Escuela de
Ingeniería y Arquitectura
Universidad Zaragoza

Trabajo Fin de Grado

Diseño e implementación de un método para evitar colisiones en un sistema multi-robot

Design and implementation of a method to avoid collisions in multi-robot systems

Autor:

Javier Oroz Joven

Director:

Cristian Mahulea

(Departamento de informática e ingeniería de sistemas)

Grado en Ingeniería de Tecnologías Industriales
Noviembre de 2016

TRABAJOS DE FIN DE GRADO / FIN DE MÁSTER

Escuela de Ingeniería y Arquitectura Universidad Zaragoza

DECLARACIÓN DE AUTORÍA Y ORIGINALIDAD

(Este documento debe acompañar al Trabajo Fin de Grado (TFG)/Trabajo Fin de Máster (TFM) cuando sea depositado para su evaluación).

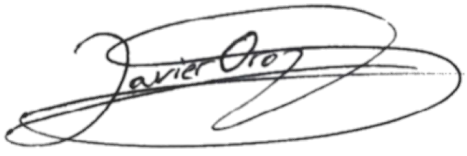
D./Dª. Javier Oroz Joven,

con nº de DNI 73158551-J en aplicación de lo dispuesto en el art. 14 (Derechos de autor) del Acuerdo de 11 de septiembre de 2014, del Consejo de Gobierno, por el que se aprueba el Reglamento de los TFG y TFM de la Universidad de Zaragoza,

Declaro que el presente Trabajo de Fin de (Grado/Máster) Grado, (Título del Trabajo) Diseño e implementación de un método para evitar colisiones en un sistema multi-robot,

es de mi autoría y es original, no habiéndose utilizado fuente sin ser citada debidamente.

Zaragoza, 23 de noviembre de 2016



Fdo: Javier Oroz Joven

RESUMEN

DISEÑO E IMPLEMENTACIÓN DE UN MÉTODO PARA EVITAR COLISIONES EN UN SISTEMA MULTI-ROBOT

Este trabajo fin de grado se centra en el estudio del comportamiento y modelado de un sistema compuesto por varios robots que tienen que seguir cada uno una trayectoria de un conjunto de trayectorias posibles calculadas previamente utilizando algoritmos de planificación (por ejemplo utilizando descomposición en celdas (*cell decomposition* en inglés). En este trabajo no se va a ahondar en los métodos de cálculo de estas trayectorias sino en la implementación de un método para evitar los problemas inherentes en los sistemas multi-robot: las colisiones entre los distintos robots y los bloqueos del sistema que hace que algunos robots no acaben las trayectorias que han empezado.

Partiendo de un conjunto de trayectorias prefijadas para cada robot, se modelará el sistema mediante el uso de las Redes de Petri, que no es más que una representación matemática con una representación gráfica de un sistema de eventos discretos. En concreto se va a usar un subconjunto de este tipo de redes, las llamadas redes S4PR. El entorno en el cual se mueven los robots se considera particionado en regiones y cada región del mapa se modela como un lugar en la red de Petri. Para evitar colisiones, se establecen regiones con capacidad finita (lugares de recurso), es decir, no pueden pasar por ellas más de un robot al mismo tiempo, de modo que si un segundo robot quiere pasar por esa misma región deberá esperar. Así que habrá que añadir lugares de espera, pero estos lugares de espera pueden conducir a bloqueos en la red, ocasionando que los robots no lleguen a su destino. Estos bloqueos se pueden caracterizar en este tipo de red de Petri utilizando algunos elementos estructurales de la red que se llaman *sifones* o *cerrojos*. Controlando que estos elementos no se vacíen, la red de Petri no se bloquea.

Por tanto el objetivo principal y más laborioso de este trabajo es la implementación de un algoritmo para el cálculo de sifones en Redes de Petri S4PR, para así hallar los posibles estados de bloqueo. Existen otros métodos en la actualidad para su cálculo, pero no son tan potentes como el algoritmo que se va a desarrollar, en términos de tiempo de cálculo y necesidades de memoria.

Una vez hallados los sifones se añadirán lugares monitor que evitarán que se vacíen y que se produzcan bloqueos. De este modo los robots podrán alcanzar su destino sin colisionar con los obstáculos del mapa, sin colisionar con otros robots y sin que se produzcan bloqueos que no los permitan seguir con su trayectoria.

Por último, mencionar que este método de evitación de bloqueos que se va a desarrollar tiene muchas más aplicaciones. Cualquier sistema modelado con una red de Petri de la clase S4PR podría utilizar esta técnica aquí presentada. En la literatura hay ejemplos de sistemas flexibles de fabricación, sistemas de salud o sistemas de transporte modelados con redes de Petri pertenecientes a esta clase.

1 TABLA DE CONTENIDO

2 Introducción ..... - 2 -

2.1 Objetivo y Alcance ..... - 2 -

2.2 Fases del desarrollo ..... - 2 -

2.3 Herramientas utilizadas ..... - 3 -

3 Motivación y posibles aplicaciones ..... - 4 -

4 Modelado y evitación de colisiones del sistema multi-robot ..... - 8 -

4.1 Modelado de trayectorias ..... - 8 -

4.2 Colisiones, recursos compartidos y bloqueos ..... - 10 -

4.3 Prevención de Bloqueos basada en el cómputo de Sifones ..... - 12 -

5 Algoritmo de búsqueda de sifones ..... - 13 -

5.1 Cálculo de los sifones mínimos de la red S4PR y Grafo de poda ..... - 13 -

5.2 Breve descripción del programa ..... - 14 -

5.3 Ventajas de este algoritmo frente a otros ..... - 17 -

6 Prevención de bloqueos y control de sifones ..... - 18 -

6.1 Control de sifones malos mediante lugares de monitorización ..... - 18 -

6.2 Obtención de una nueva red ..... - 18 -

6.3 Función para el control de sifones en C++ y su uso en el programa ..... - 19 -

7 Metodología de Programación ..... - 21 -

8 Futuras aplicaciones ..... - 23 -

9 Conclusiones ..... - 23 -

10 Bibliografía y Referencias ..... - 25 -

Anexo 1. Fundamentos de Redes de Petri, Sifones y Vivacidad ..... - 28 -

Anexo 2: Algoritmo de cálculo de sifones para redes S4PR ..... - 32 -

Anexo 3: Descripción del programa, funciones y clases ..... - 34 -

A3.1. Funciones públicas en la clase PetriNetClass: ..... - 34 -

A3.2. Funciones públicas en la subclase S4PR: ..... - 35 -

A3.3. Funciones públicas de la clase Matrix<T> ..... - 37 -

Anexo 4: Instalación de Eclipse, Creación del proyecto e inclusión de librerías. ..... - 39 -

Anexo 5: Ejemplos de cálculo de sifones en diferentes redes S^4PR ..... - 43 -

A5.1. Ejemplo A ..... - 44 -

A5.2. Ejemplo B ..... - 45 -

A5.3. Ejemplo C ..... - 46 -

A5.4. Ejemplo D ..... - 48 -

2 INTRODUCCIÓN

2.1 OBJETIVO Y ALCANCE

El objetivo de este trabajo de fin de grado consiste en hallar un método mediante el cual se puedan evitar las colisiones en un sistema compuesto por un conjunto de robots móviles, los cuales han de seguir unas trayectorias calculadas previamente. Éstas se pueden obtener o bien mediante algoritmos que calculen la trayectoria de un solo robot en un entorno conocido y después aplicarlo a robots independientes en un entorno común, o bien mediante un algoritmo de planificación multi-robot donde los robots tienen que cumplir con una misión global y que tienen que cooperar para cumplirla. Una vez obtenidas las trayectorias, las colisiones se pueden evitar introduciendo modos de espera, es decir, deteniendo ciertos robots en ciertos lugares/regiones para que esperen a que pase otro. Sin embargo, estos modos de espera pueden producir bloqueos, por tanto el objetivo principal de este proyecto es diseñar un controlador supervisor que garantice la evolución correcta evitando bloqueos.

El alcance del proyecto es conseguir modelar el sistema y trayectorias de los robots mediante una abstracción matemática computable, como son las redes de Petri y a partir de ahí, implementar una librería en lenguaje C++, la cual tome de entrada los datos abstraídos en dicha representación. Esta implementación también contendrá varios algoritmos que indiquen al usuario cuáles son los lugares que pueden inducir a bloqueos, así como otras funciones que eliminen estos bloqueos y devuelvan una nueva estructura del modelo en forma de red de Petri, ahora sí, sin bloqueos. Y por último, se pretende traducir este nuevo modelo de estados finitos obtenido al sistema multi-robot real.

2.2 FASES DEL DESARROLLO

A lo largo del desarrollo de este trabajo, donde más nos vamos a centrar es en el problema de cómo evitar las colisiones de un sistema multi-robot, y en cómo resolver este problema pasando por distintas fases de razonamiento que se van a explicar a continuación:

- La primera parte de este trabajo comienza con una pequeña motivación en la cual se detalla el porqué de la necesidad de desarrollar esta aplicación y su posible trascendencia.

- A continuación se explica la metodología y herramientas utilizadas para resolver este problema, y esto consiste en la abstracción de las trayectorias de los robots a una red de Petri (S4PR), lo cual es similar a la abstracción de los sistemas de fabricación que se basan en sistemas de asignación de recursos por cada proceso. Esta abstracción consiste en dos pasos:

    1. La secuencia de las regiones que deben ser recorridas por un robot a lo largo de su trayectoria se modela como una máquina de estados, donde cada lugar de la red de Petri modela la presencia del robot en una región determinada.

    2. La capacidad de las regiones, es decir, el número de robots que pueden estar simultáneamente en un lugar, se modela mediante lugares de recurso, a los cuales se les asigna una marca (recurso), cuando el robot entra en una región específica y que se libera cuando el robot deja esa región.

- La última fase para culminar este razonamiento es implementar las soluciones para evitar las colisiones y bloqueos en la red. Esta fase consta de tres partes y para ellas ha servido de inspiración y base para este proyecto, el trabajo realizado y publicado en dos artículos realizados por varios profesores de la Escuela de Ingeniería y Arquitectura de Zaragoza.

    1. El primero de los artículos se titula "Petri net approach for deadlock and collision avoidance in robot planning" [1] de Marius Kloetzer, Cristian Mahulea y José Manuel Colom, y trata distintos procedimientos para abstraer las trayectorias de un sistema multi-robot en una red de Petri, y también explica técnicas para mantener la vivacidad de la red de Petri, evitando bloqueos y colocando lugares monitor en la red que impida que se vacíen los sifones de la misma (ver definición de sifón en *Anexo 1*).

    2. Otro artículo comentado es el escrito por Elia Esther Cano, Carlos A. Rovetto y José Manuel Colom, con título, "An algorithm to compute the minimal siphons in S4PR nets" [2] (basado en la tesis doctoral de E.Cano [3]). En este último artículo se explica detalladamente varios conceptos sobre las redes de Petri S4PR, sus características, los sifones y el algoritmo con el que se pueden hallar todos los sifones mínimos de una de estas redes. Dicho algoritmo es uno de los más potentes existente en la literatura y es el que ha sido implementado en lenguaje C++ en este trabajo.

Uniendo el trabajo previo de estos autores se consigue crear un programa en C++, el cual será empleado posteriormente como librería para una futura implementación mayor, en futuros trabajos relacionados con la planificación de trayectorias de robots o para otros proyectos que utilicen redes de Petri y en los que se desee hallar los sifones de dicha red y controlar su vivacidad.

2.3 HERRAMIENTAS UTILIZADAS

Como este proyecto consta de partes muy distintas se han utilizado diversas herramientas.

Para la parte de la implementación del algoritmo se ha seleccionado C++ como lenguaje de programación, ya que así en un futuro se podrá juntar esta implementación en una sola librería junto a otros trabajos que se están desarrollando también en este lenguaje en el Departamento de Informática e Ingeniería de Sistemas. Como entorno de desarrollo se ha utilizado el *IDE Eclipse*, en el cual se ha podido programar el algoritmo y probar su correcto funcionamiento. Dentro de este programa ha habido que añadir una librería para trabajar con grafos, denominada *boost graph library [4]*.

Para modelar el sistema, y probar con múltiples ejemplos, se ha utilizado una herramienta para *Matlab*, llamada *Petri Net Toolbox* [5] y que permite dibujar de una forma sencilla las redes de Petri del sistema ejemplo a modelar, permitiendo también calcular ciertos parámetros interesantes de las mismas.

También se ha hecho uso de la herramienta *TimeNET* [6], la cual es también una aplicación para trabajar con redes de Petri y que incluye el cálculo de sifones. Esta herramienta se ha utilizado para validar los resultados obtenidos con el algoritmo desarrollado.

A su vez, este proyecto servirá como futura herramienta para otros alumnos que desarrollan trabajos de fin de grado relacionados con la planificación de sistemas multi-robot en el Departamento de Informática e Ingeniería de Sistemas de la EINA.

3 MOTIVACIÓN Y POSIBLES APLICACIONES

El propósito al que este trabajo quiere llegar es la mejora en el control de sistemas basados en modelos de redes de Petri, concretamente el tema de la planificación del movimiento y trayectorias de robots móviles, ya que continúa siendo hoy en día un problema con numerosas posibles soluciones. Las tareas del control de robots pueden variar desde el más simple sistema de un robot al cual se le impone que alcance una meta evitando colisionar con objetos, hasta el control de un sistema de robots múltiples. Obviamente a mayor número de robots, mayor complejidad tendrá el problema.

La importancia de este trabajo radica en que el método de cálculo y solución de bloqueos se basa en un algoritmo de búsqueda de sifones bastante novedoso y que todavía no se ha implementado hasta la fecha. La utilización de este método brinda una mayor rapidez en el tiempo de cálculo y con unos menores requerimientos de memoria que los otros algoritmos existentes.

Como ya se ha comentado la aplicación inmediata de los algoritmos implementados para este trabajo es la creación de una plataforma para el control de un sistema multi-robot con el objetivo de evitar colisiones entre los robots móviles como el de la *Figura 1*. Sin embargo, se puede aplicar a cualquier sistema que sea modelable mediante redes de Petri. Por ello este proyecto tiene trascendencia en todas las áreas del control mediante sistemas de eventos discretos.

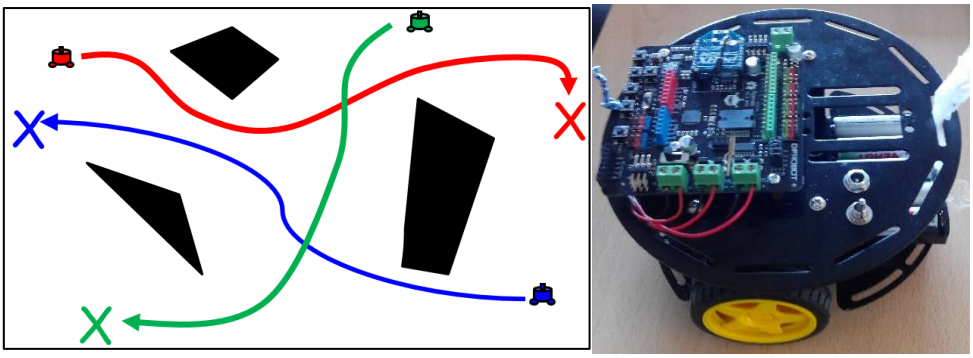


Figura 1. Ejemplo de mapa con obstáculos y tres robots junto a sus trayectorias (Izq).
Imagen de un robot móvil Arduino (dcha.)

Un ejemplo muy claro de otra aplicación distinta a la propuesta en este trabajo es el caso de los sistemas de producción que cada vez son más automatizados en la mayoría de los procesos de producción. Se tiende al uso de robots de transporte, cintas automáticas y brazos robóticos. Antes de su implementación de uno de estos procesos es muy importante modelarlo con un sistema discreto, comprobar su funcionamiento y corregirlo si hace falta. Las redes de Petri son una herramienta ampliamente utilizada cuando se trata de sistemas concurrentes, y como en cualquier otro sistema con recursos compartidos pueden aparecer bloqueos, así que este proyecto podrá servir para resolver estos bloqueos producidos por los sifones.

A continuación se desarrolla un ejemplo de un caso en el que se produce bloqueo, y se muestra como se procede a solucionarlo, siguiendo la metodología propuesta en este trabajo. Se considera un sistema informático bi-procesador equipado con dos unidades de disco D1 y D2 como

el esquematizado en la *Figura 2*. Cada uno de los dos procesadores P1 y P2 ejecutan una secuencia de tareas, cada tarea requiere la utilización de los discos. La secuencia de tareas realizadas por P1 se denomina como ST1, y la secuencia de las tareas realizadas por P2 se denomina ST2. La ejecución de la tarea correspondiente a ST1 requiere primero D1, y luego, manteniendo utilizado D1, se solicita la utilización de D2 y, por último D1 y D2 se liberan simultáneamente. La ejecución de la tarea ST2 requiere D2 primero, luego, manteniendo la asignación de D2, se solicita D1 y, por último D1 y D2 se liberan simultáneamente.

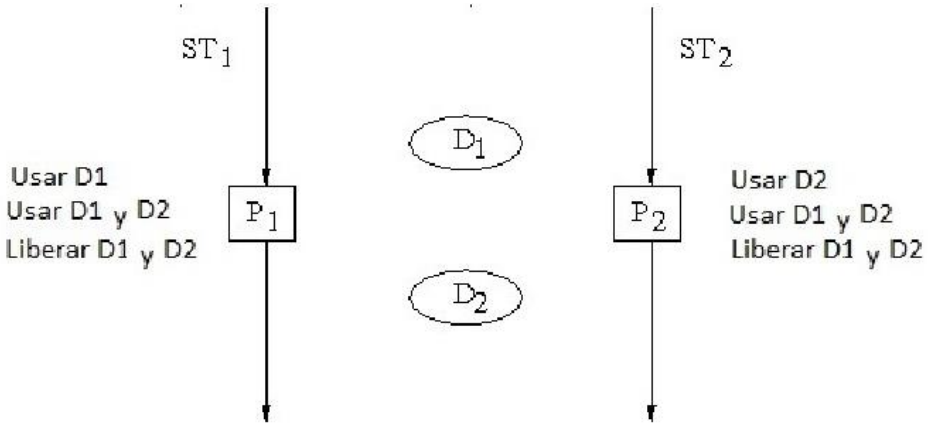


Figura 2. Representación esquemática del sistema informático de dos procesadores y dos discos.

