejecución. El resultado es el siguiente:

Figura 2.6 Comparativa de rendimiento entre los dos algoritmos con grafo con 9 nodos y 15 aristas

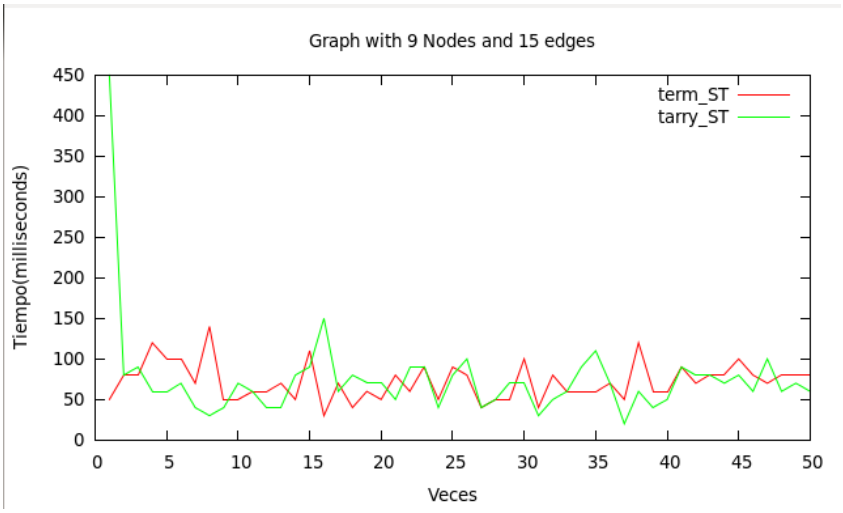
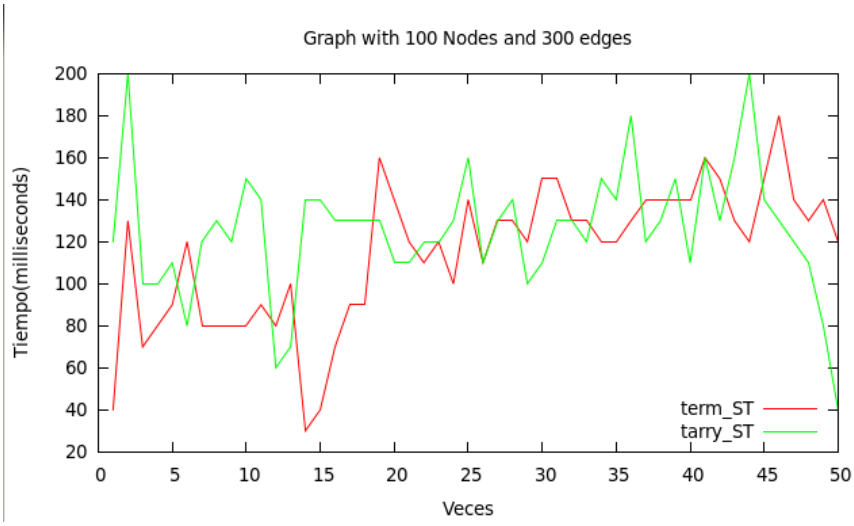


Figura 2.7 Comparativa de rendimiento entre los dos algoritmos con grafo con 100 nodos y 300 aristas



En la figura 2.6 comparamos los algoritmos con un grafo que contiene 9 nodos y 15 aristas. El promedio del tiempo de ejecución del algoritmo `term_ST` es 72 milisegundos y del algoritmo `tarry_ST` es 74.8 milisegundos. En la figura 2.7 comparamos los algoritmos con grafo que contiene 100 nodos y 300 aristas. El promedio del tiempo de ejecución del algoritmo `term_ST` es 114.6 milisegundos, y del algoritmo `tarry_ST` es 125.4 milisegundos. No obstante, a la vista de los gráficos no se observan una tendencia clara a favor de uno u otro. Con estos grafos el rendimiento medio del

algoritmo `term_ST` es mejor que el del algoritmo `tarry_ST`, aunque con grafos más grandes hemos observado que puede invertirse el resultado. Intuimos que el rendimiento está relacionado con la densidad de los grafos, pero no hemos realizado experimentos concluyentes al respecto.

Capítulo 3 - Algoritmo para el cómputo del árbol de recubrimiento mínimo

En este capítulo se introduce un algoritmo distribuido desarrollado por Gallagher y Humblet y Spira[15], para construir el árbol de recubrimiento mínimo para un grafo no dirigido. Es una versión distribuida del algoritmo de *Kruskal*[16].

3.1. Introducción al algoritmo de cálculo del árbol de recubrimiento mínimo

Este algoritmo opera sobre un grafo no dirigido valorado (o ponderado), es decir, con un peso asociado a cada arista. Un árbol de recubrimiento mínimo (en inglés *minimum spanning tree*[17], MST) para un grafo G es un árbol de recubrimiento tal que la suma total de los pesos de las aristas es mínima respecto a los árboles de recubrimiento posibles G[3]. Es decir, dado un grafo G(V, E) no dirigido, donde V es el conjunto de vértices y E es el conjunto de aristas, (υ, ν) presenta la arista entre los vértices υ y ν, $\omega(\upsilon, \nu)$ presenta el peso ponderado de esta arista. Dado $T \subseteq E$, definimos el peso de T como:

$$\omega(T) = \sum_{(u,v)\in T} \omega(u, v)$$

Si G'(V, T) es un árbol de recubrimiento de G, y ω(T) es lo más pequeño posible, diremos que G' es el árbol de recubrimiento mínimo de G.

Hay muchas aplicaciones basadas en el árbol de recubrimiento mínimo. Por ejemplo, en el diseño de redes de comunicaciones: supongamos una empresa con oficinas en distintas ciudades que necesita alquilar líneas telefónicas para tener conexión entre dos oficinas cualesquiera; supongamos además que la compañía telefónica cobra importes distintos para comunicar las distintas parejas de ciudades; el problema consiste en diseñar la red de interconexión para conectar todas las oficinas con un coste mínimo. En este caso puede utilizar el árbol de recubrimiento mínimo para excluir la utilización de las líneas más caras, reemplazándolas por otras más baratas y abaratar así el coste global. También se pueden plantear ejemplos similares, con sus peculiaridades, en otros ámbitos como el

diseño de la red eléctrica, la red hidráulica, la red de cable de televisión, la red de cable de computación, etc[18].

Hay dos variantes del algoritmo para el cálculo del árbol de recubrimiento mínimo: síncrona y asíncrona. En este capítulo, abordamos la variante asíncrona. Esta variante es útil sobre todo cuando un nodo quiere enviar el mensaje a todos los nodos del grafo (*broadcast*) o quiere pedir algo a todos los nodos (*convergecast*) . El uso de un árbol de recubrimiento mínimo garantiza un rendimiento óptimo para estas dos operaciones.

El algoritmo asíncrono que presentamos, que denominamos `AGHS_MST`, se lleva a cabo a través de lo que se denomina *crecimiento del fragmento.* Denominamos fragmento a cualquier subárbol del árbol de recubrimiento mínimo. Al inicio del algoritmo cada nodo es un fragmento y, a medida que el algoritmo avanza, los fragmentos se combinan entre sí mediante envío de mensajes hasta conseguir un único fragmento que será el árbol de recubrimiento mínimo. Un fragmento de un árbol de recubrimiento mínimo también es su subárbol[3].

El algoritmo comienza despertando uno o varios nodos, cada uno de ellos formando parte de su fragmento inicial, compuesto únicamente por dicho nodo. Como cada nodo conoce el peso de las aristas adyacentes, busca la de menor peso y contacta con el nodo que está al otro lado de esa arista. Llamamos a esta arista MWOE (del inglés: *minimum weight outgoing edge*). En general la MWOE de un fragmento es la arista de menor peso que conecta un vértice de dicho fragmento con otro vértice fuera del mismo. Cada fragmento puede buscar su MWOE de modo asíncrono. El *nivel* del fragmento está en relación con el número de fusiones que han dado lugar a dicho fragmento. Por ejemplo, el nivel de un fragmento F1 que sólo contiene un nodo es 0. Cuando se combina con otro fragmento F2, también de nivel 0, se forma un nuevo fragmento F3 cuyo nivel es 1. Una vez que se ha encontrado la MWOE de un fragmento, este fragmento se intenta combinar con otro fragmento o ser absorbido por él (dependiendo los niveles de los fragmentos involucrados) a través de esa arista. El *núcleo* de un fragmento es la arista cuyos nodos son raíces del fragmento. El núcleo del fragmento se cambia a medida que los fragmentos se van fusionando. Un aspecto curioso de este algoritmo es que la arista núcleo es considerada como la raíz del MST

correspondiente a este fragmento. Al finalizar el algoritmo puede seleccionarse cualquiera de los dos nodos adyacentes al núcleo para que sea la raíz del MST resultante. La arista núcleo se caracteriza porque los nodos adyacentes a ella se identifican mutuamente como padres. La función del núcleo del fragmento es decidir la arista MWOE. En concreto: dado un fragmento F cuyo nivel es L y sea F' el fragmento con nivel L' que está al otro lado del MWOE de F, si L = L' y ambos fragmentos F y F' tienen el mismo MWOE, se combinan a un nuevo fragmento F'' con nivel L+1 y este MWOE se convierte en el núcleo del fragmento F''. Si L > L', el fragmento F' es absorbido por el fragmento F, el nivel del fragmento resultante es el mismo que el de F; además el núcleo de F se convierte en el núcleo del fragmento resultante. Si L < L', el fragmento F no se puede combinar o ser absorbido por F' hasta que tenga un nivel mayor o igual que el nivel del fragmento F'. El algoritmo termina cuando sólo existe un fragmento. Además en el árbol de recubrimiento mínimo generado el nodo no sólo conoce a sus hijos sino también a su padre.

3.2. Descripción detallada del algoritmo AGHS_MST

El la sección anterior hemos introducido el funcionamiento general de este algoritmo. En esta sección se explica con más detalle.

Para comprender en profundidad el funcionamiento del algoritmo, hay dos propiedades importantes a destacar:

- Dada una arista que es MWOE de un fragmento, ésta une dicho fragmento con el fragmento correspondiente al vértice que está al otro lado de la arista MWOE. Es decir dos fragmentos dan lugar a una combinación o absorción a través del MWOE de uno de ellos.
- Si todas las aristas del grafo tienen distinto peso, entonces el árbol de recubrimiento mínimo resultante es único.