Suponemos que no hay información a priori sobre la duración del tiempo que las tareas *ST1* o *ST2* ocupan *D1* y *D2*, así que no se tiene en cuenta y al momento inicial, los procesadores y los discos están en reposo.

A continuación en la *Figura 3.* se ha modelado el sistema, asignando a cada acción del sistema real una variable de estado en la red de Petri. Cada variable de estado se modela con un lugar y cada uno de estos lugares es una circunferencia. Los puntos negros son marcas que indican en qué momento de la ejecución del sistema se encuentra el modelo. La situación representada corresponde al estado de reposo con su marcado inicial, es decir, cuando los procesadores P1 y P2 no realizan ninguna acción y los discos D1 y D2 están libres sin ser usados.

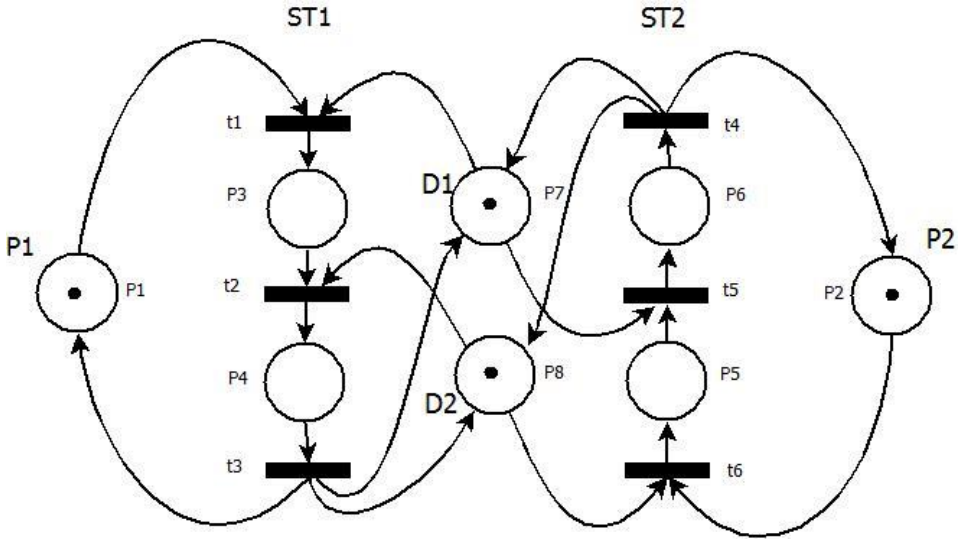


Figura 3. Red de Petri del sistema de dos procesadores y dos discos

Los lugares sin marcas inicialmente son los lugares de proceso, y representan las acciones que ha de realizar el sistema. Por ejemplo, el lugar P3 significa que el procesador P1 está realizando la tarea ST1 para la cual utiliza el disco D1. De este modo para que esto se realice, necesita dispararse la transición t1, la cual no lo hará hasta que haya una marca en P1 y una marca en D1, indicando que tanto el procesador P1 como el disco D1 están libres. Los lugares D1 y D2 son recursos compartidos ya que sus marcas son necesarias para realizar varias acciones, de manera que si uno de estos lugares de recurso no está marcado es porque está siendo usado por un proceso, y hasta que la transición de salida del lugar correspondiente al proceso, no libere la marca, esta no volverá de nuevo al recurso compartido para disparar la transición de entrada del lugar del otro proceso que necesita la marca.

Una vez modelado el sistema a controlar, lo siguiente es analizar su vivacidad, es decir, comprobar si puede haber bloqueos parciales o totales, en este caso al tratarse de un sistema con dos procesos concurrentes que requieren de dos recursos compartidos (D1 y D2), hay un sifón malo, éste se puede calcular mediante la herramienta desarrollada en este trabajo. Como se explica más adelante en la *Sección 4.3.*, estos sifones son un conjunto de lugares que si se quedan sin marcas durante la evolución de la red, no vuelven a recibir marcas, provocando un bloqueo. En la imagen se muestran en rojo los lugares por los que está compuesto el sifón malo (P4, P6, P7 y P8) y la situación del marcado cuando se produce el bloqueo.

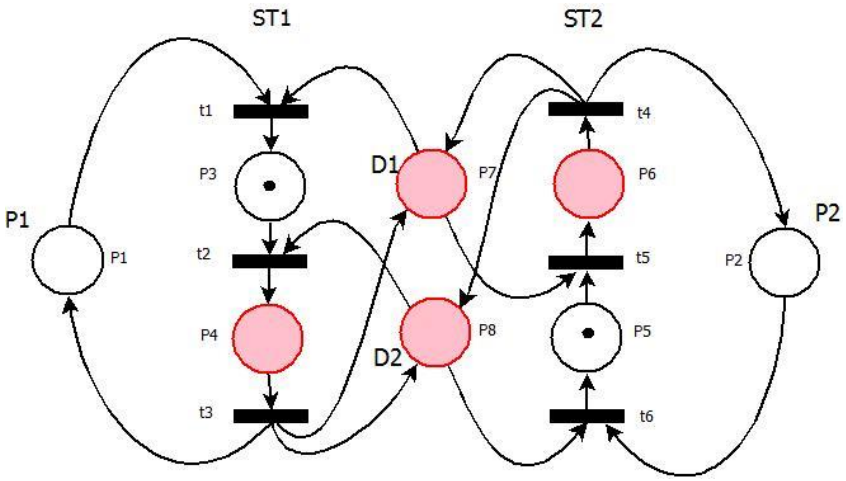


Figura 4. Red de Petri del sistema bi-procesador en situación de bloqueo. Lugares del sifón malo remarcados en rojo.

Además también hay que tener en cuenta que los lugares de los dos procesos, ST1 (P1, P3, P4) y ST2 (P2, P5, P6), forman un sifón cada uno, pero estamos ante sifones buenos, es decir, son lugares que cumplen la definición de sifón (ver *Anexo 1*), ya que el conjunto de transiciones de entrada está incluido en el conjunto de transiciones de salida, pero no producen bloqueos ya que nunca se van a vaciar de marcas. Esto se debe a que estos sifones buenos contienen a su vez una *trampa*, es decir un conjunto de lugares que no se vacía. Sin embargo, el sifón malo sí que puede llegar a producir bloqueos así que es el que va a ser objeto de estudio en este trabajo.

Tras aplicar el algoritmo de control de sifones desarrollado en este proyecto, se introduce un lugar monitor que soluciona el problema de un posible bloqueo de modo que la red de Petri quedaría de la siguiente manera, tal y como se muestra en la *Figura 5*. Este lugar monitor, pintado en verde, se encarga de impedir que se dispare la transición *t6* iniciándose ST2 cuando la tarea ST1 ya ha sido iniciada y de impedir que se dispare la transición *t1* comenzando ST1 cuando la tarea ST2 ya ha sido iniciada.

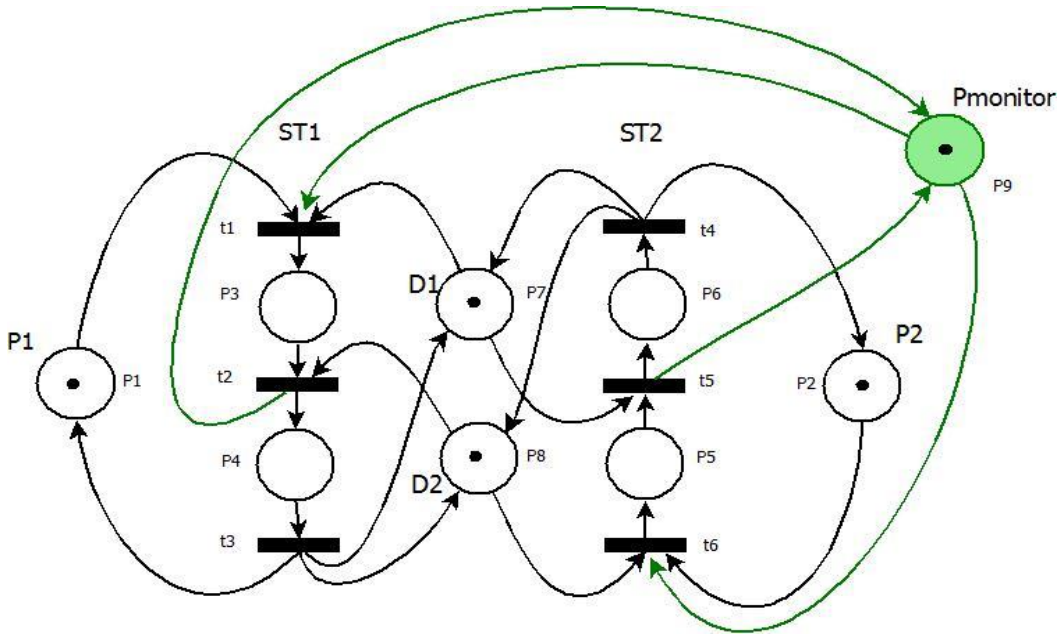


Figura 5.Red de Petri del sistema bi-procesador con los lugares de sifón con el lugar monitor en verde.

Así mediante este lugar monitor que controla la evolución del sistema, conseguimos evitar que los lugares P3 y P5 estén marcados al mismo tiempo, con el consiguiente bloqueo que ello conlleva.

Algún otro ejemplo sobre el que tendría aplicación este trabajo y en concreto los algoritmos que aquí se van a desarrollar, sería para gestión de proyectos mediante redes de Petri, en este caso los recursos compartidos serán los trabajadores que deban desarrollar cada etapa del proyecto, de modo que habrá que evitar que se produzcan sobreasignaciones de los trabajadores con sus consiguientes bloqueos en el sistema de gestión del proyecto y por tanto retrasos en la entrega del proyecto. Otra aplicación sería para el control y asignación de recursos y médicos en un hospital mediante redes de Petri. En este caso, los médicos, cirujanos y enfermeros serían los recursos compartidos y también existirían lugares con capacidades limitadas, como en el caso de los robots, que serán las habitaciones y quirófanos que sólo admitirán un número determinado de enfermos. Por tanto, estos lugares precisarán de recursos compartidos los cuales podrán inducir a bloqueos en el sistema de control, y en este momento es cuando juega un papel importante el programa de este trabajo para controlar los sifones. Esta aplicación en particular enfocada a guías clínicas se explica en detalle en [7]. Así se quiere ilustrar que con el programa desarrollado se pueden prevenir bloqueos en distintos tipos de sistemas, siempre y cuando sean modelables con sistemas de eventos discretos (redes de Petri pertenecientes a la clase S4PR) y una vez solucionados los bloqueos se podrá rediseñar el sistema de forma que éste evolucione de forma más segura. Y de siguiendo este proceso se puede llegar a aplicar a otros campos como: el análisis de datos, la optimización del flujo de trabajo en una empresa, el control de drones voladores que se muevan organizadamente evitando colisiones, etc.

4 MODELADO Y EVITACIÓN DE COLISIONES DEL SISTEMA MULTI-ROBOT

4.1 MODELADO DE TRAYECTORIAS

Antes de modelar las trayectorias se debe reconocer el mapa en el que se va a trabajar, es decir, el plano sobre el que se van a desplazar los robots. El tipo de plano sobre el que se va a centrar este trabajo y la planificación multi-robot consiste en un mapa en dos dimensiones, normalmente un cuadrilátero que va a contener una serie de obstáculos, es decir, áreas poligonales del plano por las cuales ningún robot puede pasar. No hace falta decir que los robots no deben salirse de los límites del plano.

El trabajo ya realizado por otro alumno (Jorge Barrio [8]) permite posicionar a un conjunto de robots Arduino en un mapa mediante el reconocimiento de imágenes realizado mediante una cámara situada sobre el plano y situar también los obstáculos y límites del terreno en el que se desenvolverán los robots. De modo que se obtendría un mapa similar al siguiente: *Figura 6.*

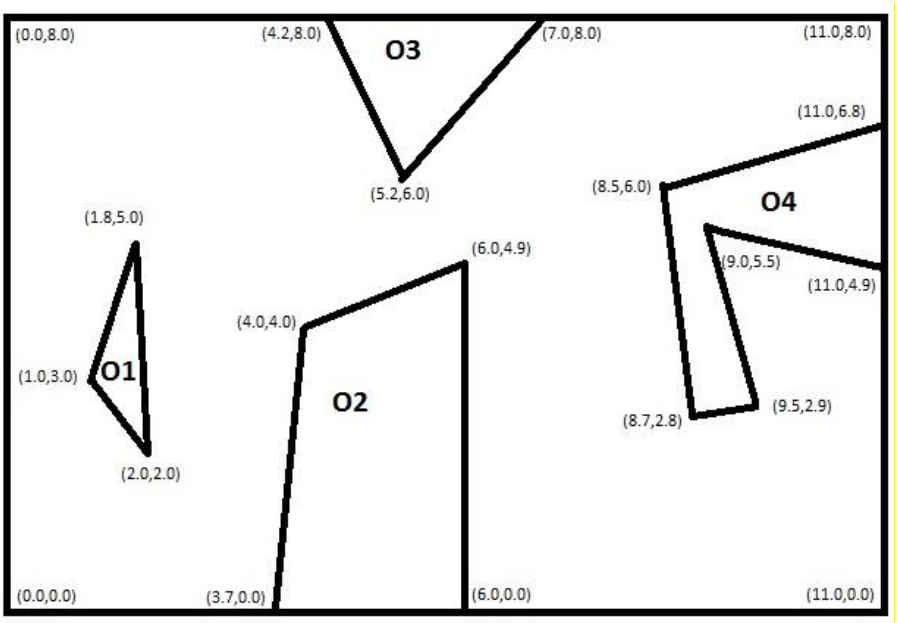


Figura 6. Mapa con obstáculos y las coordenadas de cada vértice.

El siguiente paso a realizar para obtener el modelo discreto del sistema multi-robot es dividir el mapa en regiones, ya que cada región del mapa se modelará como un lugar en el modelo de la red de Petri (RdP) o como un nodo en el modelo de un autómata finito determinista. Para resolver el problema mencionado de dividir el mapa en regiones, se utiliza el método de la *Descomposición en Celdas (Cell Decomposition)* [9]. Este método consiste en dividir las zonas sin obstáculos del plano mediante polígonos. Existen distintas técnicas como las que se muestran en la *Figura 7:*

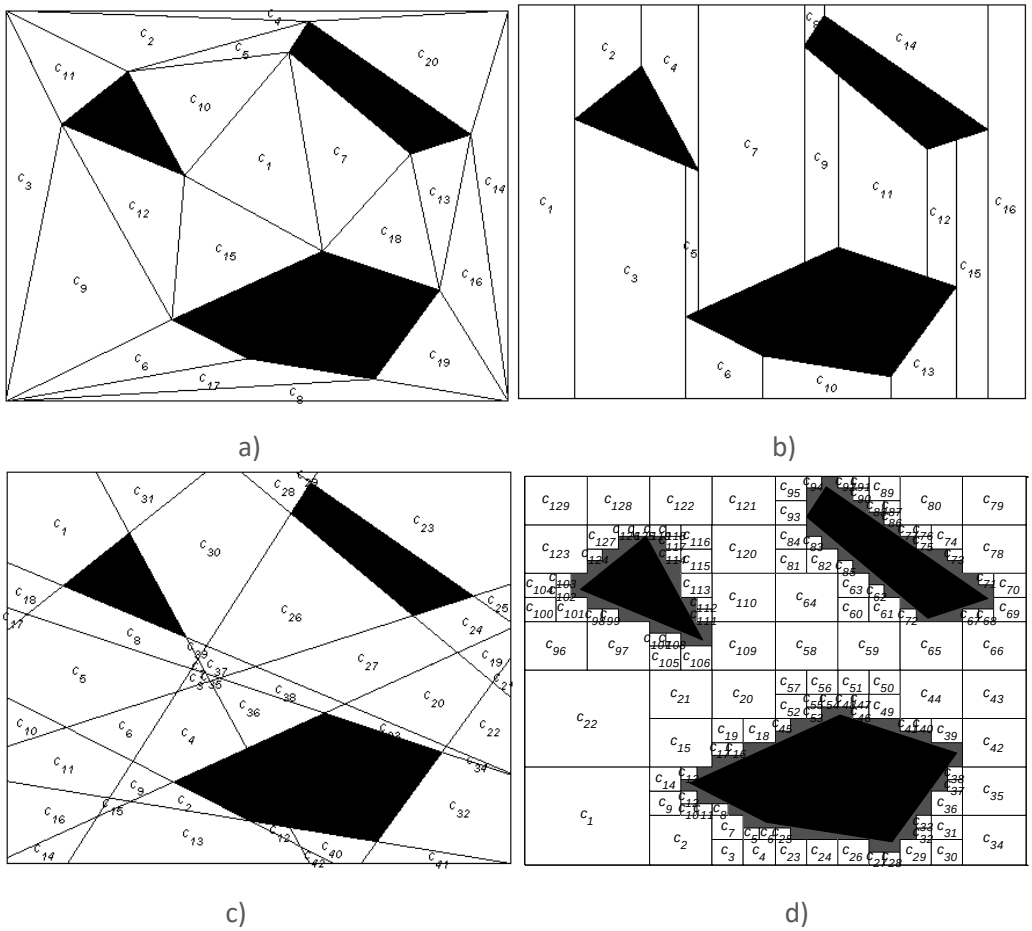


Figura 7. Descomposición en celdas: a) triangular, b) trapezoidal, c) politopal y d) rectangular

Una vez descompuesto el mapa en regiones, queda calcular las trayectorias de cada robot para saber porque regiones ha de pasar para alcanzar de la forma más óptima el objetivo prefijado. Las trayectorias son obtenidas de forma automática a partir de algoritmos de cálculo de trayectorias individualmente para cada uno de los robots ignorando al resto de ellos utilizando algoritmos de búsqueda de caminos mínimos en grafos [10], o también pueden calcularse mediante algoritmos de planificación multi-robot ignorando las posibles colisiones entre los mismos utilizando programación matemática y modelos de tipo redes de Petri [11][12].

Y por último, conociéndose las trayectorias de los robots, se obtiene una red de Petri S4PR y se diseña un controlador para evitar las colisiones entre los robots. El controlador se obtiene utilizando otro tipo de redes de Petri que modela solo las posibilidades de movimiento de las trayectorias calculadas. Además, para evitar las colisiones se introduce capacidad unitaria en los lugares por donde pasan más de un robot.

Para facilitar el entendimiento de este modelado, vamos a considerar un entorno plano como el de la *Figura 8,* en la cual se muestra un ejemplo (obtenido del *artículo* [1]) e ilustra cómo podría ser un mapa con obstáculos (regiones negras), regiones con capacidad finita (las de color azul/gris que tienen capacidad limitada de modo que no puede haber más de un robot en una de esas regiones al mismo tiempo) y regiones sin restricciones de capacidad. El espacio se divide en 20 regiones triangulares a partir de los vértices de los obstáculos. Asumiendo que hay dos robots, inicialmente situados en las regiones q1 y q2 respectivamente, la misión requiere que los robots muevan el triángulo desde q20 a q7, y el cuadrado desde q14 a q15. Para lograr esta misión, mediante los algoritmos de cálculo de trayectorias, se obtiene una lista de posibles trayectorias para

cada robot. Como ya se ha comentado antes, la construcción de las trayectorias de cada robot no es el objetivo de este trabajo. Así que se toma como ejemplo una sola posible trayectoria para cada robot, de la manera siguiente:

- El robot R1, es decir, el que debe mover el triángulo, va por las regiones: q1, q6, q11, q17, q5, q8, q3, q15, q13, q20, q13, q15, q3, q19, q7.
- El robot R2, es decir, el que debe mover el cuadrado, va por las regiones: q2, q18, q16, q9, q5, q8, q3, q19, q7, q10, q14, q10, q7, q19, q3, q15.

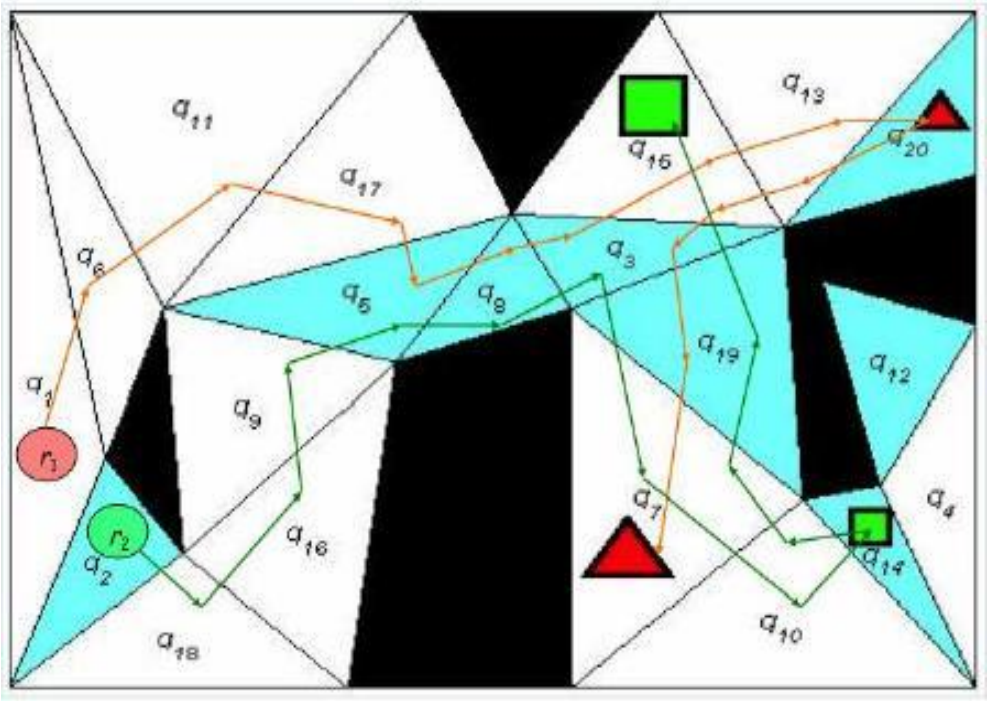


Figura 8. Ejemplo de mapa plano dividido en regiones por las que transitan dos robots. [1]

4.2 COLISIONES, RECURSOS COMPARTIDOS Y BLOQUEOS

Continuando con el ejemplo anterior, si los dos robots individualmente siguen sus propias trayectorias prefijadas, estos pueden colisionar en q5, q8, q3 o q19, ya que sus trayectorias pasan por esas regiones que tienen capacidad unitaria (un solo robot puede haber dentro de ellas en cualquier momento). Para evitar las posibles colisiones, se introducen modos de espera, es decir, se impone que por estas regiones no pueda pasar más de un robot al mismo tiempo, de modo que el segundo en llegar a esa región deberá esperar a que el primero salga de esa área.

Para abstraer este concepto de regiones con capacidad limitada a las redes de Petri, se introducen lugares adicionales, a los que se les llama lugares de capacidades o de recursos compartidos. Estos lugares contienen inicialmente tantas marcas como capacidad tenga la región, en este caso se asignarán recursos con una marca a los lugares que corresponden a regiones con capacidad unitaria. Esta marca se utiliza para disparar la transición de entrada al lugar con capacidad limitada, y cuando el robot abandona esa zona, la transición de salida de ese lugar libera la marca para que vuelva y pueda ser utilizada por el otro robot. Utilizando este concepto para todos los lugares con capacidad finita y siguiendo con el mismo ejemplo, su red de Petri quedaría como la de la *Figura 9.*

En la red de la imagen podemos identificar claramente dos procesos, uno para cada robot, en la parte superior el proceso relativo al robot R1, el cual modela la trayectoria que ha de seguir con

todos los lugares por los que debe pasar R1 y en la parte inferior los lugares relativos a R2. Cada lugar dentro de los procesos es una variable de estado y la presencia de una marca en un lugar significa que el robot Ri se encuentra en ese estado. Para denotar cada lugar de proceso se ha indicado primero el robot y después la región del mapa en la que está, de modo que la presencia de una marca en el lugar R1.q19 significa que el robot R1 está en la región q19. Los lugares R1.1 y R2.1 son los lugares de reposo o de inicio de los robots, por eso contienen una marca cada uno al inicio.

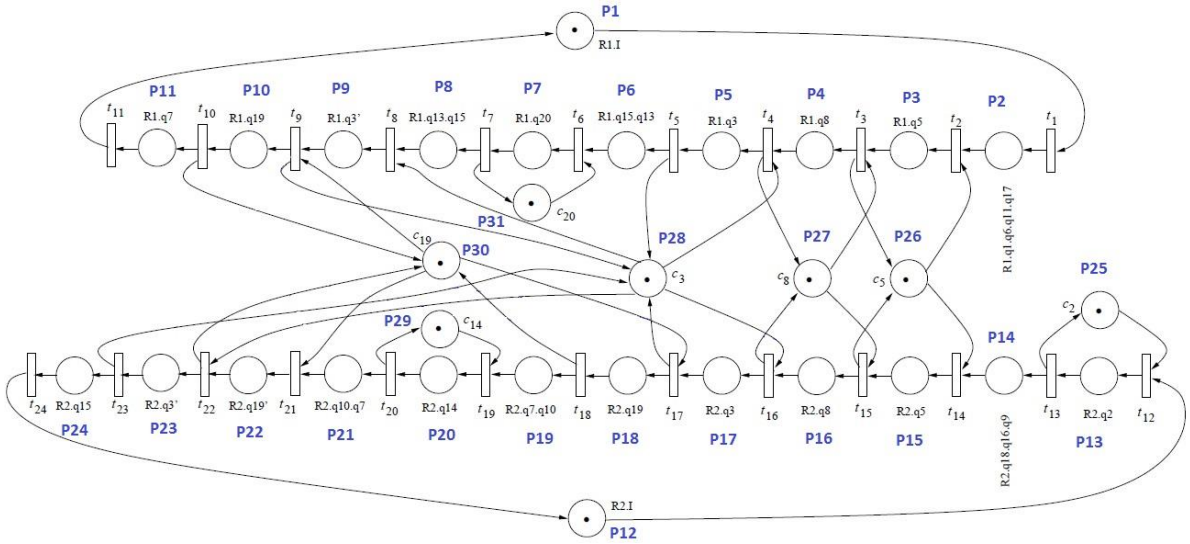


Figura 9. Red de Petri del modelo de 2 robots siguiendo las trayectorias del mapa de la Figura 8.

Cabe destacar, que conforme va evolucionando el sistema y los robots se van moviendo de una región a otra, el marcado de la red va cambiando hasta que finalmente cuando un robot termina su trayectoria, la marca vuelve al lugar de reposo, sin embargo, el modelo de red de Petri no podría volver a ser usado para supervisar el movimiento hasta que el robot sea colocado de nuevo en la posición de inicio del modelo.

Por otro lado los lugares denotados con c2, c3, c5, c8, c14, c19 y c20 son los recursos compartidos antes comentados y que indican la capacidad limitada que tienen las regiones q2, q3, q5, q8, q14, q19 y q20 respectivamente. Al resto de lugares se les asigna capacidad igual al número de robots, así que por eso no se les añaden recursos compartidos, pero en realidad estos lugares que representamos sin recursos compartidos asociados, sí que tienen capacidad limitada e igual al número de robots, así que sería como colocar un recursos virtual con capacidad 2 en este caso (2 robots) y asignar el recurso virtual a cada región con capacidad no limitada.

Por poner un ejemplo de cómo estos lugares de recurso funcionan, cuando ambos robots viajan hacia q5, el robot R1 llega primero y entra en esta región restringida. Entonces se elimina la marca del lugar correspondiente a su capacidad (c5) disparando la transición t2. Por lo tanto, si R2 llega a q5 mientras R1 está todavía allí, este no puede entrar debido a que la transición t14 no se puede disparar hasta que la marca sea devuelta a c5 cuando R1 salga de q5. Sin embargo en el caso del lugar R2.q18q16q9, no se ha dibujado ningún recurso y eso significa que el robot R2 podrá pasar por las regiones q18, q16 y q19 sin problema ya que por estas regiones caben los dos robots.

Esta simple idea, lo que hace es generar modos de espera de modo que nunca puedan coincidir en una región con capacidad restringida más de un robot. De modo que escogiendo correctamente estas regiones en las zonas donde las trayectorias de los robots coinciden, se pueden

evitar las colisiones entre robots. Esto es debido a que los recursos compartidos producen modos de espera.

Pero el gran problema es que incluso si se consiguen evitar las colisiones introduciendo modos de espera como se ha explicado, se pueden producir bloqueos (*deadlocks* en inglés).

Se entiende como bloqueo total o *deadlock* toda aquella situación que surge durante la evolución del sistema en la red de Petri y que produce cuando ninguna transición se puede disparar debido a la falta de marcas en alguno de sus lugares de entrada causando así una detención en la evolución de la red. Esto se produce en redes no vivas (ver definición de vivacidad de una red de Petri en *Anexo 1*).

Volviendo al ejemplo de los dos robots y las trayectorias de la *Figura 8* vemos que se puede producir una situación de bloqueo. Esto ocurre cuando el robot R1 viene de la región *q15* y entra en la región *q3* tomando así una marca del recurso compartido que en este caso es la capacidad unitaria *c3* de la región *q3*, y al mismo tiempo el robot R2 viene de la región *q7* y entra en la región *q19* eliminando la marca del lugar *c19*. Una vez alcanzada esta situación, los robots no se moverán a ninguna otra región y no completando sus trayectorias asignadas, produciéndose un bloqueo. Es decir, el robot R1 quiere ir desde *q3* a *q19* pero necesita que *q19* esté libre (necesita una marca en *c19*, pero está siendo usada por R2) y por otro lado el robot R2 quiere ir desde *q19* a *q3* pero necesita que *q3* esté libre (necesita una marca en *c3* la cual está siendo usada por R1).

Para solucionar estos bloqueos hay que controlar ciertos elementos estructurales de la red y que veremos a continuación en la *Sección 4.3*. Para un mejor entendimiento, siguiendo con el caso del ejemplo de la *Figura 9* para solucionar el problema de los posibles bloqueos, se debe controlar estos sifones y para ello *s*e introduce un lugar adicional (denotado *ct* y que aparece en la *Figura 14*) que actúa como un recurso compartido auxiliar. Estos lugares adicionales que sirven para controlar los sifones son conocidos como lugares monitor que se explicarán más en detalle en la *Sección 6.1*.

4.3 Prevención de Bloqueos basada en el cómputo de Sifones

Los *sifones* en una red de Petri son elementos estructurales que están íntimamente relacionados con la vivacidad de la misma, es decir, según [2], en redes S4PR cualquier estado de bloqueo está asociado con al menos un sifón mínimo vacío, el problema es que encontrar todos es una tarea laboriosa. De acuerdo con la definición formal de sifón (ver en *Anexo 1*), si los lugares pertenecientes a un sifón se vacían en un momento dado, es decir, se quedan sin marcas durante la evolución, no es posible marcarlos de nuevo, ya que el conjunto de transiciones de entrada está incluido en el conjunto de transiciones de salida.

Paralelamente se debe tener en cuenta el concepto de *trampa*, el cual se trata también de un componente estructural de las redes de Petri. En este caso, un conjunto de lugares pertenece a una trampa si tras tener inicialmente al menos una marca, nunca llegan a quedarse sin marcas, esto se debe a que el conjunto de transiciones de salida de estos lugares está incluido en el conjunto de transiciones de entrada. (Ver definición formal de trampa en *Anexo 1*)

De modo que uno de los métodos más utilizados para asegurar la vivacidad de una red S4PR radica en encontrar el conjunto de sifones que pueden producir bloqueos, es decir, los sifones que no contienen ninguna trampa inicialmente marcada. Es importante mencionar que no todos los sifones mínimos de la red inducen a bloqueos. Por ejemplo, el conjunto de todos los lugares de un proceso, por definición (ver en *Anexo 1*) forman un sifón, y por tanto habrá un sifón con los mismos lugares que cada proceso correspondiente a cada robot. Sin embargo, estos sifones no se pueden

vaciar siendo a su vez trampas. Como siempre va a haber una marca dentro de un proceso, estos sifones no se pueden vaciar conteniendo una trampa marcada. De este modo, a este tipo de sifones se los conoce como sifones buenos y al resto, es decir, a los que pueden vaciarse y producir bloqueos, se les conoce como sifones malos.

Según [13][14], los sifones "malos" están relacionados con cierto marcado en donde los procesos activos se encuentran bloqueados debido a que sus transiciones de salida están muertas porque los recursos que necesitan no se encuentran disponibles y además tampoco se encontrarán disponibles en el futuro.

En la siguiente sección de este trabajo se va a explicar el desarrollo de un algoritmo para el cálculo de sifones malos, ya que son los que nos interesan en este estudio.

5 ALGORITMO DE BÚSQUEDA DE SIFONES

5.1 CÁLCULO DE LOS SIFONES MÍNIMOS DE LA RED S4PR Y GRAFO DE PODA

Existen tres tipos de enfoques a la hora de desarrollar algoritmos que calculen el conjunto de todos los sifones mínimos en redes S4PR: utilizando aproximaciones algebraicas [15], ecuaciones lógicas [16][17] y el tercero está basado en el Grafo de Poda [2],[3]. Pero si se quiere controlar los sifones para evitar bloqueos, estos algoritmos hay que aplicarlos de forma iterativa, ya que una vez se han encontrado los sifones e introducido los lugares monitor, se ha de volver a aplicar el algoritmo para buscar nuevos sifones. Por lo cual el proceso iterativo aumenta la complejidad del asunto.

Este trabajo se centra en la implementación de un algoritmo de búsqueda de sifones para redes S4PR mediante el método del Grafo de Poda. Este nuevo algoritmo ha sido desarrollado por *E. Esther Cano et al.* en *"An algorithm to compute the minimal siphons in S4PR nets" [2],[3]* y la diferencia de este método con respecto a otros, es que trabaja con objetos de mayor nivel que simples lugares y transiciones, permitiendo una mayor eficiencia en términos de ocupación de memoria, al menos durante los pasos intermedios. En la *Sección 5.3* se explica por qué es más eficiente que otros algoritmos existentes.

Los objetos de alto nivel en que se basa este algoritmo son los sifones mínimos de la red que contienen tan sólo un lugar de recurso. Al conjunto de estos sifones se le llama $\mathcal{D}^1$. En la definición de la clase S4PR (ver en *Anexo 1*) se dice que se requiere un P-semiflujo mínimo por cada recurso, de modo que así se garantiza la existencia de uno de esos sifones mínimos por cada recurso de la red. Además para cada subconjunto de recursos existe, como mucho, un sifón mínimo que contiene ese subconjunto de recursos. Por lo tanto, este algoritmo construye cada sifón mínimo mediante la unión de los sifones mínimos con un recurso. El resultado de esta unión es otro sifón pero no es mínimo por lo general. Por esta razón se introduce el concepto de la relación de poda sobre el conjunto de sifones mínimos con un recurso usado para construir el nuevo sifón, con el objetivo de eliminar los lugares no esenciales de cada sifón mínimo con un recurso (del conjunto $\mathcal{D}^1$).