Este algoritmo se puede implementar en grafos cuyas aristas contienen pesos repetidos, pero para conseguir el único árbol de recubrimiento mínimo en este capítulo, utilizamos las aristas con peso distinto. Esto requerimiento del algoritmo no plantea ninguna limitación en la práctica. En el caso de haber varias aristas con mismo peso, deberemos añadir el peso de dichas aristas la información necesaria para que cada una de

ellas quede identificada unívocamente. Una vez hecho esto, podemos identificar cada fragmento mediante el peso de arista núcleo del mismo.

3.2.1. Los estados básicos

En este algoritmo, tienen estado no sólo los nodos, sino también las aristas. Los estados de los nodos son los siguientes:

- `sleeping`: es el estado inicial del nodo. Significa que dicho nodo todavía no ha sido llamado para participar activamente en el cálculo del MST. Cuando se quiere comenzar el algoritmo, se deberá despertar a uno o varios de los nodos. También se produce el despertar de un nodo cuando es contactado por primera vez desde otro nodo.
- `find`: un nodo está en este estado cuando está buscando su MWOE de su subárbol correspondiente
- `found`: en otros momentos, por ejemplo después de enviar mensaje `connect` o `report`. Un nodo está en el estado `found` cuando se ha encontrado el MWOE de su subárbol correspondiente y se está a la espera de (1) transmitir esta arista a la arista core del fragmento y (2) recibir confirmacion de la fusión con el fragmento que está al otro lado de la MWOE.

Dentro de cada nodo se almacena la información de cada una de sus aristas adyacentes. Cada una de ellas puede estar en uno de los estados descritos a continuación.

  - `basic`: no está decidido si esta arista será parte del MST. Los dos nodos que unen esa arista pueden estar en fragmentos diferentes o en el mismo fragmento.
  - `rejected`: esta arista no formará parte de MST. Los dos nodos que une están en el mismo fragmento.
  - `branch`: esta arista formará parte de MST. Los dos nodos que une están en el mismo fragmento.

Cada fragmento tiene su identificador que es el peso de la arista núcleo. Un nodo puede ser despertar por un mensaje externo para iniciar el algoritmo o por mensaje de otro nodo. En ese momento su estado se cambia de `sleeping` a `find`. En cualquier

caso, lo primero que hace un nodo recién despertado es elegir la arista adyacente con mínimo peso para marcar su estado como `branch` y enviar el mensaje `connect` al nodo que está al otro lado de esta arista. También cambia el estado del nodo de `find` a `found`.

3.2.2. Búsqueda del MWOE

El MWOE de un fragmento solo se puede buscar entre las aristas que están en estado `basic`. El núcleo del fragmento se encarga de enviar el mensaje `initiate` a todos sus hijos para que ellos busquen el MWOE del fragmento en el que están. Cuando un nodo recibe este mensaje `initiate`, si tiene vecinos cuyo estado es `branch` (esto es, ya forman parte del MST) les reenvía este mismo mensaje. Por otro lado, y en cualquier caso, si el estado de este nodo es `find`, que significa que se está buscando una arista MWOE, se selecciona la arista que tiene peso mínimo entre todas sus aristas adyacentes con estado `basic` (es decir, aquellas que no han sido descartadas para formar parte del MST). Luego comprueba si esta arista puede formar parte del MST comparando el identificador del fragmento de los nodos que están a ambos lados de la arista. Si denotamos a estos dos nodos mediante X e Y, esta comparación se realiza mediante el paso de un mensaje `test` desde X hasta Y, indicando el identificador del fragmento de X, y mediante una respuesta `accept/reject` desde Y hasta X, en función de si el identificador del fragmento de Y coincide con el de X.

- Si los dos nodos pertenecen a mismo fragmento , se rechaza a esta arista para formar parte del MST, pues en caso contrario se formará un ciclo en el árbol resultante. Además, en este caso, los nodos X e Y marcan el estado de esta arista como `rejected`, y el nodo X repetirá el proceso con la segunda arista cuyo peso ponderado es menor que los restantes.
- En el caso de que las identidades sean distintas, si el nivel de fragmento de X es mayor o igual que el de Y, este último responde con el mensaje `accept` Si el nivel de X es mayor que el de Y entonces Y no envía ningún mensaje a X y guarda el mensaje test hasta que el nivel de Y llegue al nivel de X.

Cuando un nodo ya ha encontrado a la mejor arista vecina que puede ser candidata a MWOE, espera a recibir los pesos de las aristas candidatas a MWOE

encontradas por sus hijos. Una vez recibidos, los compara con el peso de su propia candidata, y envía el mensaje `report` a su padre con el peso arista de menor peso entre todas ellas. Además, el estado del nodo se convierte a `found`.

Los mensajes `report` se van enviando desde los nodos hijos hasta los padres con los mejores candidatos a MWOE encontrados en sus respectivos subárboles. Cuando estos mensajes llegan a la arista núcleo del fragmento, los nodos a ambos lados de esta arista se envían el mensaje `report` mutuamente para decidir qué MWOE es menor. A continuación, el nodo de la arista núcleo que tenga el menor MWOE, envía el mensaje `changeroot` a esta arista MWOE (a través de sus hijos) para hacer la combinación o absorción del fragmento que está al otro lado del MWOE. Esto se realiza mediante el envío del mensaje `connect` a través de la arista MWOE.

3.3. Implementación en Erlang

Una vez que presentada la descripción teórica del algoritmo `AGHS_MST`, ahora se explica cómo se implementa en Erlang. En la siguientes subsecciones se detalla la implementación de los módulos correspondientes.

3.3.1. Módulo `aghs`

En este módulo se implementa la parte lógica de este algoritmo. El contenido del estado de cada proceso es el `PID` del shell, el `PID` de él sí mismo, la lista de PID de los vecinos (`Neighbors`), el estado del nodo (`StateNode`), el nivel del fragmento al que pertenece (`Level`), el identificador de este último (`Fragment`), la arista que conduce a la mejor MWOE encontrada entre las aristas (externas al fragmento actual) del nodo y las de sus hijos (`BestEdge`), el peso mínimo de dicho MWOE (`BestWT`), el padre (`in-branch`), la arista vecina externa al fragmento que se está investigando actualmente para saber si puede ser candidata a MWOE (`TestEdge`), la lista de PIDs de los hijos que quedan por responder (`FindCount`) y la lista de las aristas del subárbol que queda por debajo del nodo (`Tree`). Dentro de la lista de vecinos cada nodo se forma en una tupla{`PID` del vecino, estado de arista entre vecino y sí mismo (`StateEdge`), peso de esta arista (`Weight`)}.

Los estados que tiene cada nodo se cambian a través de mensajes y funciones tal como explicamos a continuación:

- **Mensaje `wakeup`**

Esta mensaje es para que el nodo que lo recibe llame la función `wakeup`/1 que lo despierta. Este mensaje es recibido por un nodo en estado `sleeping` (el estado inicial de todos los nodos) cuando es seleccionado para empezar el algoritmo, o bien cuando recibe algún mensaje de otro nodo.

- **Función `wakeup`**

Cuando un nodo ejecuta esta función busca la arista adyacente de menor peso. Marca el estado del vecino al otro lado de esta arista como `basic` y envía el mensaje `connect` junto con su nivel (en este caso es 0) al vecino que está al otro lado de esa arista. Cambia el estado de esta arista a `branch` y actualiza la lista de vecinos (porque StateEdge está dentro de tupla de cada nodo de `Neighbors`) y actualiza su estado a `found`.

- **Mensaje `connect(PID de remitente, nivel de remitente)`**

Este mensaje sirve para conectar un nodo con otro. Cuando un nodo recibe este mensaje, en primer lugar se examina su estado. Si está durmiendo, hay que despertarlo mediante la función `wakeup`. El siguiente paso es comparar el nivel del remitente con el propio:

  - Si el nivel propio es mayor que el del remitente, se va a producir una absorción del fragmento correspondiente al remitente. Se marca el estado de la arista como `branch`, como `StateEdge` está dentro de lista de los vecinos entonces la actualiza, y se envía el mensaje `initiate` (junto con el nivel propio, el nombre de fragmento y el estado del nodo) al vecino que está al otro lado de esta arista. Si el estado del nodo es `find`, se incrementa el valor de `FindCount` (ya que ahora habrá un hijo más que va a realizar la búsqueda de un posible MWOE) . En caso contrario, devuelve el estado actual sin cambiar nada.

- Si el nivel propio es menor o igual que la del otro nodo y el estado de esta arista es `basic`, el mensaje tiene que esperar en la cola de los mensajes del buzón hasta que se cambie esta situación (hasta que su propio nivel sea igual o mayor que el del otro nodo). En otro caso, se va a producir una fusión entre los dos fragmentos. Para ello se envía el mensaje `initiate` (junto con su nivel más 1, el peso de esta arista y el estado del nodo que es `find`) a otro nodo. Este mensaje se propagará a los nodos del fragmento correspondiente a este último.

- **Mensaje `initiate(PID de remitente, nivel, nombre de fragmento y el estado de nodo)`**

Cuando un nodo recibe este mensaje, lo primero que hace es actualizar su nivel, el identificador de fragmento al que pertenece y su estado para que sea igual que los parámetros. El nodo aceptará al remitente como su padre, y modificará su variable `inbranch`. Además, se reinician las variables que contienen la mejor arista encontrada Luego, si en este momento entre todas sus aristas adyacentes (excluida la que conecta con el remitente) hay alguna en la que el estado es `branch`, se reenvía mensaje `initiate` a los nodos que están al otro lado de este tipo de aristas (que serán sus hijos en el árbol) para que éstos actualicen su información respectiva. Además, si el estado de nodo es `find`, añade los vecinos a los que envía mensaje `initiate` en la lista de los hijos de los que esperar respuesta(`FindCount`) y llama la función `test/1` (el parámetro es el estado actual) para iniciar la búsqueda de candidatas a MWOE entre sus aristas adyacentes. En otro caso devuelve el estado sin cambiar nada.

- **Función `test`**

Si hay algunas aristas adyacentes cuyo estado es `basic`, elige la que tiene menor peso entre todas ellas y le envía mensaje `test` junto con su nivel y el identificador de su fragmento. Después actualiza la arista de prueba (`TestEdge`). En el caso contrario llama directamente a la función `report/1`.