Para la representación de esta relación de poda se utiliza un grafo, el cual nos permite obtener los sifones mínimos mediante el cálculo de los subgrafos máximos fuertemente conexos del grafo. Se trata de un grafo dirigido o dígrafo (ver *Definición 10* del *Anexo 1*) y le llamaremos gafo de poda, sus vértices serán los lugares de recurso de la red, y sus arcos o uniones entre vértices se harán siguiendo la relación de poda. De modo que se dice que los sifones mínimos D_r y D_x están

relacionados por la relación de poda (o equivalentemente, que el sifón $D_r \in \mathcal{D}^1$ poda al sifón $D_x \in \mathcal{D}^1$, siendo los recursos $r \neq x$), si y sólo si los dos sifones comparten alguna transición y existen transiciones comunes con un lugar de proceso en la entrada perteneciente a D_x y también el recurso r entre en esa transición. (Ver definición de Grafo de Poda en *Anexo 1*)

Una vez obtenido el grafo de poda G, se debe crear una función de etiquetado de poda (*prunning labelling function,* K_G). Esta es una función que para cada nodo *r* del grafo *G* devuelve el conjunto de lugares candidatos a ser podados del sifón mínimo D_r por cada sifón mínimo D_x. Es decir, serán podados aquellos lugares entre los que exista un arco *(x,r)* en el grafo *G*. (Ver desarrollo de esta función en el *Algoritmo 1* del *Anexo 2).*

Por último se va a explicar la estructura principal del algoritmo de búsqueda de sifones. Este algoritmo va a encontrar de forma iterativa un nuevo sifón mínimo *D* de la red. La idea básica es construir en cada ciclo un nuevo sifón mínimo *D* mediante la unión de los sifones mínimos del conjunto $\mathcal{D}^1$ (conjunto de sifones mínimos de la red que contienen un solo recurso) correspondiente a los recursos contenidos en *D*. Y cada sifón *D* se disminuye según los lugares de poda obtenidos de la función $K_G(r)$ (ver algoritmo detallado en *Algoritmo 2* del *Anexo 2*).

De este modo tras implementar este algoritmo e introducirle una red S4PR como entrada, obtenemos un conjunto formado por los sifones de la red que contienen recursos. Los sifones formados por todos los lugares de proceso de cada proceso no los devuelve, ya que son obvios y una de las diferencias de este algoritmo de cálculo de sifones con respecto a otros es que se ha partido del conjunto de sifones $\mathcal{D}^1$ que contienen un recurso. Así que como los procesos no contienen recursos, no se obtendrán con este algoritmo, pero habrá que tenerlos en cuenta.

Tras obtener los *sifones mínimos de la red que contienen algún recurso*, se ha implementado otro algoritmo que a partir de estos sifones, descarte los que no sean malos, para así obtener tan solo el conjunto de sifones malos, que van a ser los que habrá que controlar para evitar que produzcan bloqueos. En la siguiente sección se explica cómo se ha desarrollado esta implementación.

5.2 BREVE DESCRIPCIÓN DEL PROGRAMA

Este algoritmo, se ha implementado en las funciones llamadas *getSiphons() y getBadSiphons()* en lenguaje C++. Pero junto a estas funciones ha habido que desarrollar muchas otras para poder trabajar con las redes de Petri y en concreto las S4PR. Para empezar, ha sido necesario utilizar una librería para el uso de matrices (archivos *matrix.cpp* y *matrix.h*), esta librería ya estaba parcialmente implementada, pero ha habido que programar ciertas funciones que se le han añadido. Además se ha desarrollado una librería llamada *setsFunctions.h* en la cual se han incluido ciertas funciones para trabajar con conjuntos de enteros y vectores de enteros (de tipo *set<int>* y *vector<int>*). Y la última librería que se ha implementado y sobre la que vamos prestar más atención es una para trabajar con redes de Petri, formada por los archivos *petri.cpp* y *petri.h*. En estos archivos se han incluido funciones para leer la red de Petri de un archivo *.txt*, funciones para calcular el conjunto de lugares de salida o de entrada de un conjunto de transiciones, funciones para calcular los P-semiflujos y T-semiflujos de una RdP, los sifones de una red S4PR, etc. (Todas las funciones se explican en detalle en el *Anexo 3* y en la *Sección 7* se muestra el diagrama de clases del programa para ver a que clase pertenece cada función).

Por otro lado tenemos el programa principal que es donde se deben incluir los nombres de las librerías para poder usar las funciones.

A continuación se muestra un ejemplo sencillo de cómo se debe estructurar el programa principal:

```
#include <iostream>
#include "petri.h"   //librería que debe estar en la carpeta del proyecto
                     //ya incluye el resto de librerías.
using namespace std;
int main(){
     FILE *fichero; char nfichero[50];
     cout << "Enter the name of the txt file: " << endl;
     scanf("%s",nfichero);
     strcat(nfichero,".txt");
     fichero = fopen(nfichero,"r");

     S4PR PetriNet;                    //Definimos clase Red de Petri S4PR
     PetriNet.read_from_file(fichero); //función para leer la red de petri
     fclose(fichero);

     if (PetriNet.isWellDefinedS4PR()){
            cout<<"This Petri net is a well-defined S4PR"<<endl;
     }else{
            cout<<"This Petri net is NOT a well-defined S4PR"<<endl;
     }
     PetriNet.printSiphons();
     PetriNet.printBadSiphons();

     return 0;
} //fin del main
```

En la primera parte del código se define una variable de tipo fichero y se pide por pantalla al usuario que introduzca el nombre del archivo txt en el que están contenidos los datos de la red de Petri. Después se define una variable de la clase S4PR y se leen los datos del fichero mediante la función *read_from_file()* y por último una vez almacenada la red de Petri se ejecutan varias funciones de la librería *petri.h*, en este caso vamos a usar *isWellDefinedS4PR()* para comprobar si la red de Petri leída es una red S4PR (comprobar si cumple las condiciones de red S4PR *del Anexo 1*) y a continuación se usa la función *printSiphons()* para imprimir los sifones de la red y *printBadSiphons() para imprimir los sifones malos de la red* ya que son nuestro objeto de estudio en este trabajo.

Alternativamente se puede usar la función *getSiphons()* que devuelve los sifones de la red y pueden ser almacenados en una variable del tipo *set< vector<int> >*. Esta función *getSiphons()* es la que va a contener el algoritmo del cálculo de sifones comentado en la sección anterior.

También existe una función llamada *getBadSiphons()* que devuelve los sifones malos de la red y los almacena en una variable del tipo set< vector<int> >. Esta función obtiene los sifones malos a partir de los sifones devueltos por la función *getSiphons()* y lo que hace es eliminar aquellos sifones que contienen lugares pertenecientes a un P-semiflujo (ver definición en *Anexo 1*), ya que si un P-semiflujo es subconjunto de un sifón, significa que ese sifón no se va a vaciar, ya que contiene una trampa y por tanto no es un sifón malo. (Ver explicación en detalle de cada función del programa en *Anexo 3*)

Antes de realizar el cálculo, en la *Figura 10* se muestra la red del ejemplo del sistema de dos robots móviles de la *Figura 8*, con sus lugares denotados con números ya que esta es la forma que va a trabajar el programa.

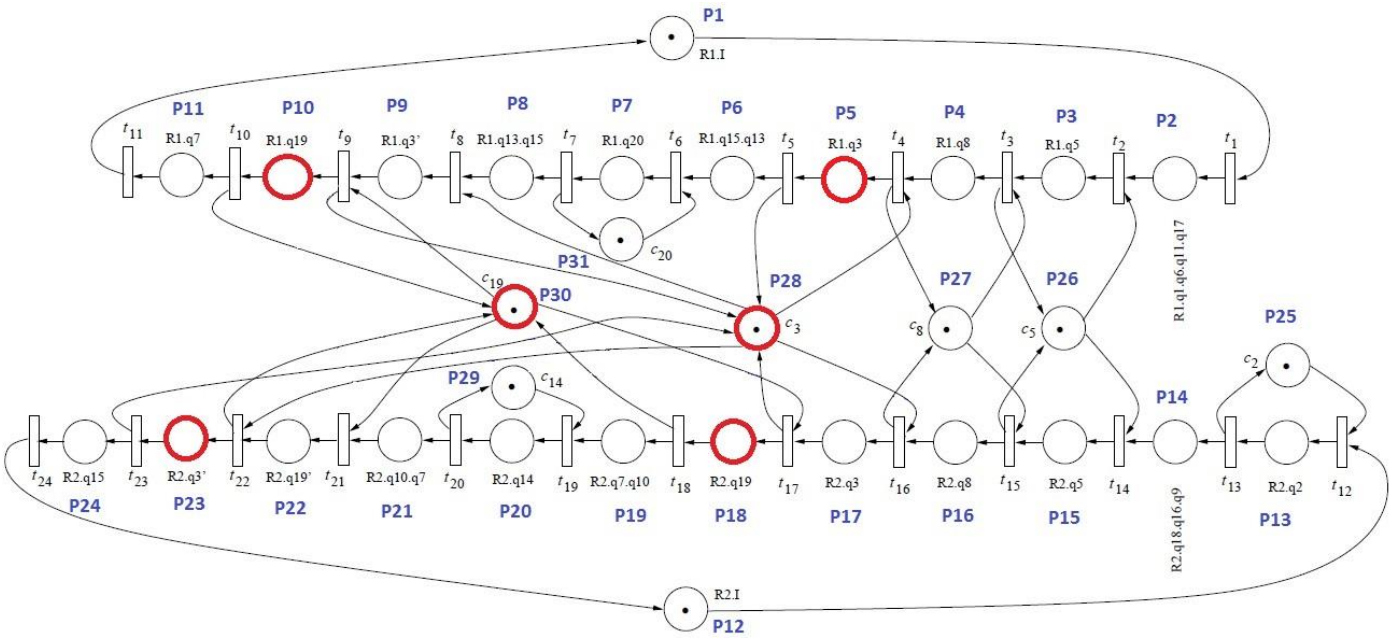


Figura 10. Red de Petri del sistema multi-robot de la Figura 8, con un sifón malo dado por sus lugares en rojo.

Una vez ejecutado el programa, los sifones calculados con la función *getSiphons()* son 8 y son los siguientes: {P3, P26, P15}, {P4, P16, P27}, {P5, P10, P18, P23, P28, P30}, {P5, P9, P17, P23, P28}, {P7, P31}, {P10, P18, P22, P30}, {P13, P25}, {P20, P29}. Estos son los sifones mínimos que contienen algún recurso, pero además sabemos que todos los lugares de los dos procesos formarán dos sifones: {P1, P2, P3, P4, P5, P6, P7, P8, P9, P10, P11}, {P12, P13, P14, P15, P16, P17, P18, P19, P20, P21, P22, P23, P24}. Pero sabemos que de estos 10 sifones, no todos son sifones malos, así que se ejecuta la función *getBadSiphons()* y se obtienen tan sólo 1 sifón malo, y es el señalado en rojo en la imagen anterior: {P5, P10, P18, P23, P28, P30}.

Para el correcto funcionamiento del programa, hay que crear un proyecto en el programa de entorno de desarrollo escogido, en este caso *IDE Eclipse*, en este proyecto se han de añadir los archivos necesarios para el programa (*controlSiphons.cpp, petri.cpp, petri.h, matrix.cpp, matrix.h y setFunctions.h*) y por último se ha de incluir la librería *boost graph*. (Ver cómo crear proyecto e incluir librerías en *Eclipse C++* en el *Anexo 4*).

Una vez preparado el programa, es necesario que este lea correctamente los datos de entrada, es decir la red de Petri de entrada debe introducirse de una forma muy concreta. Hay que introducir los datos característicos de la red en un fichero de texto ".txt", estos datos son, las matrices Post y Pre para formar la matriz de incidencias C de la red (ver definición en *Anexo 1*), el número de procesos, los lugares correspondientes a cada proceso, los lugares de recurso y el vector de marcado inicial. Un ejemplo de cómo debería escribirse un fichero de entrada se muestra en la *Figura 11.*

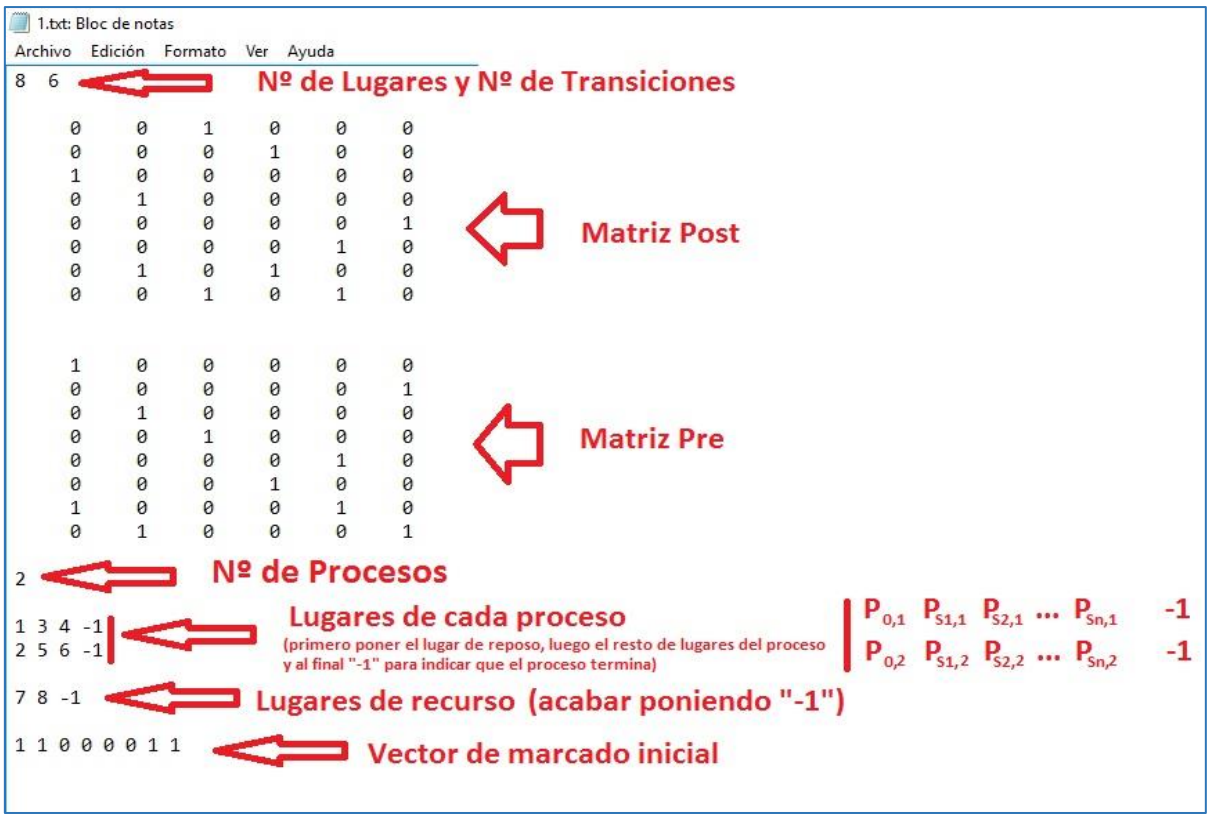


Figura 11. Fichero de entrada de la red de Petri del ejemplo de la Figura 3.

5.3 VENTAJAS DE ESTE ALGORITMO FRENTE A OTROS

La principal virtud de este algoritmo frente a los otros existentes es que calcula los sifones mínimos en un menor tiempo y requiere una menor cantidad de memoria ya que con este método se reduce el número de comparaciones que tiene que realizar el programa.

Esto se consigue gracias a que se toma el conjunto de sifones mínimos de la red con sólo un recurso (D^1) como dato de entrada para el algoritmo. El conjunto de sifones $\mathcal{D}^1$ es muy rápido de calcular a partir de los P-semiflujos correspondientes a los recursos. (Ver definición formal de P-semiflujo en *Anexo 1*). A pesar de ello, el número de sifones mínimos de una red S4PR con $|P_r|$ recursos puede ser igual a $2^{|P_r|} - 1$, (el número de subconjuntos de recursos no vacíos). Sin embargo, el número de sifones mínimos, en la práctica, es mucho menor a ese número. Esto significa que debemos evaluar muchas uniones no productivas de sifones mínimos de $\mathcal{D}^1$.

El único problema de este algoritmo es que tiene una mayor complejidad estructural a la hora de la implementación, ya que es necesario trabajar con estructuras de datos bastante complejas, como grafos, conjuntos de grafos, conjuntos de vectores, etc. Pero esto se ha solucionado al utilizar una *programación orientada a objetos* en C++ y definiendo una clase S4PR, dentro de la cual se han desarrollado todas las funciones necesarias y en las cuales se han utilizado variables de alto nivel de complejidad. Por poner un ejemplo, los grafos con los que trabaja la función *getSiphons()* son un conjunto de parejas en las cuales se almacena por un lado un vector que contiene las etiquetas de los vértices del grafo y por otro lado un grafo formado por una lista de adyacentes (*adjacency list*).

Este algoritmo sirve para redes de la clase S4PR como las que modelan sistemas multi-robot como el expuesto en este trabajo y obviamente para subclases de ésta, como las redes S3PR. Pero por supuesto para el problema de los bloqueos en redes S4PR, se pueden utilizar algoritmos de cálculo de sifones mínimos para redes de Petri generales. Algunos de estos algoritmos son los

desarrollados por: Barkaoui y Lemaire 1989 [18]; Chu y Xie 1997 [19]; Ezpeleta et al. 1993 [20]; Lautenbach 1987 [15]; Li y Zhou 2008 [21]; Wang et al. 2009 [22].

Para demostrar esta mayor rapidez del algoritmo del grafo de poda frente a otros existentes en el *Anexo 5* se muestran varios ejemplos de redes de Petri y sus sifones calculados mediante el programa desarrollado en este trabajo y los mismos sifones calculados mediante la aplicación TimeNET, que utiliza otro método distinto al propuesto para el cálculo de sifones.

6 PREVENCIÓN DE BLOQUEOS Y CONTROL DE SIFONES

6.1 CONTROL DE SIFONES MALOS MEDIANTE LUGARES DE MONITORIZACIÓN

Un método para asegurar la vivacidad de las redes de Petri ordinarias (ver definición de RdP ordinaria en *Anexo 1*) se basa en el control de los sifones malos de la red. Una vez se han obtenido los sifones de una red S4PR, el trabajo no culmina aquí pues ahora hay que solucionar los bloqueos que es el problema que los sifones malos generan. Para controlar los sifones se introduce un controlador supervisor implementado en una unidad central que asegura que estos sifones no se vacíen no dejando que la última marca del sifón abandone el conjunto de lugares del sifón. Este controlador consiste en introducir lugares monitor que se pueden considerar como lugares de recurso adicionales que se encargan de evitar que los sifones se queden sin marcas.

Los lugares que "roban" las marcas al conjunto de lugares del sifón, se les llama *lugares ladrones* (ver definición en *Anexo 1*). Estos lugares son los que se quedan bloqueados con las marcas del sifón cuando se produce la situación de bloqueo. Al introducir el lugar monitor, éste actúa como un recurso compartido para estos lugares ladrones, formando un P-semiflujo con ellos y que impide que estos lugares ladrones tengan en cualquier momento más de $x - 1$ marcas, siendo x el número de marcas inicialmente en el sifón.

6.2 OBTENCIÓN DE UNA NUEVA RED

Mediante una nueva función en el programa se obtiene una nueva red de Petri con estos lugares monitor incluidos en ella. Para obtener esta nueva red, conocemos los datos de la red S4PR que se desea controlar y los sifones malos de la misma calculados con la función *getSiphons()* descrita en la *Sección 5.* A partir de estos datos se obtiene una nueva red.

Siendo $S_k = \{p_1, \dots, p_n\}$ uno de los sifones malos y siendo p_k el lugar monitor que previene que S_k sea vaciado, se conecta el lugar monitor p_k de la siguiente manera:

1) Quedando la fila de la matriz de incidencias C correspondiente al lugar monitor p_k, tal que así:

$$C[p_k, t_j] = \sum_{l=1}^{n} C[p_l, t_j];\ \text{siendo}\ p_l \in S_k\ \text{y}\ t_j \in T$$

2) Y siendo el marcado inicial del lugar monitor p_k:

$$m_0[p_k] = \sum_{l=1}^{n} m_0[p_l]\ - 1;\ \text{siendo}\ p_l \in S_k.$$

Tras realizar estos cálculos obtendremos una fila más en la matriz C por cada sifón malo (correspondiente al lugar monitor) que exista y un elemento más en el vector del marcado inicial.

Por otro lado es evidente darse cuenta que estamos ante un proceso que hay que hacer de forma iterativa, pues una vez modificada la matriz C, obtenemos una red de Petri nueva, la cual puede tener nuevos sifones malos, de modo que hay que realizar iterativamente el proceso de

recalcular los sifones y añadir los lugares monitor hasta que finalmente no se obtengan sifones malos.

6.3 FUNCIÓN PARA EL CONTROL DE SIFONES EN C++ Y SU USO EN EL PROGRAMA

Este algoritmo se ha desarrollado dentro de una función en nuestro programa, a la cual se le ha llamado get*ControlledPetriNet()*. Esta función devuelve una nueva red de Petri de la clase S4PR a la cual se le han asignado las nuevas filas a la matriz C de incidencias y los nuevos elementos al vector del marcado inicial m_0 tal y como se han calculado en el algoritmo anterior.

Para utilizar esta función habrá que crear un programa principal parecido al de la *Sección 5.2* pero en lugar de utilizar la función *PetriNet.getSiphons()* se debe definir una nueva red de Petri y utilizar la función antes mencionada, por ejemplo:

```
S4PR newPetriNet; //Definimos clase Red de Petri S4PR
newPetriNet = PetriNet.getControlledPetriNet();
```

Siendo *PetriNet* una variable también de clase S4PR definida antes en el programa y en la que se ha almacenado la red de Petri de entrada. Tras obtener la nueva red podemos aplicar la siguiente función para imprimir los parámetros de la red y comprobar cuáles son los nuevos lugares monitor introducidos en la red. Estos los podemos ver tanto en la matriz de incidencias *C*, como en el conjunto de los lugares de recurso de la red P_r o directamente en el conjunto de lugares $P_{monitor}$.

```
newPetriNet.printPNparameters();
```

```
PARAMETERS OF THE PETRI NET:

Number of places: 32
Number of transitions: 24
Incidence matrix: C=
[-1,0,0,0,0,0,0,0,0,1,0,0,0,0,0,0,0,0,0,0,0,0;
1,-1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0;
0,1,-1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0;
0,0,1,-1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0;
0,0,0,1,-1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0;
0,0,0,0,1,-1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0;
0,0,0,0,0,1,-1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0;
0,0,0,0,0,0,1,-1,0,0,0,0,0,0,0,0,0,0,0,0,0,0;
0,0,0,0,0,0,0,1,-1,0,0,0,0,0,0,0,0,0,0,0,0,0;
0,0,0,0,0,0,0,0,1,-1,0,0,0,0,0,0,0,0,0,0,0,0;
0,0,0,0,0,0,0,0,0,1,-1,0,0,0,0,0,0,0,0,0,0,0;
0,0,0,0,0,0,0,0,0,0,-1,0,0,0,0,0,0,0,0,0,0,1;
0,0,0,0,0,0,0,0,0,0,1,-1,0,0,0,0,0,0,0,0,0,0;
0,0,0,0,0,0,0,0,0,0,0,1,-1,0,0,0,0,0,0,0,0,0;
0,0,0,0,0,0,0,0,0,0,0,0,1,-1,0,0,0,0,0,0,0,0;
0,0,0,0,0,0,0,0,0,0,0,0,0,1,-1,0,0,0,0,0,0,0;
0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,-1,0,0,0,0,0,0;
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,-1,0,0,0,0,0;
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,-1,0,0,0,0;
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,-1,0,0,0;
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,-1,0,0;
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,-1,0;
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,-1;
0,0,0,0,0,0,0,0,0,0,-1,1,0,0,0,0,0,0,0,0,0,0;
0,-1,1,0,0,0,0,0,0,0,0,0,-1,1,0,0,0,0,0,0,0,0;
0,0,-1,1,0,0,0,0,0,0,0,0,0,-1,1,0,0,0,0,0,0,0;
0,0,0,-1,1,0,0,-1,1,0,0,0,0,0,0,-1,1,0,0,0,0,-1,1,0;
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,-1,1,0,0,0;
0,0,0,0,0,0,0,0,-1,1,0,0,0,0,0,0,-1,1,0,0,-1,1,0,0;
0,0,0,0,0,-1,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0;
0,0,0,0,0,0,0,-1,1,0,0,0,0,0,0,-1,1,0,0,0,-1,1,0;
];
```

```
Initial marking: m0 =[1,0,0,0,0,0,0,0,0,0,0,1,0,0,0,0,0,0,0,0,0,0,0,1,1,1,1,1,1,1,1]
Number of Processes: 2
Idle Places: Po ={1,12}
Process Places: Ps ={2,3,4,5,6,7,8,9,10,11,13,14,15,16,17,18,19,20,21,22,23,24}
Resources places: Pr ={25,26,27,28,29,30,31,32}
Monitor places: Pm ={32}
_____________________________END_____________________________________
```

Figura 12. Salida del programa tras ejecutar la función getControlledPetriNet() y printPNparameters().

Siguiendo con el ejemplo de la *Figura 8* desarrollado durante todo este documento, ejecutando la función para imprimir los parámetros de la red de Petri tras aplicarse el control de sifones, se muestran los resultados que se obtienen en el programa (ver *Figura 12*).

Como se puede ver, ahora la red de Petri tiene 32 lugares en vez de 31, y también se ha añadido una fila más a la matriz de incidencias C para conectar este lugar monitor añadido con las transiciones que sean necesarias. También se ha modificado el vector de marcado inicial, añadiendo un 1 en el último lugar, ya que este lugar monitor (*P32* en este caso) tendrá una marca. Por último se ve que el número de procesos, los lugares de reposo y los de proceso, no cambian y son los mismos que antes de introducir el lugar monitor.

Trasladando estos datos obtenidos a nuestra representación gráfica, a continuación se muestra cómo quedaría la red de Petri (ver *Figura 13*) con el lugar de monitor introducido y correctamente conectado a las transiciones que nos indica la matriz C. Con este lugar monitor se consigue evitar que el sifón malo, formado por los lugares {P5, P10, P18, P23, P28, P30}, se quede sin marcas, impidiendo que las marcas de P28 y P30 lleguen al mismo tiempo a los lugares ladrones {P9, P17, P22} .

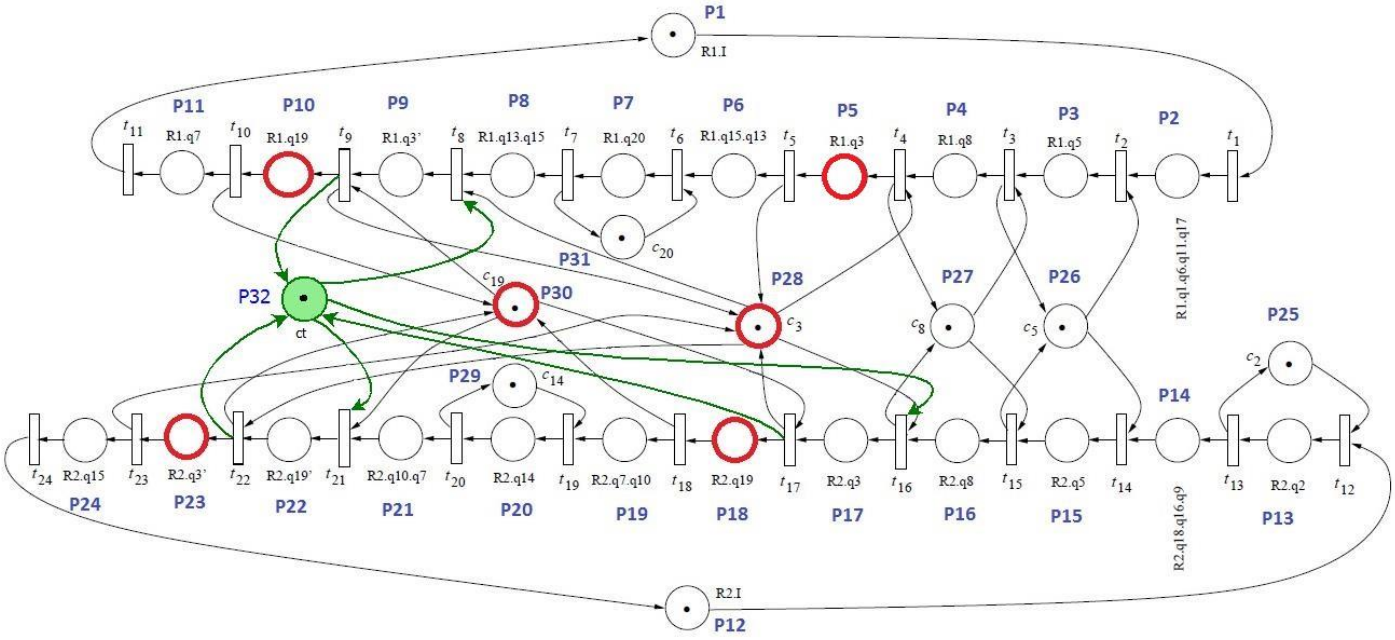


Figura 13. Red de Petri de la Figura 8. con lugar monitor añadido (en verde) que evita los bloqueos ocasionados por el sifón malo (lugares en rojo).

Como ya se ha comentado, este lugar adicional *ct* será integrado en una unidad central de control que supervisará el correcto desarrollo del sistema, evitando que los robots entren en ciertas regiones que ocasionen bloqueo. Es decir gracias al sistema de reconocimiento de imagen, un ordenador controlará en tiempo real en qué posición se encuentra cada robot y este controlador le

comunicará a los robots si pueden entrar en las regiones restringidas. La implementación será igual a la ya usada para el control de cualquier lugar de recurso.

7 METODOLOGÍA DE PROGRAMACIÓN

Para la realización de todas estas funciones antes comentadas en las cuales se han implementado los algoritmos de búsqueda de sifones y de control de sifones, ha sido necesario desarrollar un programa en lenguaje C++ el cual incluye muchas más funciones que se comentarán a continuación. Primero hay que destacar que el lenguaje escogido es C++ debido a que en este lenguaje se van a desarrollar el resto de aplicaciones para el proyecto global de la plataforma multi-robot que englobará los programas realizados por distintos alumnos, y todos estos serán implementados en C++.

Por otro lado independientemente del lenguaje escogido, se ha trabajado con una metodología de programación orientada a objetos. Este método de programación consiste en definir objetos o clases, a las cuales se les asignan propiedades, atributos y funciones, de modo que si se define una variable como una clase determinada, se pueden usar las funciones que tiene dicha clase sobre la variable. Esto permite programar de una forma más eficiente y legible para que otro programador entienda de manera más sencilla el código.

Como ya se ha mencionado brevemente en la *Sección 5.2*, para esta aplicación se han utilizado varias librerías:

1. Librería para trabajar con matrices (compuesta por los archivos *matrix.cpp* y *matrix.h*) y que contiene la clase *Matrix<T>* que permite crear variables de tipo matriz. Además contiene funciones para trabajar con ellas.
2. Librería para trabajar con redes de Petri (compuesta por los archivos *Petri.cpp* y *Petri.h*) y que contiene las clases *PetriNetClass* y *S4PR*. Estas dos clases contienen bastantes funciones para leer la red de entrada de un fichero, devolver su matriz de incidencias, calcular sus P-semiflujos, sus sifones, controlar su vivacidad mediante la eliminación de sifones, entre otras muchas más funciones necesarias para el correcto funcionamiento de estas anteriores.
3. Librería *boost* para trabajar con grafos (esta librería se puede descargar desde [4] y consta de varias carpetas y archivos y cuya ruta ha de ser añadida a la lista de rutas de librerías del entorno de programación, en este caso se ha usado *Eclipse* como IDE, ver explicación en *Anexo 4*). Esta librería permite crear grafos formados por vértices y arcos, calcular sus componentes fuertemente conexas y demás funciones usadas para implementar el algoritmo de cálculo de sifones mediante el método del Grafo de Poda usado en la función *getSiphons()*.

A continuación (ver *Figura 14*) se va a mostrar un diagrama de clases en el cual aparecen las clases creadas en las librerías (1.) y (2.), en el cual se pueden ver las correspondientes relaciones entre ellas y todos los atributos y funciones que tienen incluidas. (Cada función será explicada en detalle en el *Anexo 3*).

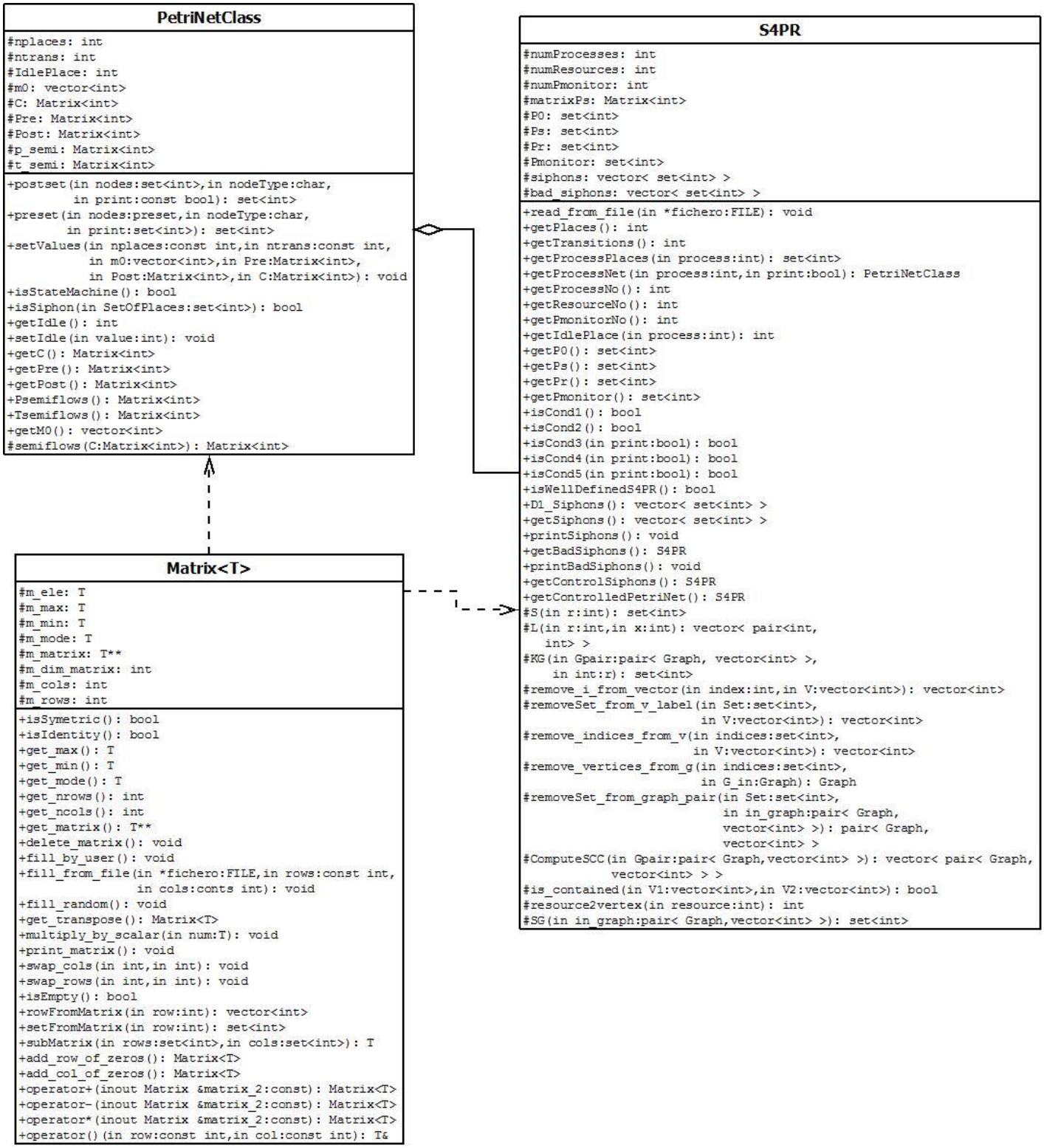


Figura 14. Diagrama de clases del programa de control de sifones en C++.

8 FUTURAS APLICACIONES

Una vez obtenida la red de Petri con el problema de los sifones y sus correspondientes bloqueos solucionados, queda diseñar un controlador que traslade la solución al sistema real en tiempo continuo. Como caso de estudio se considera una plataforma multi-robot del laboratorio L0.05a del Departamento de Informática e Ingeniería de Sistemas de la EINA.