- **Función `report`**

Comprueba que si todos sus hijos ya han enviado el mensaje `report` y no quedan aristas externas vecinas para saber si son candidatas a MWOE. En este caso, actualiza el estado de nodo como `found`, envía mensaje `report` junto con el mejor peso encontrado por este nodo a su padre. En caso contrario, no hace nada.

- **Mensaje `test(PID de remitente, nivel y fragmento)`**

  Cuando un nodo recibe este mensaje, comprueba si está durmiendo, y se despierta en este casoo. El proceso es igual como la primer parte de el mensaje `connect`.

  - En el caso de que el nivel del remitente sea mayor que el propio, pone este mensaje en la cola del buzón hasta que esta situación cambie.
  - En el caso de que el nivel del remitente sea menor que el propio y los identificadores de ambos fragmentos sean distintos, se envía mensaje `accept` al remitente.
  - En otro caso, los identificadores de los fragmentos a ambos lados son iguales Por tanto, si el estado de arista es `basic`, se cambia a `reject`. Como los dos nodos están en el mismo fragmento, si se conectasen mutuamente se crearía un ciclo en el MST. Si el estado de arista no es `basic` no se cambia su estado (pues, en este caso, el estado de la arista ya es `reject`). Además, se comprueba si la arista de prueba (`TestEdge`) es el remitente de este mensaje. Si lo es llama a la función `test/1` para buscar otra arista de prueba. Si no lo es, envía mensaje `reject` al remitente.

- **Mensaje `accept(PID de remitente)`**

  Cuando el nodo recibe este mensaje, reinicia la arista de prueba (`TestEdge`). Luego compara el peso de la arista a través de la cual recibe el mensaje con el peso de la arista de menor peso encontrada hasta ahora que ha calculado en función de los mensajes `report` de sus hijos recibidos hasta ahora (`BestWT`). Si el peso de la arista remitente es menor que `BestWT`, actualiza `BestEdge` y `BestWT` con los datos de la arista remitente. . En cualquier caso, se termina llamando a la función `report/1` para comprobar si quedan hijos por responder.

- **Mensaje `reject(PID de remitente)`**

Cuando un nodo recibe este mensaje, modifica el estado de la arista correspondiente a `reject`, y llama la función `test/1`. En otros casos, llama la función `test/1` directamente.

- **Mensaje `report(PID de remitente, Peso)`**
  Cuando un nodo recibe este mensaje, comprueba si el remitente es el padre.
    - En el caso negativo, si el peso del mejor MWOE encontrada por el remitente es menor o igual que el del mejor MWOE encontrado hasta ahora (`BestWT`) actualiza los datos de la mejor MWOE y elimina al remitente de la lista de los hijos de los que esperar respuesta. Luego comprueba si quedan más hijos que responder (llama la función `report/1`).
    - En el caso afirmativo, resulta que el nodo destinatario y el nodo remitenten están a ambos lados de la arista núcleo. Si el estado de nodo destinatario es `find`, entonces todavía no ha terminado de buscar el candidato a MWOE correspondiente a su árbol. En este caso pone este mensaje a la cola del buzón hasta que se cambie esta situación. En otros caso, en primer lugar comprueba los valores del peso recibido (`Peso`) desde el otro lado del núcleo con el del mejor MWOE encontrado (`BestWT`). Si `Peso` es mayor que `BestWT` se entonces la MWOE definitiva está al otro lado de la arista núcleo. Entonces se llama la función `changeroot/1`, que se propagará desde la arista núcleo hasta la MWOE. En caso contrario, la MWOE definitiva está en su lado y el nodo no hace nada (en algún momento recibirá el mensaje `changeroot` del nodo que está al otro lado de la arista núcleo y lo propagará por su propio subárbol hasta la MWOE). Por último, si recibe un `Peso` infinito, esto significa que ya no quedan aristas externas al otro lado del núcleo. Si, además, `BestWT` también tiene este valor, significa que tampoco quedan aristas externas en el subárbol del nodo actual. En este caso ya no quedan más nodos que incluir en el MST, y el algoritmo ha terminado.

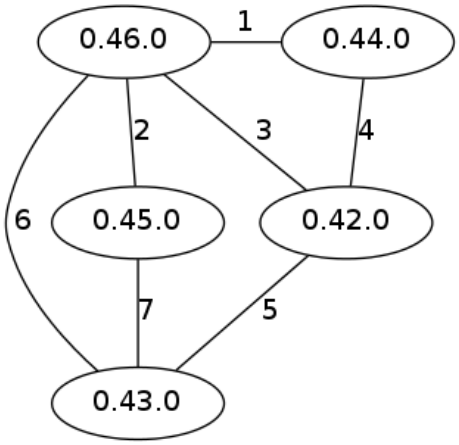
- **Mensaje `changeroot(PID de remitente)`**

Cuando un nodo recibe este mensaje, llama a la función `changeroot/1` directamente.

- **Función `changeroot`**

Propaga el mensaje `changeroot` a la arista adyacente que conduce al MWOE (`BestEdge`). Si el estado de esta arista es distinto de `branch`, entonces `BestWT` es el propio MWOE. En este caso envía el mensaje `connect` a dicha arista y cambia el estado de la misma a `branch`.

Figura 3.1 Ejemplo del algoritmo AGHS: antes de empezar el algoritmo

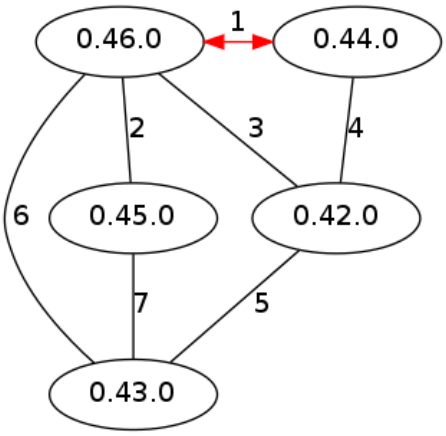


Como ejemplo de funcionamiento de este algoritmo, supongamos el grafo de la figura 3.1, en el que hay cinco nodos que son 0.42.0, 0.43.0, 0.44.0, 0.45.0, y 0.46.0., y siete aristas con los pesos distintos tales como 1, 2, 3, 4, 5, 6, 7. En este ejemplo, se elige el nodo 0.42.0 como el `root`, y empieza este algoritmo despertando a este nodo.

El nodo root compara los pesos de aristas adyacentes, envía mensaje `connect` al nodo correspondiente a la arista de menor peso, que es la 0.46.0. En este momento el estado de 0.42.0 es: el estado de nodo es `found`, el estado de la arista que tiene peso 3 es `branch`, su nivel y `FindCount` (hijos por responder) es 0. Cuando el nodo 0.46.0 recibe el mensaje `connect`, despierta y envía el mensaje `connect` al nodo 0.44.0, poniendo la arista correspondiente a `branch`. Aún no procesa el mensaje `connect`, ya que el estado de la arista que conecta con el remitente 0.42.0 es `basic`. Entonces pone este mensaje en la cola de mensajes para que se procese más adelante. Cuando el nodo 0.44.0 recibe el mensaje `connect`, despierta y envía el mensaje `connect` al nodo

0.46.0, poniendo también la arista correspondiente a `branch`. En este momento, como el estado de la arista que conecta con el nodo 0.46.0 es `branch` (no es `basic`), y los niveles de ambos son iguales a cero, envía mensaje `initiate` a 0.44.0 con nivel 1, fragmento de 1 (peso de la arista) y el estado del nodo pasará a ser `find`. El nodo 0.46.0 recibe el mensaje `connect` desde el nodo 0.44.0, y como tiene la arista correspondiente a `branch`, hace lo mismo que el nodo 0.44.0, (es decir, envía al nodo 0.44.0 el mensaje `initiate`). Es decir, 0.44.0 y 0.46.0 se han enviado el mensaje inititate mutuamente. Cuando el nodo 0.46.0 recibe el mensaje `initiate`, actualiza su nivel como 1, fragmento como 1, estado de nodo como `find` y el padre como el nodo 0.44.0, e inicializa los valores de la mejor MWOE a infinito (es decir, no se ha encontrado MWOE de momento). El nodo 0.44.0 hace lo mismo, y pone a 0.46.0 como su padre. Como de momento los dos nodos no tienen ningún arista cuyo estado sea `branch`, no se propaga `initiate`. Además, dado que el estado de nodo es find, entonces se ejecuta la función `test/1`.

Figura 3.2 Ejemplo del algoritmo AGHS: se conectan los dos nodos 0.46.0 y 0.44.0 como root

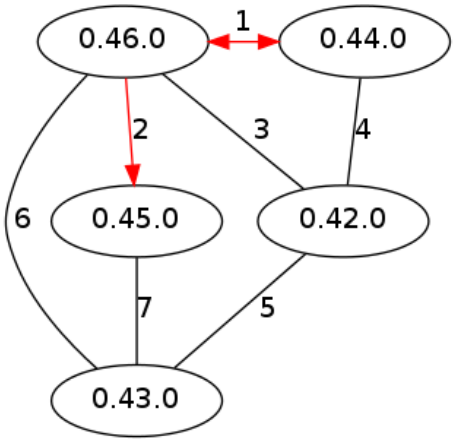


Para el nodo 0.46.0, la mejor arista candidata a MWOE (la arista de prueba)es la que tiene peso 2. Entonces envía el mensaje `test` que contiene su nivel 1 y el fragmento 1 al nodo 0.45.0. En el caso del nodo 0.44.0, la arista de prueba es la que tiene peso 4. Del mismo modo, envía mensaje `test` al nodo 0.42.0.

Cuando el nodo 0.45.0 recibe el mensaje `test`, despierta envía mensaje `connect` al nodo 0.46.0. cuando el nodo 0.46.0 recibe el mensaje `connect` desde el nodo 0.45.0, se realiza una absorción (pues el nivel del fragmento de 0.45.0 es 0, mientras que el de 0.46.0 es 1). Por tanto, 0.46.0 acepta la absorción y cambia el estado de la arista correspondiente a `branch`, y envía mensaje `initiate` que contiene sus suyos (nivel 1, fragmento 1 y el estado del nodo `find`). También añade al nodo 0.45.0 en la lista de hijos que quedan por responder(`findcount`). Ahora el nodo 0.45.0 pertenece al mismo fragmento que el nodo 0.46.0. Ahora procesa el mensaje `test` que recibió del nodo 0.46.0 inicialmente. Como el nodo 0.45.0 ya está en el mismo fragmento que 0.46.0, envía mensaje `reject` a este último. Cuando el nodo 0.46.0 recibe el mensaje `reject`, ejecuta la función `test/1` para buscar el siguiente mejor candidato a MWOE.

Por otro lado, el nodo 0.42.0 recibe el mensaje `test` desde el nodo 0.44.0, como el nivel que se lleva el mensaje es mayor que el suyo, pone este mensaje en final de la cola de mensajes del buzón para esperar hasta que se cambie su situación.

Figura 3.3 Ejemplo del algoritmo AGHS: absorber el nodo 0.45.0 en el fragmento

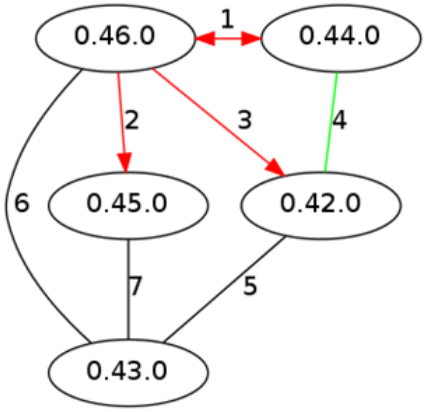


El nodo 0.46.0 tiene un mensaje pendiente (mensaje `connect`) que fue enviado por 0.42.0 al inicio del algoritmo. Como ahora 0.46.0 tiene un nivel mayor que 0.42.0, va a absorber a este último. Para ello pone el estado de la arista que está entre los dos como `branch` y envía mensaje `initiate` al nodo 0.42.0. Cuando 0.42.0 recibe este mensaje `initiate`, actualiza su nivel y fragmento a los del nodo 0.46.0, y también llama la

función `test` para buscar su arista de prueba. El 0.42.0 tiene un mensaje pendiente (mensaje `test`) que recibió de 0.44.0. Como ahora están en el mismo fragmento, el nodo 0.42.0 cambia el estado de esta arista como `rejected` y envía mensaje `reject` al nodo 0.44.0.

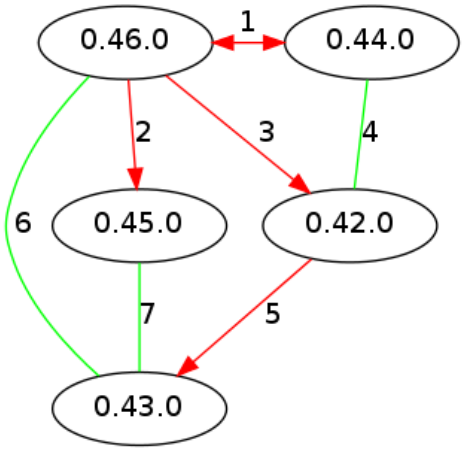
Cuando el nodo 0.44.0 recibe el mensaje `reject` desde 0.42.0, marca esta arista como rejected también. Como no tiene más arista adyacente, llama la función `report/1` directamente.

Figura 3.4 Ejemplo del algoritmo AGHS: absorber y rechazar



El nodo 0.44.0 cambia su estado del nodo como `found` y envía mensaje `report` que contiene el mejor peso encontrado (que es infinito, pues no hay aristas MWOE en su lado) a su padre 0.46.0. El nodo 0.42.0 y el nodo 0.45.0 buscan sus aristas de preuba, y envían el mensaje al nodo 0.43.0. El nodo 0.43.0 despierta y envía mensaje `connect` al nodo 0.42.0. El nodo 0.42.0 recibe este mensaje y cambia el estado de esta arista como `branch` y envía mensaje `initiate` a 0.43.0. El nodo 0.43.0 recibe el mensaje `initiate` y actualiza su fragmento y nivel. Luego envía mensaje `reject` a las peticiones `test` del nodo 0.45.0 y al nodo 0.46.0, porque ya están en el mismo fragmento.

Figura 3.5 Ejemplo del algoritmo AGHS: termina el algoritmo



Como el nodo 0.46.0 no tiene más aristas adyacentes, llama la función `report/1`. En la función de `report/1`, envía mensaje `report` que tienen su peso mejor que es infinito (ya no hay más aristas candidatas a MWOE en su lado) al nodo 0.44.0.

Como los nodos a ambos lados de la arista núcleo, 0.46.0 y 0.44.0, han recibido un `report` con el peso infinito, termina el algoritmo.

Capítulo 4 - Broadcast y convergecast

En este capítulo se van a explicar los algoritmos de `broadcast` y `convergecast` y sus implementaciones en Erlang.

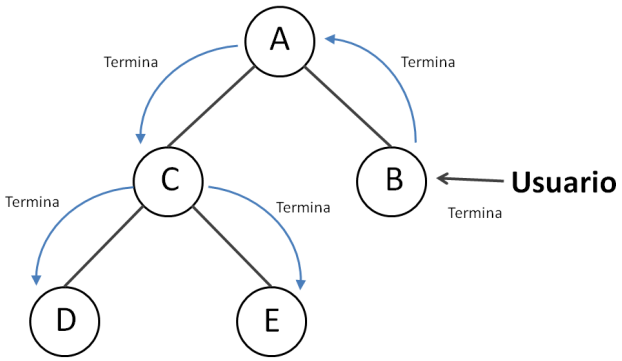
El algoritmo `broadcast` sirve para enviar una notificación a todos los nodos del grafo, sin esperar ninguna respuesta de ellos. Este tipo de notificaciones se pueden generar, por ejemplo, cuando se crea un nodo nuevo, cuando se cae un nodo, etc. Para propagar la notificacion utilizamos el árbol de recubrimiento del grafo, o su árbol de recubrimiento mínimo.

El algoritmo `convergecast` se utiliza para enviar una petición a todos los nodos del grafo y esperar una respuesta de los mismos. Este tipo de peticiones pueden servir, por ejemplo, para conseguir la distancia del nodo más lejano a partir de un nodo dado, la suma de los valores de todos los nodos (si, por ejemplo, cada nodo contuviese un valor), acceder a determinada información que se encuentra de manera distribuida a lo largo de varios nodos, etc. Al igual que el algoritmo `broadcast`, propagamos las peticiones y respuestas de `convergecast` a través del árbol de recubrimiento o el árbol de recubrimiento mínimo del grafo. En ambos algoritmos, el cliente puede realizar la notificación y petición a cualquier nodo de la red para su propagación; no tiene por qué empezar necesariamente desde la raíz del árbol de recubrimiento o del árbol de recubrimiento mínimo.

4.1. Algoritmo `broadcast`

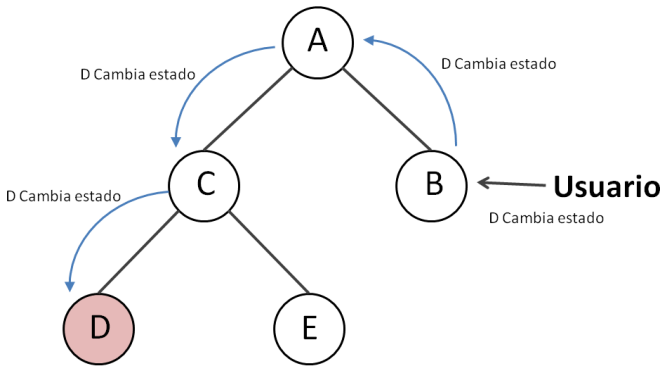
Antes de iniciar el algoritmo, supondremos que ya hemos calculado el árbol de recubrimiento (mínimo) de un grafo mediante alguno de los algoritmos vistos en los capítulos anteriores[19]. Cuando un nodo recibe una notificación de `broadcast`, la reenvía a todos los sus restantes vecinos del árbol de recubrimiento, esto es, el padre (si lo hubiese) y sus hijos. El envío es asíncrono, por lo que no es necesario esperar ninguna respuesta por parte de los nodos. La notificación puede ser de cualquier tipo que no requiera respuesta, por ejemplo, para forzar la terminación de los nodos de la red, o para cambiar el estado de alguno de sus nodos.

Figura 4.1 Ejemplo de broadcast



En la figura 4.1 se muestra un ejemplo del funcionamiento de `broadcast`. Supongamos cinco nodos A, B, C, D, y E en el grafo. Supongamos ya está construido el árbol de recubrimiento. El nodo raíz es A. El usuario quiere notificar la terminación de todos los nodos del grafo. Para ello selecciona aleatoriamente un nodo (en este caso B) y le envía la petición `terminar`. B recibe esta petición, la reenvía a A y termina. De la misma manera para A, que la renvía a C y termina, y éste último, a su vez, la reenvian a D y E, que no la propagan más, ya que no tienen más vecinos en el árbol del recubrimiento (excepto C, que está excluido) Tras la terminación de D y E,, el algoritmo termina.

Figura 4.2 Ejemplo de broadcast para un nodo de la red



En este ejemplo el grafo ye el árbol de recubrimiento es igual como lo de ejemplo anterior. Pero en este caso el usuario solamente quiere cambiar el estado de D, y envía la petición a B. B recibe la petición y se lo envía a A, porque no es D. A recibe la petición de la misma manera se lo envía a C. Como D es un hijo de C, C envía esta petición a D no la envía a E. Cuando D recibe la petición, la implementa y termina esta algoritmo.