La plataforma consiste en robots móviles basados en Arduino, de este modo sobre el microcontrolador de cada robot se cargará el programa que controle los movimientos que debe realizar cada robot para ejecutar las trayectorias deseadas. Para evitar las colisiones e implementar los lugares monitor hace falta una unidad central tal y como se ha descrito en la *Sección 6.1.* La plataforma será el resultante de la unión de todas las implementaciones hechas en otros trabajos de fin de grado realizados por otros alumnos.

El programa global de la plataforma se comunicará con la cámara situada sobre el plano por el que se mueven los robots sobre los cuales se colocarán cartulinas con códigos en forma de imagen que la cámara pueda enviar al ordenador para ser interpretados. El módulo de visión también reconocerá los límites o fronteras del mapa y los obstáculos colocados en él y traducirá en forma de coordenadas todos estos objetos reconocidos. El siguiente paso a hacer por el programa será descomponer en celdas el mapa según alguno de los métodos comentados y obtener un sistema discreto que modele el sistema. Después, habrá que situar a cada robot en su región inicial correspondiente, según las coordenadas. Posteriormente mediante los algoritmos de planificación de trayectorias se obtendrán las regiones por las que ha de pasar cada robot para llegar a su destino de la forma más eficiente. Se puede obtener un conjunto de trayectorias posibles para cada robot dejando a éste que elija una de estas trayectorias. Las trayectorias se pueden modelar con una red de Petri S4PR. Entonces es cuando interviene el algoritmo desarrollado en este trabajo obteniendo los sifones causantes de bloqueos del modelo y mejorando el modelo. Ahora ya es cuando se manda la correspondiente señal al controlador Arduino de cada robot para que se muevan según el *planning* establecido por el programa. Mientras estos se mueven el sistema de reconocimiento de imágenes mediante la cámara seguirá mandando información al programa de en qué posición se encuentra cada robot para controlar en bucle cerrado los movimientos de los robots.

9 CONCLUSIONES

Este trabajo aporta una solución al problema de la planificación y control de trayectorias de múltiples robots en un entorno compartido. Permitiendo así una correcta ejecución de los movimientos sin colisiones de ningún tipo ni bloqueos. Además estos posibles bloqueos se logran evitar mediante el cálculo de sifones utilizando un algoritmo más eficiente que cualquiera de los otros existentes hasta la fecha para este mismo fin.

Como se ha mostrado en los ejemplos del *Anexo 5* esta implementación del algoritmo del Grafo de Poda es mucho más eficiente en términos de uso de memoria. Sin embargo, en redes de Petri sencillas con pocos lugares resulta que es más rápida la herramienta TimeNET para el cálculo de sifones. También es verdad que con esta herramienta el tiempo de cálculo se incrementa de forma exponencial conforme se aumentan los lugares de la red de Petri. Sin embargo, utilizando la implementación del algoritmo del Grafo de Poda realizada en este trabajo, se aprecia cómo el

tiempo de cálculo no aumenta tan drásticamente con el incremento de los lugares de la red. Este comportamiento se puede ver en detalle en la *Figura 15*.

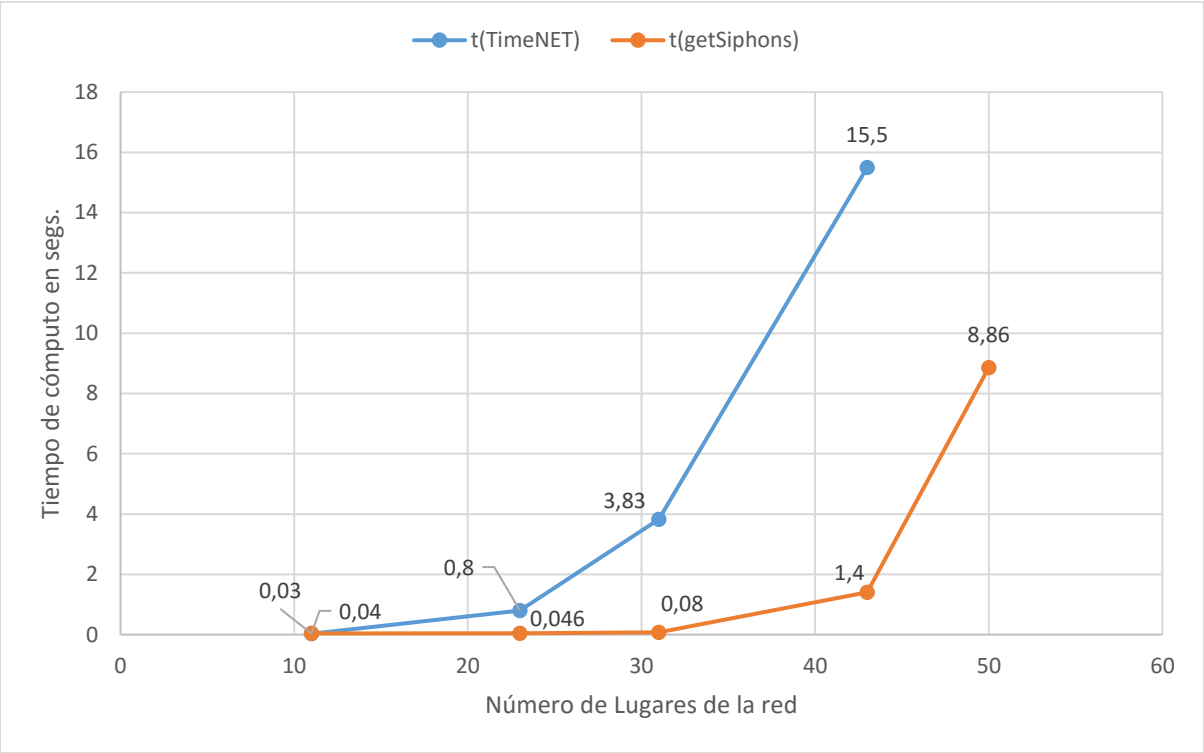


Figura 15. Gráfica comparativa entre el tiempo utilizado en el cálculo de sifones entre ambos métodos. (Cálculos realizados en un portátil HP del 2012 - Intel Core i7 2.10GHz - 6GB RAM - Windows 10)

Hay que tener en cuenta que en el caso de la red de 50 lugares, la herramienta TimeNET tarda 5055,70 segundos pero no obtiene finalmente ningún resultado (por eso no se ha incluido ningún valor en la gráfica) ya que se supera la capacidad de memoria, esto se debe a que los requerimientos de memoria de los otros métodos aumentan de forma exponencial con el número de recursos. (Ver ejemplos en detalle en *Anexo 5*).

Los objetivos propuestos se han cumplido satisfactoriamente pues se ha logrado implementar un algoritmo para encontrar los sifones en redes S4PR y para gestionarlos de modo que no se produzcan bloqueos en el sistema. También es cierto que habría sido más satisfactorio si se hubiesen podido realizar simulaciones y/o experimentos con robots reales basados en Arduino, para probar el funcionamiento de los algoritmos en una situación real. Sin embargo, esto no depende tan solo de este trabajo, pues para poderse llevar a la práctica, es necesario combinar las aportaciones de distintos compañeros que están desarrollando su trabajo de fin de grado en el Departamento de Informática e Ingeniería de Sistemas de la Universidad de Zaragoza. Por tanto, cuando se logren aunar todos estos trabajos se obtendrá una plataforma completa para controlar sistemas de robots móviles.

Aunque este trabajo sea tan sólo una pequeña parte de un proyecto mayor, se ha dedicado mucho tiempo y esfuerzo en él, sobre todo en la implementación de los algoritmos en lenguaje C++, a lo cual se ha dedicado la mayoría del tiempo desde que se comenzó este trabajo el pasado abril.

10 BIBLIOGRAFÍA Y REFERENCIAS

[1] Marius Kloetzer, Cristian Mahulea y J.M. Colom. *Petri net approach for deadlock and collision avoidance in robot planning*.

[2] Elia Esther Cano, Carlos A. Rovetto y José Manuel Colom. *An algorithm to compute the minimal siphons in S4PR nets*.

[3] Elia Esther Cano Acosta. *Teoría de Grafos Aplicada al Análisis de Redes S4PR y Subclases: El problema de Cálculo de los Cerrojos Mínimos.* Tesis doctoral

[4] Librería para el uso de grafos en C++, *Boost Graph Library*: http://www.boost.org/doc/libs/1_61_0/libs/graph/doc/table_of_contents.html

[5] *Petri Net Toolbox* for Matlab: https://es.mathworks.com/products/connections/product_detail/product_35741.html?requestedDomain=www.mathworks.com

[6] Programa *PetriNet: https://www.tu-ilmenau.de/sse/timenet/*

[7] S. Bernardi, C. Mahulea, and J. Albareda. *Toward a decision support system for the clinical guidelines assessment.* Technical report, I3A, Universidad de Zaragoza, 2016.

[8] Jorge Barrios. 2016. Trabajo Fin de Grado. *Localización de múltiples robots mediante una cámara cenital.* Director: Jose María Martínez Montiel.

[9] M. D. Berg, O. Cheong, and M. van Kreveld, *Computational Geometry: Algorithms and Applications,* 3rd ed. Springer, 2008. (Cell Decomposition).

[10] Dijkstra, E.W. Numer. Math. (1959) 1: 269. doi:10.1007/BF01386390.

[11] M. Kloetzer and C. Mahulea, *Multi-Robot Path Planning for Syntactically Co-Safe LTL Specifications*, In WODES'2016: 13rd IFAC International Workshop on Discrete Event Systems, Xi'an, China, May 2016.

[12] C. Mahulea and M. Kloetzer, "Planning Mobile Robots with Boolean-based Specifications", In CDC'2014: 53rd IEEE Conference on Decision and Control, Los Angeles, California, USA, December 2014.

[13] Daniel Clavel, Cristian Mahulea y Manuel Silva. *On liveness enforcement of DSSP net systems*.

[14] Daniel Clavel. *Desarrollo de un algoritmo basado en la preasignación de buffers que permita la vivacidad de sistemas de red DSSP inicialmente no vivos*. Trabajo Fin de Master.

[15] K. Lautenbach (1987). *Linear algebraic calculation of deadlocks and traps*. In G. Voss and Rozenberg, editors, *Concurrency and Nets Advances in Petri Nets*, pages 315–336, New York, 1987. Springer-Verlag.

[16] M. Kinuyama and T. Murata. *Generating siphons and traps by petri net representation of logic equations*. In Proceedings of 2th Conference of the Net Theory SIG-IECE, pages 93–100, 1986.

[17] M. Minoux and K. Barkaoui. *Deadlocks and traps in petri nets as horn-satisfiability solutions and some related polynomially solvable problems*. Discrete Applied Mathematics, 29:195–210, 1990.

[18] Barkaoui K, Lemaire B (1989) *An effective characterization of minimal deadlocks and traps in Petri nets based on graph theory*. In: Proceedings of the 10th international conference on theory and application of Petri nets, Bonn, pp 1–22.

[19] Chu F, Xie XL (1997) Deadlock analysis of Petri nets using siphons and mathematical programming IEEE Trans Robot Autom 13(6):793–804.

[20] Ezpeleta J, Couvreur JM, Silva M (1993) *A new technique for finding a generating family of siphons, traps and ST-components*. Application to colored Petri nets. In: Advances in Petri nets. Lecture notes in computer science, vol 674. Springer, New York, pp 126–147.

[21] Li ZW, Zhou MC (2008) *On siphon computation for deadlock control in a class of Petri nets*. IEEE Trans Syst Man Cybern Part A Syst Humans 38(3):667–679.

[22] Wang AR, Li ZW, Jia JY, Zhou MC (2009a) *An effective algorithm to find elementary siphons in a class of Petri nets*. IEEE Trans Syst Man Cybern Part A Syst Humans 39(4):912–923

[23] Manuel Silva (1985) *Las redes de Petri en la automática y la informática*.

[24] Fernando Tricas (2003) *Analysus, prevention and avoidance of deadlocks in sequential resource allocation systems*. PhD thesis. Departamento de Ingeniería Eléctrica e Informática, Universidad de Zaragoza.

[25] Sergio Frauca 2016. *Programa en C++ para el cálculo de P-semiflujos.* Trabajo de la asignatura Evaluación y Control de los Sistemas de Producción del Master de Ingeniería Industrial.

ANEXOS

ANEXO 1. FUNDAMENTOS DE REDES DE PETRI, SIFONES Y VIVACIDAD

***Definición 1. Red de Petri (RdP)*:** [23]
Una red de Petri es una terna $\mathcal{N} = \{P, T, F, W, M_0\}$, donde:

- $P = \{p_1, p_2, \ldots, p_m\}$ es un conjunto finito de lugares;
- $T = \{t_1, t_2, \ldots, t_n\}$ es un conjunto finito de transiciones;
- $F \subseteq (P \times T) \cup (P \times T)$ es un conjunto de arcos;
- $W: F \to \{1,2,3, \ldots\}$ es la función de ponderación de los arcos;
- $M_0: P \to \{0,1,2,3, \ldots\}$ es la función del marcado inicial.

Algunos comentarios relacionados con esta definición: [Apuntes ingeniería de control]

- Los conjuntos P y T son disjuntos $P \cap T = \emptyset$.
- Para asegurar la definición anterior, los conjuntos P y T satisfacen la condición $P \cup T \neq \emptyset$.
- Una estructura de red de Petri $N = \{P, T, F, W\}$, sin ninguna indicación del marcado se denota por N.
- Una red de Petri con el marcado inicial m_0 se denota por $\{\mathcal{N}, m_0\}$.
- Una red de Petri con el marcado cualquiera m se denota por $\{\mathcal{N}, m\}$.

Si un lugar p es lugar de entrada y salida de la transición t, entonces p y t forman un auto-bucle (self-loop en inglés). Una red de Petri que no contiene auto-bucles se llama ***pura*** y una red de Petri está libre de auto-bucles, si y solo si $\forall x, y \in P \cup T$, si $W(x, y) > 0$ implica que $W(y, x) = 0$.

Una red de Petri se llama es una red ***generalizada*** si todos sus arcos tienen pesos positivos y se llama red ***ordinaria*** cuando todos sus arcos tienen peso unidad, es decir, la función de ponderación W puede ser definida como $(P \times T) \cup (T \times P) \to \{0,1\}$.

Una red de Petri ordinaria se llama ***máquina de estados*** si para cada transición $t \in T, |{}^\bullet t| = |t^\bullet| = 1$. Es decir, si sus transiciones tan solo tienen un lugar de entrada y un lugar de salida. [2]

El marcado de una red de Petri tiene el significado del estado de la red y va evolucionando y cambiando el marcado de acuerdo con la llamada ***ley de transición*** *(o de sensibilización y disparo):*

a) Se dice que una transición t está sensibilizada (o habilitada) si cada lugar de entrada p de t está marcado con al menos $W(p, t)$ marcas, donde $W(p, t)$ es el peso del arco del lugar p a la transición t.
b) Una transición sensibilizada se puede o no se puede disparar dependiendo de si el evento externo relacionado ocurre o no.
c) El disparo de una transición elimina $W(p, t)$ marcas de cada lugar de entrada p de t, y añade $W(p, t)$ marcas en cada lugar de salida p de t, donde $W(p, t)$ es el peso del arco del lugar p a la transición t.

Definición 2. Matriz de Incidencias C: *[24]*

Una red de Petri $\mathcal{N} = \{P, T, F, W, M_0\}$ si es *pura* puede ser también definida por la terna $\mathcal{N} = \{P, T, C\}$, donde C es la ***matriz de incidencias*** de la red. La matriz C es una matriz de enteros de dimensión $P \times T$ que se forma a partir de las funciones de ponderación de todos los arcos de la red, de modo que: $C[p,t] = W(p,t) - W(t,p)$. *[2]*

En el caso de no ser una red pura, es decir, si la red contiene bucles (*self-loops*) la matriz de incidencias C no representa esos bucles, de modo que esta representación se queda corta. Para solucionar este problema, la red de Petri se ha de definir como $\mathcal{N} = \{P, T, Pre, Post\}$, donde Pre es la matriz de pre-incidencias de dimensión $P \times T$, siendo $Pre[p,t] = W(t,p)$ y $Post$ es la matriz de post-incidencias de dimensión $P \times T$, siendo $Post[p,t] = W(p,t)$.

Definición 4. P-semiflujo *[2]:* Un p-semiflujo es un $Y \in \mathbb{N}^{|\boldsymbol{P}|}$, $Y \neq 0$, el cual es un anulador izquierdo de la matriz de incidencia $Y \cdot C = 0$.

El soporte de un p-semiflujo es denotado $||Y||$, y sus lugares están cubiertos por Y. Un *p-semiflujo mínimo* es un p-semiflujo tal que el máximo común divisor de sus componentes no-nulas es uno y su soporte $||Y||$ no contiene el soporte de ningún otro p-semiflujo.

Si se multiplica la ecuación de estado por un P-semiflujo y se obtiene una relación lineal entre las variables del marcado que permanece cierta en todos los estados alcanzables.

$$y \cdot m = y \cdot m_0 + y \cdot C \cdot \sigma = y \cdot m_0 = constante.$$

Definición 5. T-semiflujo *[2]:* Un t-semiflujo es un $X \in \mathbb{N}^{|\boldsymbol{P}|}$, $X \neq 0$, el cual es un anulador derecho de la matriz de incidencia $C \cdot X = 0$.

El soporte de un t-semiflujo es denotado $||X||$, y sus lugares están cubiertos por X. Un t-*semiflujo mínimo* es un t-semiflujo tal que el máximo común divisor de sus componentes no-nulas es uno y su soporte $||X||$ no contiene el soporte de ningún otro t-semiflujo.

Definición 6. Sifón o Cerrojo *[2]: Sea* $\mathcal{N} = \{P, T, C\}$ una red de Petri, un subconjunto de lugares $D \subseteq P$ es un sifón si, y solo si, el conjunto de transiciones de entrada está contenido en el conjunto de transiciones de salida, es decir, ${}^{\bullet}D \subseteq D^{\bullet}$. (Ver ejemplo *Figura 16*)

Los *Sifones mínimos* son aquellos sifones que no contienen ningún otro sifón que sea subconjunto del mismo.