4.2. Algoritmo `convergecast`

Al igual que `broadcast`, `convergecast` utiliza el árbol de recubrimiento o árbol de recubrimiento mínimo, que se supone calculado previamente. El algoritmo comienza desde cualquier nodo escogido por el cliente. Cuando un nodo recibe la petición la reenvía a todos los sus vecinos del árbol de recubrimiento (excepto al remitente de la misma) y espera sus respuestas. El algoritmo `convergecast` no termina hasta que las respuestas de cada nodo se han propagado y reunido a través de la red hasta el nodo que recibió la petición del cliente por primera vez. La petición puede ser de cualquier tipo y viene acompañada de una *función de recolección* concreta. Esta función indica a los nodos cómo deben combinar las respuestas obtenidas en una única respuesta que se devolverá al peticionario. El usuario (cliente) deberá implementar esta función de recolección y pasársela al algoritmo de `convergecast`. Por ejemplo si queremos calcular la suma de los valores de los nodos contenidos en una red utilizaríamos la siguiente función:

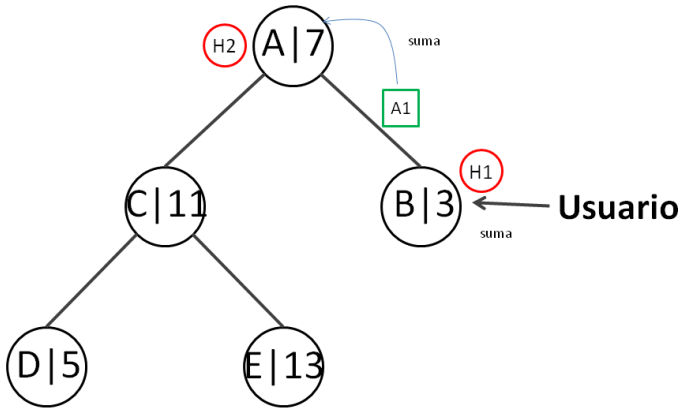
```
funSuma(CurrentPid, Results) ->
   Number = gen_server:call(CurrentPid, get_number),
   Number + lists:sum(Results).
```

Donde la llamada `gen_server:call(CurrentPid, get_number)` devuelve el valor del nodo actual y `lists:sum(Results)` devuelve el valor de la suma de sus vecinos. Similarmente para calcular la distancia al nodo más lejano de la red utilizaríamos la función:

```
funMax(_CurrentPid, []) ->
   1;
funMax(_CurrentPid, Results) ->
   lists:max(Results) + 1.
```

Como el usuario necesita esperar el resultado después de enviar petición a través de `convergecast`, la implementación es síncrona en lugar de asíncrona. Hay que tener en cuenta que, con lo explicado anteriormente, cada nodo de la redtambién se quedará bloqueado después de enviar la petición `convergecast` mientras nole respondan todos sus vecinos. Pero el nodo no debe quedarse bloqueado esperando las respuestas, sino que deberá seguir atendiendo otras peticiones. Para evitar el bloqueo de un nodo, se crean varios procesos que son de dos tipos: proceso `helper` y procesos auxiliares. Cuando un nodo recibe una petición de `convergecast`, lanza un proceso `helper` y varios procesos auxiliares (uno para cada vecino). Cada proceso auxiliar se encarga de enviar una petición `convergecast` al proceso correspondiente y espera conseguir el resultado, para luego enviarlo al proceso `helper`. El proceso `helper` se encarga de recibir los valores que le envían los procesos auxiliares. Una vez que consigue todas las respuestas, aplica la función de recolección dada por el usuario sobre ellas y envía el resultado al proceso que realizó la petición al nodo correspondiente (el nodo que lanzó el `helper`). Este proceso puede ser el cliente que lanzó la petición por primera vez, o el proceso auxiliar creado por otro nodo. A continuación se muestra un ejemplo de funcionamiento del algoritmo:

Figura 4.3 Ejemplo de convergecast, el usuario envía la petición a un nodo



Dado el árbol de recubrimiento de la figura 4.3 con 5 nodos que son A, B, C, D y E, supongamos que cada nodo contiene un valor numérico dentro de su estado.

Supongamos que el usuario quiere obtener la suma de los valores de todos los nodos de la red de procesos y para ello envía la petición `suma` a B.

B recibe la petición y crea H1 como `helper` y A1 como proceso auxiliar, que redirige la petición a A. Del mismo modo: A crea su `helper` H2 y su proceso auxiliar A2 que envía la petición a C. Este último crea H3 y los procesos auxiliares A3 y A4. Como D y E no tienen más vecinos (excepto C, del que recibieron la petición), sólo crean H4 y H5 respectivamente.

Figura 4.4 Ejemplo de convergecast, todos los nodos ya crean helper y assistant(D Y E no lo crean)

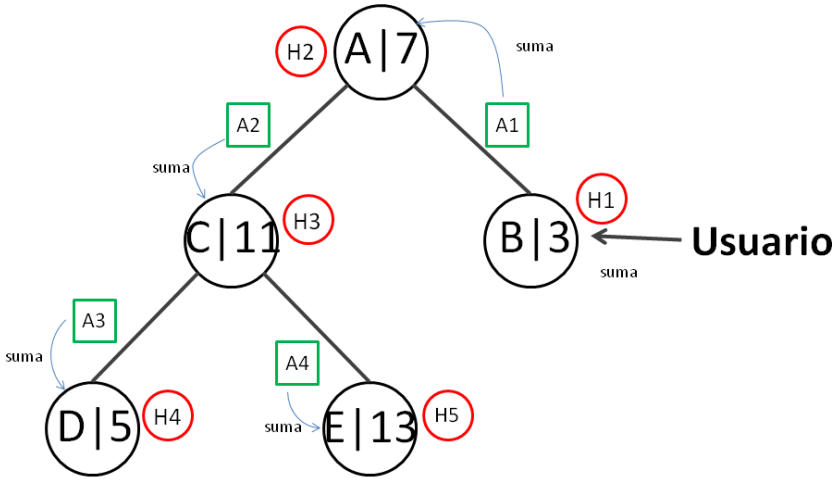
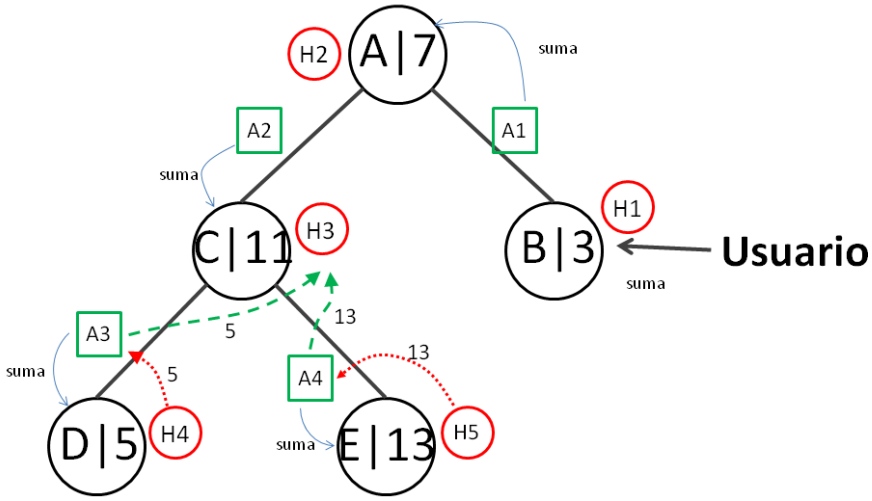
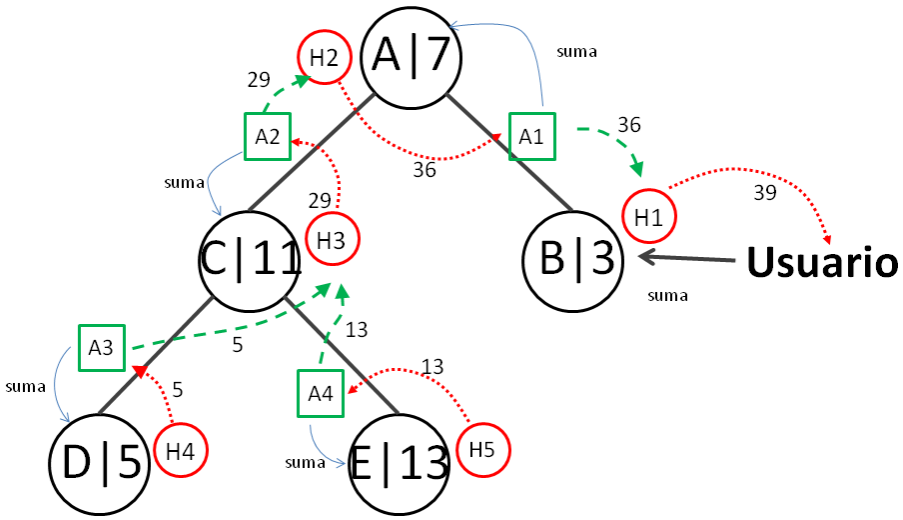


Figura 4.5 Ejemplo de convergecast, A3 y A4 envían respuestas al H3



H4 y H5, al no esperar respuestas de ningún proceso auxiliar, envían directamente 5 y 13 a los procesos auxiliares A3 y A4 respectivamente. A3 recibe el valor 5 y lo envía a H3, el cual sigue esperando la respuesta de A4. A4 envía el valor 13 a H3. Tras esto, H3 ya no tiene más respuestas pendientes para esperar. Entonces aplica la función de recolección que suma los valores de A3 y A4 y C y se lo envía al remitente de la petición suma realizada sobre C, es decir A2. Este último recibe el valor 29 y se lo envía a H2, que no tiene más vecino de los que esperar respuesta, y entonces suma 29 con el valor de A, que es 7 y envía el resultado a A1. Este último recibe el valor 36 y se lo envía a H1. que lo suma con el valor de B para, finalmente, enviar el resultado 39 al usuario. Así termina la ejecución de `convergecast`.

Figura 4.6 Ejemplo de convergecast, termina el algoritmo



4.3. Implementación en Erlang

En esta sección se explican los detalles de implementación en Erlang de los algoritmos vistos en este capítulo. Dado que implementamos este algoritmo partiendo de los algoritmos `term_ST` o `AGHS_MST`, el estado de cada nodo no sólo contiene la información necesaria vista en capítulos anteriores para ejecutar estos algoritmos, sino que también contiene un estado propio que depende de la funcionalidad concreta que se le quiera dar al nodo. En el ejemplo visto anteriormente esta información era un número, pero puede contener otro tipo de información, como un nombre, un fragmento de una base de datos distribuida, etc.

Añadimos estos nuevos elementos en los módulos de los algoritmos asíncronos `term_ST` y `AGHS_MST`.

Cambios a los módulos `term_ST` y módulo `aghs`

Lo primero es añadir el estado propio del nodo a la lista del estado del nodo.