Siendo $D \subseteq P$ un sifón mínimo no-vacío de $\mathcal{N}$ que contiene al menos un lugar de recurso. D es el único sifón mínimo de $\mathcal{N}$ que contiene exactamente el conjunto de recursos $D_R = D \cap P_R$. Al conjunto de todos los sifones de una red de Petri que contienen un único lugar de recurso cada uno, se le llama $\mathcal{D}^1$.

Definición 7. Trampa *[2]: Sea* $\mathcal{N} = \{P, T, C\}$ una red de Petri, un subconjunto de lugares $D \subseteq P$ es una trampa si, y solo si, el conjunto de transiciones de salida está contenido en el conjunto de transiciones de entrada, es decir, $D^{\bullet} \subseteq {}^{\bullet}D$. (Ver ejemplo *Figura 16*)

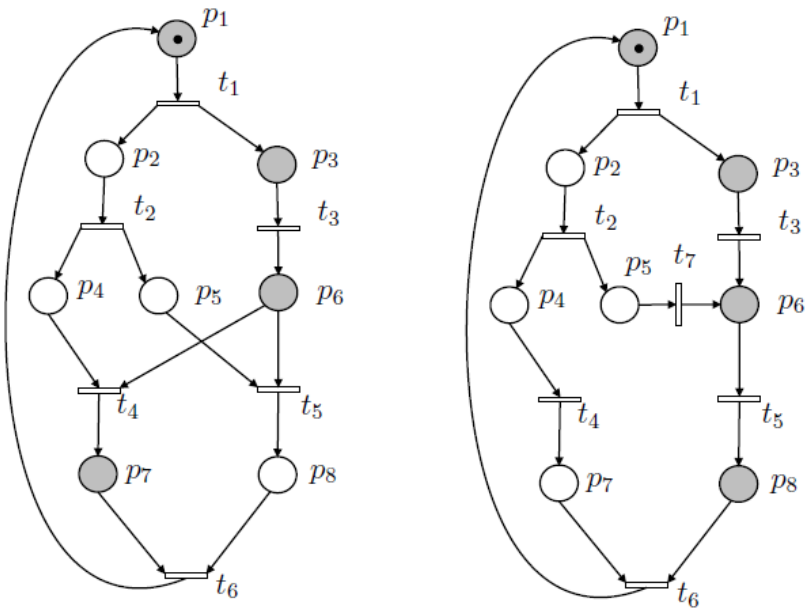


Figura 16. Ejemplo de red de Petri con un sifón (izq.) con los lugares pertenecientes al sifón resaltados en gris y ejemplo de red de Petri con una trampa (drcha.) con los lugares pertenecientes a la trampa resaltados en gris [6]

Definición 3. Red S4PR: *Se trata de una subclase de las redes de Petri.* [2]

Siendo $I_N = \{1, 2, \dots, m\}$ un conjunto finito de índices. Una red S4PR es una red de Petri $\mathcal{N}$ conectada generalizada pura (libre de auto bucles), tal que $\mathcal{N} = \langle P, T, \boldsymbol{C} \rangle$ donde:

1) $P = P_0 \cup P_S \cup P_R$ es un conjunto tal que:
   a) $P_S = \bigcup_{i \in I_N} P_{S_i}$, $P_{S_i} \neq \emptyset$ *y* $P_{S_i} \cap P_{S_i} = \emptyset$, para todo $i \neq j$.
   b) $P_0 = \bigcup_{i \in I_N} \{p_{0_i}\}$.
   c) $P_R = \{r_1, r_2, \dots, r_n\}, n > 0$.
2) $T = \bigcup_{i \in I_N} T_i$, $T_i \neq \emptyset, T_i \cap T_j = \emptyset$, para toda $i \neq j$
3) Para toda $i \in I_N$, la subred N_i generada por $P_{S_i} \cup \{p_{0_i}\} \cup T_i$ es una máquina de estados fuertemente conexa, tal que cada ciclo contenga p_{0_i} .
4) Para cada $r \in P_R$ existe un *P-Semiflujo,* $y_r \in \mathbb{N}^{|\boldsymbol{P}|}$, tal que $\{r\} = ||y_r|| \cap P_R$, $y_r[r] = 1$, $P_0 \cap ||y_r|| = \emptyset$, y también $P_S \cap ||y_r|| = \emptyset$.
5) $P_S = \bigcup_{r \in P_R} (||y_r|| \setminus \{r\})$.

Los lugares del conjunto P_S son los lugares de proceso, a cada lugar p_{0_i} se le llama lugar de reposo. Los recursos compartidos pertenecen al conjunto P_R y a cada recurso se le llama r.

Las sigas de S4PR no significan nada, tan solo representan que son una extensión de las redes S3PR (*Systems of Simple Sequential Processes with Resources*) que son un subconjunto de las redes S4PR.

Definición 8. Lugares Ladrones [24]

Sea $\langle \mathcal{N}, m_0 \rangle$, $\mathcal{N} = \{P_0 \cup P_S \cup P_R$, T, $C\}$ una red S4PR marcada. Y sea D un sifón de $\mathcal{N}$. Entonces $\mathcal{TH}_D = \mathcal{H}_{D_R} \setminus D_S$ es el conjunto de ladrones de D. Estos son los lugares de la red que usan los recursos del sifón pero no pertenecen al sifón.

Se define al *conjunto de usuarios* del recurso r ($\mathcal{H}_r$ o *holders* en terminología inglesa) como el conjunto de los lugares proceso que pueden utilizar dicho recurso. Esta definición puede extenderse a los conjuntos de recursos, como en este caso $D_R \subseteq P_R$: $\mathcal{H}_{D_R} = \bigcup_{r \in D_R} \mathcal{H}_r$.

Definición 9. Vivacidad: ausencia de un marcado alcanzable en el que la red se bloquee, aunque sea parcialmente. [24]

Una transición t está viva para un marcado inicial dado m_0, si existe una secuencia de disparos a partir de un marcado m sucesor de m_0 que incluya a t. Una RdP es viva para m_0 si todas sus transiciones son vivas para m_0.

Según Tricas [24], Una red S4PR, $\mathcal{N} = \{P_0 \cup P_S \cup P_R\ ,\ T,\ C\}$, es no-viva si, y solo si, existe un sifón D, y un marcado m_D alcanzable a partir del marcado inicial m_0, tal que:

1. $m_D[P_S] > 0$.
2. $m_D[P_S \backslash \mathcal{TH}_D] = 0$.
3. $\forall p \in \mathcal{TH}_D$ tal que $m_D[p] > 0$, el disparo de cada $t \in p^\bullet$ se previene con un conjunto de lugares pertenecientes al sifón

Definición 10. Grafo Dirigido o Digrafo: [3]

Un grafo dirigido es una terna $G = (V, E, \varphi)$ con $V \neq \emptyset$ donde:

- $V = \{v_1, v_2, \dots, v_n\}$: conjunto de vértices o nodos.
- $E = \{e_1, e_2, \dots, e_n\}$: conjunto de aristas o arcos.
- $\varphi: A \to V \times V$ función de incidencia. En donde, si $\varphi(e) = (v, w)$ se dice que:
  - los vértices v y w son adyacentes.
  - e incide positivamente en w y negativamente en v.
  - v es extremo inicial de la arista e, y w es extremo final de e.

Definición 11. Grafo de Poda (Prunning Graph en inglés): *[2]*

Siendo $\mathcal{N}$ una red S4PR y P_R el conjunto de lugares de recurso. El *Grafo de Poda* (*PG*) de $\mathcal{N}$ es un digrafo $G = (V, E)$ donde, (1) $V = P_R$; y (2) $E \subseteq V \times V$ y para todo recurso $r, x \in P_R$, $(r, x) \in E$ si, y solo si, $U_r = r^\bullet \cap T_{rx} \cap ({}^\bullet T_{rx} \cap D_x \cap P_s)^\bullet \neq \emptyset$ con $T_{rx} = D_r^\bullet \cap D_x^\bullet$.

Dado un *Grafo de Poda* $G = (V, E)$, se define el *Subgafo de Poda* G' *de* G inducido sobre el conjunto de vértices $V' \subseteq V$, así como el grafo $G' = \big(V', E \cap (V' \times V')\big)$. Asociadas al Grafo de Poda o al Subgrafo de poda, se definen las siguientes tres funciones:

- Función de etiquetado de recursos (*Resource Labelling Function*) $S: P_R \to \mathcal{D}^1$, donde para todo $r \in P_R$, $S(r) = D_r \in \mathcal{D}^1$, y $D_r \cap P_R = \{r\}$.
- Función de etiquetado de arcos (*Arc Labelling Function*) $L: E \to 2^{T \times P_S}$, donde $\forall (r, x) \in E, L\big((r, x)\big) = \{(t, p) | t \in r^\bullet \cap T_{rx} \cap ({}^\bullet T_{rx} \cap D_x \cap P_S)^\bullet; T_{rx} = S(r)^\bullet \cap S(x)^\bullet; \{p\} = {}^\bullet t \cap P_S\}$.
- Función de etiquetado de poda (*Prunning Labelling Function*) $K_G: V \to 2^{P_S}$, donde para todo $r \in V$, $K_G(r) \subseteq P_S$ es calculado mediante el *Algoritmo 1* del *Anexo 2.*

ANEXO 2: ALGORITMO DE CÁLCULO DE SIFONES PARA REDES S4PR

Algoritmos utilizados para la implementación del programa de cálculo de sifones mínimos en redes S4PR, desarrollados por E.E.Cano et al. en [2]:

Algoritmo 1 Función de etiquetado de Poda de G.

Entrada: $\mathcal{N} = \langle P_0 \cup P_s \cup P_R, T, C \rangle$ -- La red S^4PR.
$r \in P_R$ -- El recurso de un cerrojo $D_r \in \mathscr{D}^1$ para el cual calcularemos el conjunto de poda en el grafo G.
$G = (V, E)$ -- El grafo o subgrafo de poda de $\mathcal{N}$ y las funciones asociadas L y S.
Salida: $K_G(r) \subseteq P_s$

1. **Inicio**
2. $T_F = \{t | (t,p) \in L((x,r)) \text{ y } (x,r) \in E\}$
3. $P_{parcial} = \{p | (t,p) \in L((x,r)); (x,r) \in E; \text{ y } p^\bullet \cap T_F \neq p^\bullet\}$
4. $K_G(r) = \{p | (t,p) \in L((x,r)); (x,r) \in E; \text{ y } p^\bullet \subseteq T_F\}$
5. $T_{nuevo} = {}^\bullet K_G(r) \backslash r^\bullet$
6. **Mientras que** $T_{nuevo} \neq \emptyset$ **Hacer**
7. $T_F = T_F \cup T_{nuevo}$
8. $P_{nuevo} = {}^\bullet T_{nuevo} \cap P_s \cap S(r)$
9. $A = \{p | p \in P_{nuevo}; p^\bullet \subseteq T_F\}$
10. $B = \{p | p \in P_{parcial}; p^\bullet \subseteq T_F\}$
11. $C = \{p | p \in P_{nuevo}; p^\bullet \cap T_F \neq p^\bullet\}$
12. $K_G(r) = K_G(r) \cup A \cup B$
13. $P_{parcial} = (P_{parcial} \backslash B) \cup C$
14. $T_{nuevo} = {}^\bullet(A \cup B) \backslash r^\bullet$
15. **Fin Mientras que**
16. **Fin**

Figura 17. Algoritmo 1 desarrollado por E.E.Cano et. al.

Algoritmo 2 Cálculo de los Cerrojos Mínimos de una red S^4PR $\mathcal{N}$.

Entrada: $\mathcal{N} = \langle P_0 \cup P_s \cup P_R, T, C \rangle$ – – La red S^4PR.
$\mathcal{D}^1$ – – Los cerrojos mínimos con solo un recurso.

Salida: $\mathcal{D}$ – – Los cerrojos mínimos de $\mathcal{N}$ conteniendo al menos un recurso.

1. **Inicio**
2. $\mathcal{D} := \mathcal{D}^1$
3. Calcular el grafo de poda $G = (V, E)$ de $\mathcal{N}$ (definición 8), y las funciones S, L y K_G (definición 9).
4. Calcular todos los subgrafos fuertemente conexos máximos (componentes fuertemente conexas, cfc) de G, y para cada cfc, G', calcular las funciones S, L y $K_{G'}$ para formar el grafo de poda G'. Agregar todos los grafos de poda G' con más de un vértice al conjunto g_{nuevo}.
5. $g_{bueno} := \emptyset$
6. **Mientras que** $g_{nuevo} \neq \emptyset$ **Hacer**
7. Extraer $G' = (V', E')$ de g_{nuevo}
8. **Si** existe $r \in V'$ tal que $K_{G'}(r) = \emptyset$ **Entonces**
9. Calcular $A := \{r | r \in V' \text{ y } K_{G'}(r) = \emptyset\}$
10. Calcular el subgrafo de poda de G' inducido por $V' \backslash A$, denominado G''.
11. Calcular todos los cfc de G'', y para cada cfc, G'', calcular las funciones S, L y $K_{G''}$ para formar el grafo de poda G'''. Agregar todos los grafos de poda G''' con más de un vértice al conjunto g_{nuevo}.
12. **Sino**
13. Agregar G' al conjunto g_{bueno}
14. **Para** cada $r \in V'$ **Hacer**
15. Calcular el subgrafo de poda de G' inducido por $V' \backslash \{r\}, G''$
16. Calcular todos los cfc de G'', y para cada cfc, G''', calcular las funciones S, L y $K_{G'''}$ para formar el grafo de poda G'''. Agregar todos los grafos de poda G''' con más de un vértice al conjunto g_{nuevo}
17. **Fin Para**
18. **Fin Si**
19. **Fin Mientras que**
20. Remover todos los grafos de $g_{bueno}, G = (V, E)$, para los cuales exista otro $G' = (V', E')$ en g_{bueno} que satisface: $(1) V' \subseteq V$; y $(2) S_{G'} \subset S_G$
21. **Para** cada grafo $G' = (V', E') \in g_{bueno}$ **Hacer**
22. Agregar a $\mathcal{D}$ el cerrojo $D = \bigcup_{r \in V'} S(r) \backslash K_{G'}(r) = S_{G'}$
23. **Fin Para**
24. **Fin**

Figura 18. Algoritmo 2 desarrollado por E.E.Cano et. al.

ANEXO 3: DESCRIPCIÓN DEL PROGRAMA, FUNCIONES Y CLASES

Para el programa se ha definido una clase llamada *PetriNetClass* y una subclase o clase hija dentro de esta llamada *S4PR*. Además hay una tercera clase, *Matrix*, para trabajar con matrices. Dentro de cada clase se han definido una serie de funciones. Para un mejor entendimiento ver el diagrama de clases del programa (ver *Figura 14* en la *Sección 7*), donde aparecen también las variables públicas y protegidas de cada clase. Como se trata de programación orientada a objetos para usar estas funciones habrá que definir previamente una variable de la clase sobre la que se quiere trabajar y después llamar a la función como se indica a continuación en este ejemplo:

```
PetriNetClass Red_de_Petri;
Matrix<int> matrizC;
matrizC = Red_de_Petri.getC();
```

A continuación se explican sólo las funciones públicas de cada clase, es decir, las que pueden ser usadas en el programa principal, ya que las privadas se han usado sólo para desarrollar estas otras:

A3.1. FUNCIONES PÚBLICAS EN LA CLASE PETRINETCLASS:

- ***postset(set<int> nodes, char nodeType, bool print):*** es una función de tipo *set<int>* que devuelve un conjunto de enteros el cual representa los nodos (transiciones o lugares) de salida o posteriores al conjunto de nodos (lugares o transiciones) de la variable de entrada *nodes.* La variable de entrada *nodeType* es un carácter que representa el tipo de los nodos de entrada, es decir, si *nodeType='t'* el conjunto de entrada serán transiciones y si *nodeType='p'* el conjunto de entrada serán lugares (places). Por último la variable en entrada *print* es un booleano que si toma el valor 1 (o *true*) la función imprimirá el conjunto de nodos y su correspondiente *postset*, conjunto de nodos de salida o posteriores. Ejemplo: "`The output (post) set of the set of places {1,2,3} is the set of transitions {2,3,4}`".
- ***preset(set<int> nodes, char nodeType, const bool print):*** es una función de tipo *set<int>* que devuelve un conjunto de enteros el cual representa los nodos (transiciones o lugares) de entrada o predecesores al conjunto de nodos (lugares o transiciones) de la variable de entrada *nodes.* La variable de entrada *nodeType* es un carácter que representa el tipo de los nodos de entrada, es decir, si *nodeType='t'* el conjunto de entrada serán transiciones y si *nodeType='p'* el conjunto de entrada serán lugares (places). Por último la variable en entrada *print* es un booleano que si toma el valor 1 (o *true*) la función imprimirá el conjunto de nodos y su correspondiente *postset* o conjunto de nodos de salida. Ejemplo: "`The input (pre) set of the set of places {1,2,3} is the set of transitions {2,3,4}`".
- ***setValues(int nplaces, int ntrans, vector<int> m0, Matrix<int> Pre, Matrix<int> Post, Matrix<int> C):*** es una función de tipo *void* que lo que hace es asignar a la clase *PetriNetClass* los siguientes valores: número de lugares, número de transiciones, el vector de marcado inicial y las matrices Pre, Post y C. Es decir, asigna los valores de las variables de entrada *nplaces, ntrans, m0, Pre, Post* y *C,* a las correspondientes variables de la clase.
- ***isStateMachine():*** es una función de tipo *bool* que devuelve un booleano que toma el valor 1 (o *true)* si la red de Petri es una máquina de estados. (Ver definición de máquina de estados en *Anexo 1.).*
- ***isSiphon(set<int> SetOfPlaces):*** es una función de tipo *bool* que devuelve un booleano que toma el valor 1 (o *true)* si el conjunto de enteros de entrada *SetOfPlaces* es un sifón de la

red, es decir comprueba si las transiciones de entrada de ese conjunto de lugares están incluidas en el conjunto de transiciones de salida, cumpliendo así con la definición de sifón (ver en *Anexo 1).*

- ***getC():*** es una función del tipo *Matrix<int>* que devuelve la matriz de incidencias C de la red de Petri que se ha almacenado en la variable protegida *C*, obtenida de restar *Post-Pre.* (Ver definición de Matriz de incidencias en *Anexo 1*.)
- ***getPre():*** es una función del tipo *Matrix<int>* que devuelve la matriz de Pre de la red de Petri que se ha leído del fichero y que se ha almacenado en la variable protegida *Pre*.
- ***getPost():*** es una función del tipo *Matrix<int>* que devuelve la matriz de Post de la red de Petri que se ha almacenado en la variable protegida *Post*.
- ***getM0():*** es una función del tipo *vector<int>* que devuelve un vector de enteros de dimensión igual al número de lugares de la red y en el que se almacena el número de marcas correspondiente a cada lugar.
- ***Psemiflows():*** es una función de tipo *Matrix<int>* que devuelve una matriz en la que cada fila es un vector correspondiente a un p-semiflujo de la red de Petri. Cada p-semiflujo será un vector que será un anulador izquierdo de la matriz de incidencias. (Ver definición del *Anexo 1.*). Para esta función se hace uso de una función protegida llamada *semiflows* que tiene como entrada la matriz C y que no ha sido desarrollada en este trabajo, esta función fue desarrollada por Sergio Frauca en [25].
- ***Tsemiflows():*** es una función de tipo *Matrix<int>* que devuelve una matriz en la que cada fila es un vector correspondiente a un t-semiflujo de la red de Petri. Cada t-semiflujo será un vector que será un anulador izquierdo de la matriz de incidencias. (Ver definición del *Anexo 1.*). Para esta función se hace uso de una función protegida llamada *semiflows* tomando como entrada la traspuesta de la matriz C.