- **Mensaje `{convergecast, PID del cliente o del proceso vecino, función de recolección}`**

Cuando un nodo recibe este mensaje, crea una lista de los vecinos del árbol de recubrimiento a los que se les redirigirá la petición. Esta lista difiere en función de si el nodo es la raíz del árbol de recubrimiento o uno de sus descendientes. . En el caso en que el nodo es raíz, la lista de vecinos adecuados son todos sus hijos excepto aquel del que recibió la petición (en el caso de que la petición no haya sido recibida de un cliente). En otro caso (el nodo no es raíz), la lista de vecinos adecuados son el nodo padre del árbol de recubrimiento y todos sus hijos (de nuevo, se excluye de esta lista el que envió la petición). Luego usa la función `spawn/3` de Erlang para generar el proceso `helper`. El estado del proceso `helper` contiene cinco parámetros que son: el PID del nodo que crea este `helper`, la longitud de la lista de los vecinos adecuados (que es el número de respuestas que debe recibir), la lista de resultados recibidos hasta ahora, PID del cliente o proceso vecino del que recibió la petición (para enviarle la respuesta) y la función de recolección. Una vez generado el proceso `helper`, también se usa función `spawn/1` para crear los procesos auxiliares correspondientes a cada uno de los vecinos a los que redirigir la petición. Para la ejecución del proceso auxiliar, la función tiene cuatro parámetros que son el PID del proceso al que enviarán la petición, el PID del proceso `helper`, el PID del proceso cliente o del proceso vecino que efectuó la petición y la función de recolección. Una vez que el nodo ha creado todos los procesos involucrados en el algoritmo (`helper` y auxiliares) estos se encargarán de contiunar el algoritmo de convergecast, por lo que el nodo devuelve ninguna respuesta (el `helper` responderá por él) y pasa a recibir la siguiente petición.

- **Función `auxiliaryProcess/4`**

Esta función es la que ejecuta cada proceso auxiliar. Esta función envía el mensaje `convergecast` al proceso correspondiente, espera al resultado y lo envía al proceso `helper`.

- **Función `loopHelper/1`**

Esta función `loop`/1 es un bucle de procesos, y es la que ejecuta el proceso `helper`. En este bucle el proceso `helper` sólo espera el resultado enviado por los procesos auxiliares correspondientes a sus vecinos en el árbol de recubrimiento. Si la lista de vecinos es vacía, ejecuta la función de recolección sobre la lista vacía para conseguir el valor que `helper`. responderá al usuario o al proceso vecino. Si la lista de vecinos no

es vacía, cada vez que recibe un mensaje, decrementa el número de respuestas pendientes en uno y añade el resultado recibido a la lista de resultados. Si aún quedan vecinos por responder, actualiza el estado del proceso y sigue esperando. Por el contrario, si ya han respondido todos los vecinos que están en la lista, añade el último resultado en la lista de los resultados y Ejecuta la función de recolección definida por el usuario con la lista de resultados actualizada, y obtiene el resultado que enviará al usuario o al proceso vecino.

Capítulo 5 - Abstracción mediante un comportamiento de Erlang

OTP, la plataforma abierta de Telecom (*Open Telecom Platform*), es una parte integral de la distribución estándar de herramientas del lenguaje Erlang, que son de código abierto[20]. La plataforma es un sistema operativo de aplicaciones que incluye un conjunto de librerías y arquitecturas que permiten construir aplicaciones escalables, tolerantes a fallos y distribuidas[21]. Con respecto a las arquitecturas proporcionadas por OTP, todas se basan en el concepto de comportamiento (*behaviour*).

Un comportamiento encapsula patrones de diseño, de manera similar a las interfaces de un lenguaje orientado a objetos, como Java. Mediante un comportamiento es posible aislar la implementación de una aplicación en dos partes: una parte genérica y una parte específica de la implementación de la aplicación. La parte genérica implementa los aspectos de la arquitectura que no tienen que ver con la funcionalidad de la aplicación concreta en sí, sino aquellos que son comunes a todos los problemas basados en esta arquitectura. La parte específica de la implementación de la aplicación se compone de una serie de funciones *callback* encargadas de implementar la parte funcional de la aplicación, que varía según el problema a resolver. Una función *callback* es un código ejecutable que es pasado como argumento a otro código[22]. La diferencia entre una función estándar en programación y una función *callback* es el llamante de la misma. Una función normal es invocada por el programador, mientras que una función *callback*, aunque está implementada por el programador, es llamada por sistema o arquitectura genérica en el momento apropiado.

Un comportamiento permite utilizar implementaciones genéricas estables que han sido refinadas y probadas, y redundan en una reducción de errores, ya que el programador sólo tiene que concentrarse en los aspectos específicos de su aplicación. También evitan la repetición de código.

Las librerías de OTP proporcionan los siguientes comportamientos:

- `gen_server`: se utiliza para implementar una arquitectura cliente/servidor.
- `gen_fsm`: se utiliza para implementar un servidor que funciona como una máquina de estados finitos.

- `gen_event`: se utiliza para implementar componentes de procesamiento de eventos.
- `supervisor`: se utiliza para implementar árboles de supervisión, en los que un nodo supervisor monitoriza una serie de nodos trabajadores, y se encarga de reinicializarlos en caso de que fallen. Los nodos trabajadores pueden ser, a su vez, supervisores de otros nodos.

El lenguaje también permite al programador definir sus propios comportamientos. Este es el objeto de este capítulo. Se presentará un comportamiento que permitirá realizar peticiones de `convergecast` y `broadcast` sobre una red de nodos, extendiendo uno de los patrones más utilizados de las librerías OTP: el comportamiento `gen_server` (servidor genérico). Este comportamiento implementará los algoritmos vistos en los capítulos 2 y 3 y utilizará el árbol de recubrimiento resultante para realizar las operaciones de `convergecast` y `broadcast`.

5.1. El comportamiento `gen_server`

El comportamiento `gen_server` es el tipo de patrón más común para desarrollar servidores que procesan y responden peticiones de un cliente. El objetivo de utilizar el comportamiento `gen_server` es garantizar la separación entre el código genérico de un servidor y el código de las funciones específicas que obtienen la respuestas de las peticiones realizadas en el mismo. Los códigos de funciones específicas se pueden implementar por el usuario mediante funciones *callback*. De este modo, el usuario sólo necesita implementar la lógica del servidor, en lugar de abordar detalles de bajo nivel, como el paso de mensajes. Además, facilita el mantenimiento de la aplicación, ya que la parte genérica puede ser actualizada sin tener que modificar la funcionalidad lógica del servidor.

Cuando el programador necesita escribir un servidor específico basado en el comportamiento `gen_server` tiene que implementar las funciones *callback* especificadas por éste último. En particular, hay tres pasos que son los siguientes:

- Declarar el comportamiento a implementar mencionando el módulo donde están especificadas las funciones *callback*. Esto se realiza en Erlang mediante la directiva `-behaviour(gen_server).`

- Escribir las funciones *callback* necesarias que implementen la funcionalidad del servidor. Hay seis funciones obligatorias que son `init/1`, `handle_call/3`, `handle_cast/2`, `handle_info/2`, `code_change/2`, `terminate/2`.
- Arrancar el servidor y realizar, en su caso, las peticiones del cliente mediante las funciones `start/3`, `call/2` y `cast/2`. En lugar de llamar a estas funciones directamente, es recomendable proporcionar una interfaz para el cliente que encapsule las llamadas a estas funciones.

A continuación se ofrece una breve descripción de la semántica de las funciones *callback* que se han de implementar para un servidor genérico cualquiera. En todas ellas se hace referencia al *estado* del servidor, que contiene la información que el servidor necesita para atender las peticiones. Esta información depende del servidor concreto a implementar. Por ejemplo, si el servidor implementa una base de datos remota, el estado estará formado por la información almacenada en la base de datos. Las peticiones realizadas al servidor pueden modificar el estado.

- La función `init/1` es llamada cuando se arranca el servidor. Recibe un parámetro que especificará el cliente cuando éste arranque el servidor. Si la inicialización es correcta, debe devolver el estado inicial.
- La función `handle_call/3` maneja las peticiones síncronas realizadas por el cliente a través de la función `call/2`. Recibe la petición, el PID del cliente y el estado actual del servidor, y normalmente devuelve una respuesta junto con el estado siguiente. Al ser una petición síncrona, el cliente que llame a la función `call/3` queda bloqueado hasta que reciba la respuesta. La función `handle_call` puede devolver otro tipo de resultados que alteran el funcionamiento del servidor. Por ejemplo, puede devolver una petición de parada para que el servidor finalice.
- La función `handle_cast/2` maneja las peticiones asíncronas realizadas por el cliente a través de la función `cast/2`. Recibe la petición y el estado actual, y devuelve el estado siguiente. No se devuelve ningún resultado al cliente, ya que éste continúa su ejecución sin esperar ninguna respuesta del servidor.

- La función `handle_info/2` se encarga de gestionar los mensajes recibidos por el servidor que no han sido realizados a partir de las funciones `call` y `cast`. Por ejemplo, aquellos realizados con el operador `!` de envío de mensajes, así como mensajes derivados de notificaciones de *timeout*.
- La función `code_change/3` se utiliza para indicar el nuevo estado en el caso en que se haya realizado un cambio de código en el módulo que implementa las funciones del servidor. Esto es necesario porque en Erlang se pueden realizar actualizaciones del código de un servidor en caliente (es decir, mientras éste está funcionando).
- La función `terminate/2` es llamada cuando se envía una petición de parada al servidor.

5.2. Extensión del comportamiento `gen_server`.

El objetivo de este capítulo es la creación de un comportamiento de Erlang que permita manejar una serie de procesos conectados en red. Cada proceso de la red funciona como un servidor, de forma similar a uno implementado mediante `gen_server`. Sin embargo, en la extensión propuesta en este capítulo, el usuario puede enviar peticiones a uno de estos servidores o a todos ellos a través de las funciones `broadcast` y `convergecast` que se habrán incorporado a este marco. Estas funciones harán uso de un árbol de recubrimiento que se generará en la red de procesos servidores de manera transparente al cliente.

5.2.1. Módulo behaviour_termST/behavior_aghs

En cada uno de estos módulos se especificará, en primer lugar, la interfaz de las funciones *callback* que el usuario de nuestro comportamiento deberá implementar para poder hacer uso del mismo. En nuestro caso hemos decidido especificar una interfaz con las mismas funciones que las requeridas por `gen_server`, ya que cada proceso servidor de la red ofrecerá la misma funcionalidad de un servidor. La únicas funcionalidades adicionales son, como se ha comentado, la posibilidad de hacer `broadcast`, cuyos mensajes pueden ser tratados por la función `handle_cast`, y la posibilidad de hacer `convergecast`, que no requiere el uso de ninguna función adicional. Existe una

diferencia con las funciones *callback* definidas en `gen_server`. Las funciones de nuestro comportamiento comenzarán con el prefijo `b_` (por ejemplo, `b_handle_call`). Las funciones podrán haberse llamado exactamente igual que las de `gen_server`, pero hemos añadido este prefijo para facilitar su distinción con las de este último.

La parte genérica de nuestra implementación de red de servidores utiliza, a su vez, un `gen_server`, que actuará como *envoltorio* de las funciones *callback* implementadas por el programador. El estado del servidor envoltorio se compone de tres partes: el nombre del módulo que contiene las funciones *callback* implementadas por el programador del servidor, la parte que comprende las estructuras necesarias en la generación del árbol de recubrimiento, y la parte que corresponde al estado del servidor "envuelto", que es aquella que es visible para el programador de las funciones *callback* anteriormente mencionadas. Esta parte recibirá, en adelante, el nombre de *estado interior*. A continuación se describe el funcionamiento de las funciones implementadas para el comportamiento `gen_server` del servidor envoltorio, es decir, la parte genérica que implementa la red de procesos:

- **init/1**

Esta función es para inicializar el estado de un nodo. Recibe una tupla que contiene `ModName` (nombre del módulo que va a implementar las funciones *callback* del comportamiento `behaviour_termST o behaviour_aghs`) y `Args` (los argumentos que serán pasados a la función `b_init` de dicho módulo). En el ejemplo que se mostrará en el siguiente apartado el argumento (que coincide con el estado interior) será un número. Este argumento vendrá dado por la función `start_network`[link], que será la que arranque la red de procesos. La función `init/1` llama la función `b_init/1` especificada por el programador dentro del módulo `ModName` pasándole el parámetro `Args` y ésta devolverá el estado interior inicial del proceso. El estado del servidor envoltorio será una tupla formada por `ModName`, el estado interior del proceso y la información relacionada con el algoritmo que construye el árbol de recubrimiento (ya presentada en los capítulos 2 y 3) [link].

- **handle_call/3**

Esta función es para manejar las peticiones síncronas. En nuestro caso hay que implementar dos tipos de peticiones que puede realizar un cliente, que son las peticiones a un único servidor (`call/2`)[link] y las peticiones a toda la red (`convergecast/2`)[link]. Las primeras tendrán la forma `{call, Peticion}` y las segundas tendrán la forma `{convergecast, PidPadre, Peticion}`.

Cuando la petición es simplemente a un único proceso(`call/2`), este proceso recibe la petición y la pasa a su módulo llamando la función `b_handle_call/3`. En `b_handle_call/3` recibe tres parámetros que son la petición, el PID del proceso que realiza la petición (`From`) y el estado interior del servidor (`StateInt`) . Devuelve al servidor su respuesta junto con su nuevo estado interior. El valor devuelto por `handle_call` ha de tener la misma estructura que el devuelto por `b_handle_call`, pero debemos adaptarla para incluir el estado interior dentro del estado del servidor envoltorio, e incluir este último en la tupla resultado.

Si la petición es de tipo `convergecast`, usamos el algoritmo que se ha explicado en la sección 4.2 para implementarla.

- **handle_cast/2**

Esta función es para realizar las peticiones asíncronas. Recibe dos parámetros: la petición (`Request`) y el estado del servidor. No devuelve ninguna respuesta.

Esta función debe manejar distintos tipos de petición, incluyendo aquellas que se realizan en la construcción del árbol de recubrimiento (o el árbol de recubrimiento mínimo, dependiendo del algoritmo).

  - Cuando la petición está relacionada con la construcción de la red de procesos (por ejemplo, `{addEdge, Who}`) o del árbol de recubrimiento (por ejemplo, `{root, P}`, `{probe, Pid}`, `{ack, Pid, RankChild, TreeChild}`, `{reject, Pid}` y `{tree, _Pid, NewTree}` para el algoritmo `term_ST`), se realizan las acciones comentadas en la sección 2.3.3, y se modifica la parte del estado del servidor envoltorio relativa a la construcción del árbol, manteniendo el estado interior sin modificar.

- Cuando la petición es de tipo `{cast, Peticion}`, se trata de una petición asíncrona realizada sobre un único proceso de la red. La manera de implementar esto es similar a las peticiones de la forma `{call, Peticion}` vistas anteriormente. La diferencia es que ahora se llama a la función `b_handle_cast`, que sólo devuelve el estado siguiente.
- Cuando la petición es `{broadcast, {PidOrigin, Peticion}}`, significa que esta petición será propagada por todos los procesos de la red empezando a partir del nodo que recibe esta petición. En este caso, envía la misma petición a todos sus vecinos del árbol de recubrimiento (excepto el padre y el llamante), y luego llama a la función `b_handle_cast` de `ModName` para actualizar esta petición en un proceso. Como `b_handle_cast` no devuelve ninguna respuesta sino tan solo el nuevo estado interior del proceso, el servidor envoltorio actualiza su estado utilizando el nuevo estado interior del proceso.

- **handle_info/2**

Esta función se implementa de manera similar a las dos anteriores. Para ello se delega en la función `b_handle_info` de `ModName`.

- **terminate/2**

Esta función es ejecutada cuando el servidor se va a detener. Recibe el motivo de la terminación (`Reason`) y el estado `State`, y llama a la función *callback* `ModName:b_terminate/2`.

- **code_change/3**

Esta función es para modificar el estado cuando se actualiza el código del servidor en caliente. Recibe la versión antigua (`OlsVsn`), el estado (`State`) y una serie de argumentos adicionales con detalles de la actualización (`Extra`). Llama a la función `b_code_change` de `ModName` y recibe el nuevo estado interior, que será encapsulado dentro del estado del servidor envoltorio.

Las funciones descritas hasta ahora son funciones *callback* que se han utilizado en la implementación particular de `gen_server`. Estas funciones no deben ser llamadas por el cliente. Sin embargo, las funciones que se presentan a continuación sí han sido concebidas para el uso de los clientes.

- **start_network/3**

Esta función es para arrancar una red de servidores. Recibe tres parámetros `Modulos`, `Conexiones` y `Args`, y devuelve una lista con los PIDs de los procesos generados.

`Modulos` es una lista de nombres de módulos. Esta lista tiene tantos elementos como vértices del grafo. Cada uno de los elementros indica el nombre del módulo que implementa las funciones *callback* de este vértice. En el ejemplo mostrado en el capítulo anterior, donde cada nodo almacenaba un único número, todos los nombres de módulos son `modulo_number` (explicado en la siguiente sección). `Conexiones` es una lista de pares `{Num1, Num2, Peso}`, que indica las aristas del grafo. `Peso` es el valor ponderado de la arista entre `Num1` y `Num2`. Estos últimos números representan las posiciones dentro de la lista `Modulos`. `Args` es una lista con tantos elementos como vértices. Cada uno de estos argumentos se van a pasar a la función `b_init/1` del nodo correspondiente. Cada elemento de lista de `Args` puede ser de distinto tipo. En nuestro ejemplo particular, `Args` es una lista de números, que son los números que se almacenarán en los respectivos nodos de la red.

Para implementarlo, lo primero es iniciar los procesos para cada uno de los `Modulos` (mediante `gen_server:start/3`) y pasar a cada uno el elemento correspondiente de `Args`. Luego se crean las conexiones de la red según la lista `Conexiones`. Para ello se utiliza `gen_server:cast/3` para que cada servidor añada su vecino con la petición `{addEdge, Who}`. El siguiente paso es construir el árbol de recubrimiento o el árbol de recubrimiento mínimo, que depende del algoritmo. El código es similar al ya explicado en los capítulos 2 y 3.

- **stop_network/1**

Esta función es para terminar la red de servidores interconectados. Recibe el parámetro PID de un proceso, pues tan sólo es necesario indicar el PID de uno de los nodos de la red. Este nodo envía la petición a los restantes en la red mediante la función `broadcast/2`.

- **call/2**

Esta función es para realizar una petición síncrona a un proceso, que responderá de manera individual. Es decir la petición no se transmite a otros nodos restantes. Recibe dos parámetros `PIDServer` y `Peticion` y envía el mensaje `{call, Peticion}` al PID pasado como parámetro utilizando `gen_server:call/2`.

- **cast/2**

Esta función es similar a `call/2`, pero con peticiones asíncronas. Recibe dos parámetros `PIDServer` y `Peticion`. Envía mensaje `{cast, Peticion}` al PID pasado como parámetro utilizando `gen_server:cast/2`.

- **reply/2**

Esta función es sólamente para llamar a `gen_server:reply/2`. Recibe dos parámetros: `Cliente` y `Reply`, que pasan directamente a `gen_server:reply/2`.

- **broadcast/2**

Esta función es para enviar una petición asíncrona a toda la red. Recibe dos parámetros: `PID` del servidor y `Peticion`. Basta con el PID de uno de los procesos de la red. Esta función envía el mensaje `{broadcast, Peticion}` al proceso indicado en `PID`.

- **convergecast/2**

Esta función es para realizar un cómputo a lo largo de la red. Recibe dos parámetros: `PIDServer` y `F`. El cómputo a realizar viene dado por la función `F`, como se ha explicado en el capítulo anterior. Para ello envía el mensaje `{convergecast, PidPadre, F}` al PID indicado.

5.2.2. Módulo modulo_number

Este módulo implementa la funcionalidad de la red de servidores simple mostrada en el capítulo anterior, en el que cada nodo almacena un número. Para ello implementa las funciones *callback* del comportamiento `behaviour_termST` o `behaviour_aghs`, que se encargan de procesar las peticiones del servidor y de establecer si estado inicial. Lo primero es especificar el comportamiento que se implementa `behaviour_termST` o `behaviour_aghs`, mediante la directiva `-behaviour`, y luego definir las funciones *callback* exigidas por este comportamiento. En `b_init/1`, recibe el número asignado por el programador y lo asigna a su estado. En `b_handle_call/3`, si la petición es `get_number` devuelve su estado, que es el número almacenado. En `b_handle_cast/2`, si la petición es `stop_network`, termina este proceso mediante la devolución de una tupla `{stop, normal, State}`. En este caso, la tupla `{stop, normal, State}` activa `b_terminate/2` e indica el motivo de la terminación (terminación por causas "normales"). En el caso en que vengan otro tipo de peticiones, no se devuelve ningún mensaje.

```
-module(modulo_number).

-behaviour(behaviour_termST).

-export([b_init/1, b_handle_call/3, b_handle_cast/2,
b_handle_info/2, b_terminate/2, b_code_change/3]).

b_init(N) -> {ok, N}.

b_handle_call(get_number, _From, State) ->
     {reply, State, State}.

b_handle_cast({_Pid, Peticion}, State) ->
   case Peticion of
      stop_network ->
         {stop, normal, State};
      {change_state, Value} ->
         {noreply, Value};
      _ ->
      {noreply, State}
```

```
   end.

b_handle_info(_, State) -> {noreply, State}.

b_terminate(_, _) ->
     io:format("Proceso ~p terminado ~n", [self()]).

b_code_change(_, State, _) -> {ok, State}.
```

Capítulo 6 - Conclusiones y trabajo futuro

Los algoritmos presentados en este trabajo responden a necesidades reales en computación distribuida y han sido una buena forma de introducirse en este área. Los dos primeros, `term_ST` y `tarry_ST`, si bien no son los más interesantes (porque no dan solución óptima) han sido una buena base sobre la que entender el algoritmo AGHS, que sí tiene mayor interés. La implementación concreta de estos algoritmos en Erlang no es inmediata y requiere profundizar tanto en los propios algoritmos como en el lenguaje Erlang.

Por otro lado, la presentación del algoritmo AGHS en[3] es incorrecta, lo cual dificultó su comprensión y retrasó el avance del trabajo. Finalmente se tuvo que buscar la fuente original de dicho algoritmo en[22].

La visualización de los grafos y los árboles usando *graphviz* ha resultado ser una herramienta útil para depurar los algoritmos. Esta herramienta muestra el grafo conexo y árbol de recubrimiento (o árbol de recubrimiento mínimo) diferenciando ambos mediante los colores de las aristas. Así el cliente puede revisar, entender y depurar los algoritmos.

En la siguiente parte, que es original de este trabajo, se desarrollan dos algoritmos, `broadcast` y `convergecast`, para propagar peticiones por la red de nodos. Estos dos algoritmos son bien conocidos. Las implementaciones realizadas funcionan de manera efectiva y eficiente gracias al árbol de recubrimiento mínimo, y puede utilizarse para aplicaciones prácticas. Por ejemplo para implementar funciones de acceso y modificación en bases de datos distribuidas.

La implementación del servidor genérico que proponemos no sólo es útil para desarrollar los algoritmos distribuidos presentados, sino que también es un servidor potente para funcionar en aplicaciones con alta concurrencia, estables, tolerantes a fallos y escalables.

Desde el punto de vista personal, este trabajo me ha ayudado mucho a entender Erlang en mayor profundidad. Por ejemplo, en cuanto a cómo trabajan los procesos entre ellos, sobre todo en la parte del comportamiento OTP. Con el trabajo realizado en el capítulo 6, he clarificado mi conocimiento sobre servidores genéricos, en cuanto a su funcionalidad e implementación. Las aplicaciones prácticas sobre Erlang y algoritmos

distribuidos son muy amplias y pueden investigarse en más profundidad. Erlang ha resultado ser un lenguaje muy potente para abordar los algoritmos pretendidos.

Trabajo futuro

Los algoritmos implementados en este trabajo pueden fácilmente encontrar aplicación práctica en diversos contextos, que no se han abordado aún en el mismo. En particular, pueden ser útiles para aplicaciones de bases de datos distribuidas. El acceso a un componente sería un `convergecast`. Por ejemplo, los distintos números telefónicos de una persona pueden estar en distintos nodos de una base de datos distribuida; se podría obtener la lista de números utilizando un `convergecast` con los parámetros adecuados. Se podría introducir redundancia en el almacenamiento de los números de modo que si se pierde uno de ellos, los restantes nodos pueden sincronizarse para su recuperación.

Por otro lado hay problemas abiertos muy interesantes en relación con estos algoritmos:

- La modificación dinámica del grafo de procesos. Como los procesos trabajan en una red, si de repente muere un nodo, puede avisar a todos los nodos de la red de que ya está muerto mediante `broadcast`. Pero habría que actualizar el árbol de recubrimiento. El caso de añadir un nuevo nodo es análogo y plantea el problema de reconstruir el árbol de recubrimiento sin necesidad de recalcularlo desde cero.
- Respecto a la seguridad: en el `convergecast`, los servidores están ejecutando código arbitrario dado por el cliente, por lo que esta librería sólo tiene sentido en contextos *trusted*, ya que si el cliente es externo al sistema, podría hacer que los servidores ejecutasen código malicioso.

Chapter 6 - Conclusions and future work

The algorithms presented in this work correspond to actual needs in distributed computing and have been a good way to introduce ourselves into this area. The first two algorithms, `term_ST` and `tarry_ST`, in spite of not being the most interesting ones (because they do not yield an optimal solution) have been a good basis on which to understand the AGHS algorithm, which is more interesting. The concrete implementation of these algorithms in Erlang is not immediate and requires a deeper knowledge of the algorithms themselves and of the Erlang language.

Moreover, the presentation of AGHS algorithm in[3] is incorrect, which made it difficult to understand and it delayed the progress of this work. Finally we had to refer to the original source of that algorithm in[22].

The display of graphs and trees using *Graphviz* turned out to be an useful tool for debugging the algorithms. This tool shows the connected graph and spanning tree (or minimum spanning tree) by differentiating the colors of the edges, so the user can review, understand and debug the results of these algorithms.

In the next part, which is the original one of this work, we have developed two algorithms, `broadcast` and `convergecast`, that propagate requests in a process network. These algorithms are well known. The developed implementations work effectively and efficiently thanks to the construction of a minimum spanning tree, and can be used for practical applications. For example, to implement functions such as access and modification on distributed databases.

The implementation of the generic server that we propose is not only useful for developing the distributed algorithms presented in this work, but also is a powerful server for running stable, fault tolerant and scalable applications with high concurrency requirements.

Form a personal point of view, this work has helped me to understand Erlang in a deeper way. For example, to know how the processes interact between themselves, especially in the context of OTP behaviours. With the work done in Chapter 6, I have clarified my understanding of generic servers, including its functionality and implementation. The practical applications of Erlang and distributed algorithms are very

wide and can be investigated in more depth. Erlang has proved to be itself a very powerful language for implementing the algorithms described in this work.

Future work

The algorithms implemented in this work can easily find practical applications in different contexts, which have not yet been dealt with in this work. In particular, they can be useful for applications based on distributed databases. The access to a component of the database would be a `convergecast`. For example, the different phone numbers of one person can be in different nodes of a distributed database. The list of numbers could be accessed with a `convergecast` with the appropriate parameters. It could also introduce redundancy in the storage of numbers, so that if one of them is lost, the remaining processes can be synchronized for its recovery.

Besides this, there are very interesting open problems related to these algorithms:

- The dynamic modification of the process graph. As the processes work in a network, if a node suddenly fails, the system can tell the remaining nodes of the network that it is already dead by using a `broadcast`, but in addition we must update the spanning tree. The problem of adding of a new node is analogous and poses the issue of reconstructing the spanning tree without recomputing it from scratch.
- With regard to safety, in the `convergecast` function the processes are running arbitrary code given by the client, so the use of this function only makes sense in trusted environments. Otherwise, a client external to the system might cause the execution of malicious code in the servers.

Referencias

[1] George Michaillidis. (2005). Minimum Spanning Tree.

[2] Logan, Martin. (2011). Erlang and OTP in action. ISBN-978-1-933988-78-8.

[3] K. Erciyes. (2013). Distributed Graph Algorithms for Computer Networks. 1-82. ISBN 978-1-4471-5172-2. ISBN 978-1-4471-5173-9 (eBook)

[4] Gerard Tel. (2000). Introduction to Distributed Algorithms. 1-36. ISBN-0-521-79483-8

[5] Afek, Yehuada. (2013). Distributed Computing. 16-18. ISBN: 978-3-642-41526-5 (Print) 978-3-642-41527-2 (Online)

[6] Andrew S. Tanenbaum, Maarten Van Steen. Distributed Systems Principles and Paradigms. 17-18.

[7] https://es.wikipedia.org/wiki/Computaci%C3%B3n_distribuida

[8] Nivio Ziviani. (2007). Diseño de algoritmos con implementaciones en Pascal y C. 269-271. ISBN: 978-84-9732-538-7

[9] Hutchison David, Takeo Kanade, Josef Kittler, Jon M. Kleinberg, Friedemann Mattern, John C. Mitchell, Moni Naor, Oscar Nierstrasz, C. Pandu Rangan, Bernhard Steffen, Madhu Sudan, Demetri Terzopoulos, Doug Tygar, Moshe Y. Vardi, Gerhard Weikum, Jürgen Lerner, Dorothea Wagner, Katharina A. Zweig (2009). Algorithmics of Large and Complex Networks: Design, Analysis, and Simulation. 13-27. ISBN-9783642020940.

[10] Simon St. Laurent. (2013). Introducing Erlang. Print ISBN-13: 978-1-4493-3176-4

[11] Jim Larson. (2009). Erlang for Concurrent Programming.

[12] Fred Hebert. (2013). Learn You Some Erlang for Great Good! Print ISBN-13: 978-1-59327-435-1

[13] B. Awerbuch. (1987). Optimal distributed algorithms for minimum weight spanning tree, counting, leader election, and related problems. ISBN:0-89791-221-7

[14] http://www.math.ucsd.edu/~ronspubs/85_07_minimum_spanning_tree.pdf

[15] R. G. Gallager, P. A. Humblet, P. M. Spira. (1983). A Distributed Algorithm for Minimum-Weight Spanning Trees.

[16] https://es.wikipedia.org/wiki/Algoritmo_de_Kruskal

[17] Seth Pettie. (2002). An optimal minimum spanning tree algorithm.

[18] http://www.geeksforgeeks.org/applications-of-minimum-spanning-tree/

[19] Yi Song, Jiang Xie. (2014). Broadcast Design in Cognitive Radio Ad Hoc Networks. ISBN: 978-3-319-12621-0 (Print) 978-3-319-12622-7 (Online)

[20] Rubio Jiménez, Manuel Ángel. (2012). Erlang/OTP. Vol. 1, Un mundo concurrente. ISBN-978-1-4709-2152-1.

[21] Joe Armstrong. (2007). Programming Erlang Software for a Concurrent World. 292-310. ISBN-10: 1-9343560-0-X

[22] http://www.cs.tau.ac.il/~afek/p66-gallager.pdf

Bibilografía recomendada

- Cesarini, Francesco. (2009). Erlang programming. ISBN-978-0-596-51818-9.
- Balsa-Canto, Eva. (2010). Algoritmos eficientes para la optimización dinámica de procesos distribuidos.
- Sekerinski, Emil, Sere, Kaisa (Eds.). (1999). Program Development by Refinement. 79-114. ISBN-978-1-4471-0585-5
- J. David Schaffer. (1989). Proceedings of the third international conference on Genetic algorithms. 434-439. ISBN: 1-55860-006-3