A3.2. FUNCIONES PÚBLICAS EN LA SUBCLASE S4PR:

- ***read_from_file(FILE *fichero):*** es una función de tipo *void* que lee del fichero de entrada y almacena las variables *nplaces, ntrans, Post, Pre, Ps, Pr, numProceses, numResources* y *numPmonitor.* Estas variables quedan almacenadas como variables protegidas de la clase S4PR.
- ***getPlaces():*** es un función de tipo *int* que devuelve un entero que es el número de lugares que tiene la red de Petri, es decir, devuelve el valor almacenado en la variable protegida *nplaces.*
- ***getTransitions():***es un función de tipo *int* que devuelve un entero que es el número de transiciones que tiene la red de Petri, es decir, devuelve el valor almacenado en la variable protegida *ntrans.*
- ***getProcessPlaces(int process):*** es un función de tipo *set<int>* que devuelve un conjunto de enteros que almacena todos los lugares del proceso indicado en la variable de entrada *process*.
- ***getProcessNet(int process,bool print);*** es un función de tipo *PetriNetClass* que devuelve una red de Petri que contiene únicamente los lugares correspondientes al proceso indicado en la variable de entrada *process.* La variable de entrada *print* es un booleano que en el caso de ser 1, la función imprimirá los resultados.
- ***getProcessNo():*** es un función de tipo *int* que devuelve un número entero con el número de procesos que tiene la red de Petri, es decir, devuelve el valor almacenado en la variable protegida *numProcesses.*

- ***getResourceNo():*** es un función de tipo *int* que devuelve un número entero con el número de lugares de recurso que tiene la red de Petri, es decir, devuelve el valor almacenado en la variable protegida *numResources.*
- ***getPmonitorNo():*** es un función de tipo *int* que devuelve un número entero con el número de lugares monitor para controlar los sifones malos que tiene la red de Petri, es decir, devuelve el valor almacenado en la variable protegida *numPmonitor.*
- ***getIdlePlace(int process):*** es un función de tipo *int* que devuelve un entero que almacena el número del lugar de reposo correspondiente al proceso indicado en la variable de entrada *process*.
- ***getP0():*** es una función de tipo *set<int>* que devuelve un conjunto de enteros con los lugares de reposo de cada proceso de la red de Petri, es decir, devuelve el conjunto almacenado en la variable protegida *P0.*
- ***getPs():*** es una función de tipo *set<int>* que devuelve un conjunto de enteros con los lugares de proceso de cada proceso de la red de Petri, es decir, devuelve el conjunto almacenado en la variable protegida *Ps.*
- ***getPr():*** es una función de tipo *set<int>* que devuelve un conjunto de enteros con los lugares de recurso de toda la red de Petri, es decir, devuelve el conjunto almacenado en la variable protegida *Pr.*
- ***getPmonitor():*** es una función de tipo *set<int>* que devuelve un conjunto de enteros con los lugares de control de los sifones o lugares monitor de la red de Petri, es decir, devuelve el conjunto almacenado en la variable protegida *Pmonitor.*
- ***isWellDefinedS4PR():*** es una función de tipo *bool* que devuelve un booleano, es decir, devuelve un 1 (o *true*) si la red de Petri almacenada en la clase es una red de tipo S4PR y está bien definida como tal, en caso contrario devuelve 0 (o *false*). Esta función lo que hace es ejecutar las funciones *isCond1(), isCond2(), isCond3(), isCond4()* y *isCond5(),* y estas funciones lo que hacen es comprobar si se cumplen las condiciones 1,2,3,4 y 5 de la definición de red S4PR (ver *Definición 3.* del *Anexo 1.*).
- ***D1_Siphons():***es una función de tipo *vector< set<int> >* que devuelve un vector y en cada componente almacena un conjunto de enteros, cada uno de esos conjuntos es un sifón de la clase $\mathcal{D}^1$, es decir devuelve los sifones que contienen un solo recurso de la red de Petri.
- ***getSiphons():*** es una función de tipo *vector< set<int> >* que devuelve un vector y en cada componente almacena un conjunto de enteros, cada uno de esos conjuntos es un sifón de la red de Petri. Esta función utiliza el algoritmo del grafo de poda sobre el que se centra este trabajo para la obtención de los sifones. Así que se obtienen todos los sifones mínimos con algún recurso, es decir, no se obtienen los sifones que no contienen lugares de recurso como los sifones formados por todos los lugares de proceso de un proceso. Esta función además almacena los sifones calculados en la variable protegida *siphons*.
- ***printSiphons():*** es una función de tipo *void* que imprime en pantalla una lista de todos los sifones de la red de Petri, es decir, imprime los sifones almacenados en la variable protegida *siphons* y en el caso de que esta variable esté vacía porque aún no se hayan calculado los sifones, esta función usa la función *getSiphons* para calcularlos.
- ***getBadSiphons():*** es una función de tipo *vector< set<int> >* que devuelve un vector y en cada componente almacena un conjunto de enteros, cada uno de esos conjuntos es un sifón "malo" de la red de Petri. Esta función utiliza la función *getSiphons* para la obtención de los sifones y a partir de ellos, elimina los sifones que contengan un p-semiflujo (calculados con la función *Semiflows(C)*). Así que se obtienen todos los sifones malos mínimos, es decir, sólo

son los sifones que generan bloqueos. Esta función además almacena los sifones malos calculados en la variable protegida *bad_siphons*.

- ***printBadSiphons():*** es una función de tipo *void* que imprime en pantalla una lista de todos los sifones "malos" de la red de Petri, es decir, imprime los sifones almacenados en la variable protegida *bad_siphons* y en el caso de que esta variable esté vacía porque aún no se hayan calculado los sifones, esta función usa la función *getBadSiphons* para calcularlos.
- ***getControlSiphons():*** es una función de tipo S4PR que devuelve una red de Petri de clase S4PR modificada. Esta red de Petri de salida ha sido modificada añadiendo un lugar monitor por cada sifón malo que tenga la red inicialmente.
- ***getControlledPetriNet():*** es una función de tipo S4PR que devuelve una red de Petri de clase S4PR modificada igual que hacía la función anterior, pero esta función hace el proceso de forma iterativa, es decir una vez obtenida la red con los lugares monitor, la función recalcula los sifones y si aparecen nuevos sifones malos, vuelve a añadir lugares monitor hasta que no aparezcan nuevos sifones malos.

A3.3. FUNCIONES PÚBLICAS DE LA CLASE MATRIX<T>

Siendo *T* el tipo de variable del cual se defina la clase en el programa principal, en este caso, en la aplicación de este trabajo se ha trabajado íntegramente con matrices de enteros, es decir, tipo *Matrix<int>*.

- ***isSymmetric():*** es una función de tipo *bool* que devuelve un booleano, *true* si la matriz es simétrica.
- ***isIdentity():*** es una función de tipo *bool* que devuelve un booleano, *true* si la matriz es la matriz identidad.
- ***get_max():*** es una función de tipo *T* (es decir del mismo tipo que los valores de la matriz sobre la que se trabaja) que devuelve la componente de la matriz que sea mayor de toda la matriz.
- ***get_min():*** es una función de tipo *T* (es decir del mismo tipo que los valores de la matriz sobre la que se trabaja) que devuelve la componente de la matriz que sea menor de toda la matriz.
- ***get_mode():*** es una función de tipo *T* (es decir del mismo tipo que los valores de la matriz sobre la que se trabaja) que devuelve la moda calculada a partir de todas las componentes de la matriz.
- ***get_nrows():*** es una función de tipo *int* que devuelve el número de filas que tiene la matriz de la clase sobre la que se trabaja.
- ***get_ncols():*** es una función de tipo *int* que devuelve el número de columnas que tiene la matriz de la clase sobre la que se trabaja.
- ***get_matrix():*** es una función de tipo *T*** que devuelve la matriz almacenada en la clase sobre la que se trabaja.
- ***delete_matrix():*** es una función de tipo *void* que elimina la matriz almacenada en la clase sobre la que se trabaja.
- ***fill_by_user():*** es una función de tipo *void* que pide al usuario por pantalla el número de filas y columnas y todas las componentes una a una de la matriz que se quiere almacenar en la clase sobre la que se trabaja y la almacena en dicha clase.
- ***fill_from_file(FILE *fichero, int rows, int cols*):** es una función de tipo *void* que lee de un fichero primero el número de filas y columnas y después todas las componentes una a una

de la matriz que se quiere almacenar en la clase sobre la que se trabaja y la almacena en dicha clase.

- ***fill_random():*** es una función de tipo *void* que rellena la matriz de la clase sobre la que se trabaja, de forma aleatoria, obteniendo así una matriz en la que todas sus componentes toman números aleatorios.
- ***get_transpose():*** es una función de tipo *Matrix<T>* que devuelve la matriz traspuesta de la matriz almacenada en la clase sobre la que se trabaja.
- ***multiply_by_scalar(T):*** es una función de tipo *void* que sustituye la matiz almacenada en la clase sobre la que se trabaja por la misma matriz multiplicada por el número escalar de entrada de tipo *T.*
- ***print_matrix():*** es una función de tipo *void* que imprime en pantalla la matriz almacenada en la clase sobre la que se trabaja.
- ***swap_cols(int, int):*** es una función de tipo *void* que sustituye la matiz almacenada en la clase sobre la que se trabaja por la misma matriz pero con dos columnas intercambiadas. Estas dos columnas a intercambiar son las indicadas en las dos variables de entrada de tipo *int.*
- ***swap_rows(int, int):*** es una función de tipo *void* que sustituye la matiz almacenada en la clase sobre la que se trabaja por la misma matriz pero con dos filas intercambiadas. Estas dos filas a intercambiar son las indicadas en las dos variables de entrada de tipo *int.*
- ***isEmpty():*** es una función de tipo *bool* que devuelve un booleano, *true* si la matriz está vacía.
- ***rowFromMatrix(int row):*** es una función de tipo *vector<int>* que devuelve un vector de enteros cuyas componentes son los números de la fila de la matriz sobre la que se trabaja. El número de la fila se indica en la variable de entrada *row.*
- ***setFromMatrix(int row):*** es una función de tipo *set<int>* que devuelve un conjunto de enteros formado por los números de la fila de la matriz sobre la que se trabaja. El número de la fila se indica en la variable de entrada *row.*
- ***subMatrix(set<int> rows, set<int> cols):*** es una función de tipo *Matrix<T>* que devuelve una submatriz que contiene tan solo el número de las filas y columnas de la matriz sobre la que se trabaja indicadas en las variables de entrada que son conjuntos de enteros, rows (filas) y *cols* (columnas).
- ***add_row_of_zeros():*** es una función de tipo *Matrix<T>* que devuelve una matriz igual que la matriz sobre la que se trabaja pero con una fila de ceros adicional.
- ***add_col_of_zeros():*** es una función de tipo *Matrix<T>* que devuelve una matriz igual que la matriz sobre la que se trabaja pero con una columna de ceros adicional.

ANEXO 4: INSTALACIÓN DE ECLIPSE, CREACIÓN DEL PROYECTO E INCLUSIÓN DE LIBRERÍAS.

Lo primero es descargar el *software* que servirá como entorno de programación, Eclipse. Para ello se debe entrar en la página https://eclipse.org/ → DOWNLOAD → Download Packages y descargar el Eclipse IDE for C/C++ Developers.

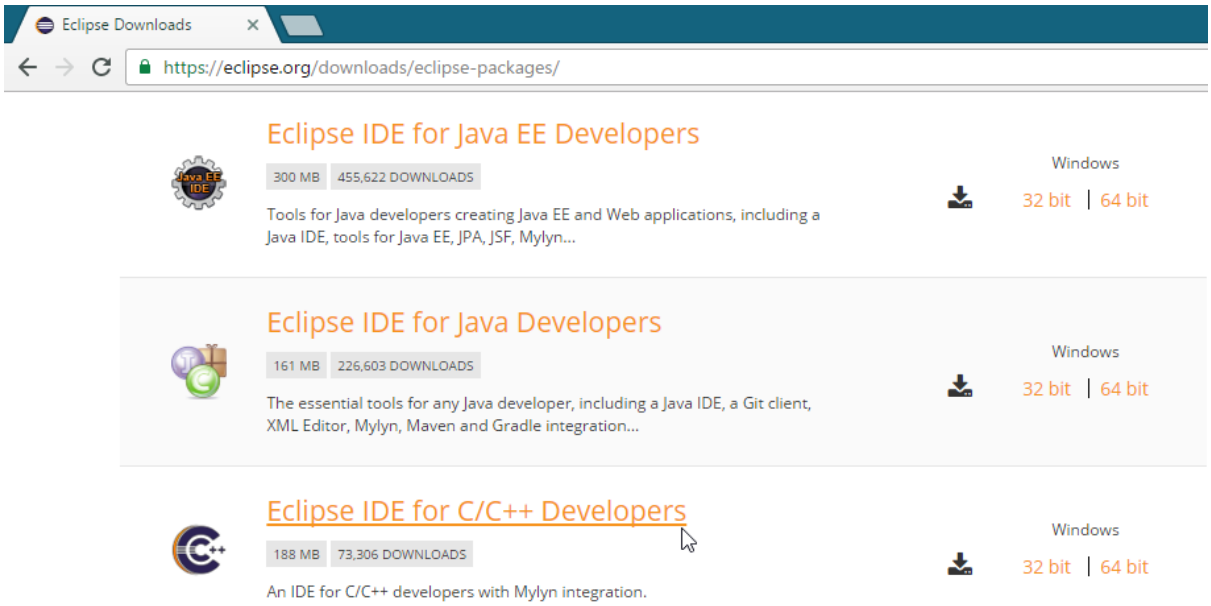


Figura 19. Descarga de Eclipse IDE for C++ Developers

Después hay que extraer el archivo comprimido, y guardar la carpeta eclipse donde se desee y ejecutar el archivo *eclipse.exe* que hay dentro de la carpeta extraída para ejecutar el programa, si se desea se puede crear un acceso directo de la misma.

Es posible que para *Windows* haya problemas con los compiladores para C++, si esto ocurriese o habría que instalar el compilador *MinGW*. Y en el caso de que siguiese apareciendo error al ejecutar eclipse, habrá que actualizar la versión de *Java.*

Una vez se consigue ejecutar eclipse sin problemas se establece la ruta donde se guardaran los trabajos realizados.

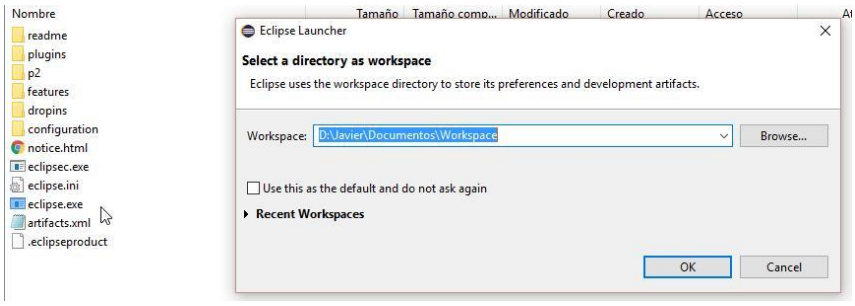


Figura 20. Sección de la carpeta de Trabajo (Workspace)

Después con eclipse abierto hay que ir a *File* → *New* → *C ++ Proyect*.

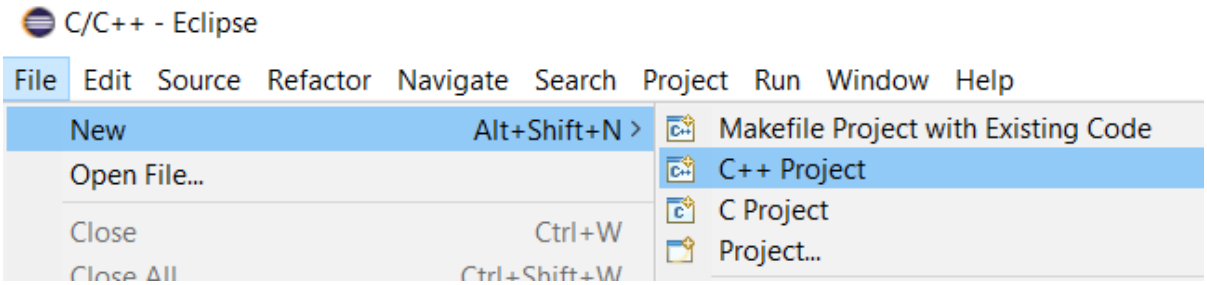


Figura 21. Crear proyecto C++ nuevo.

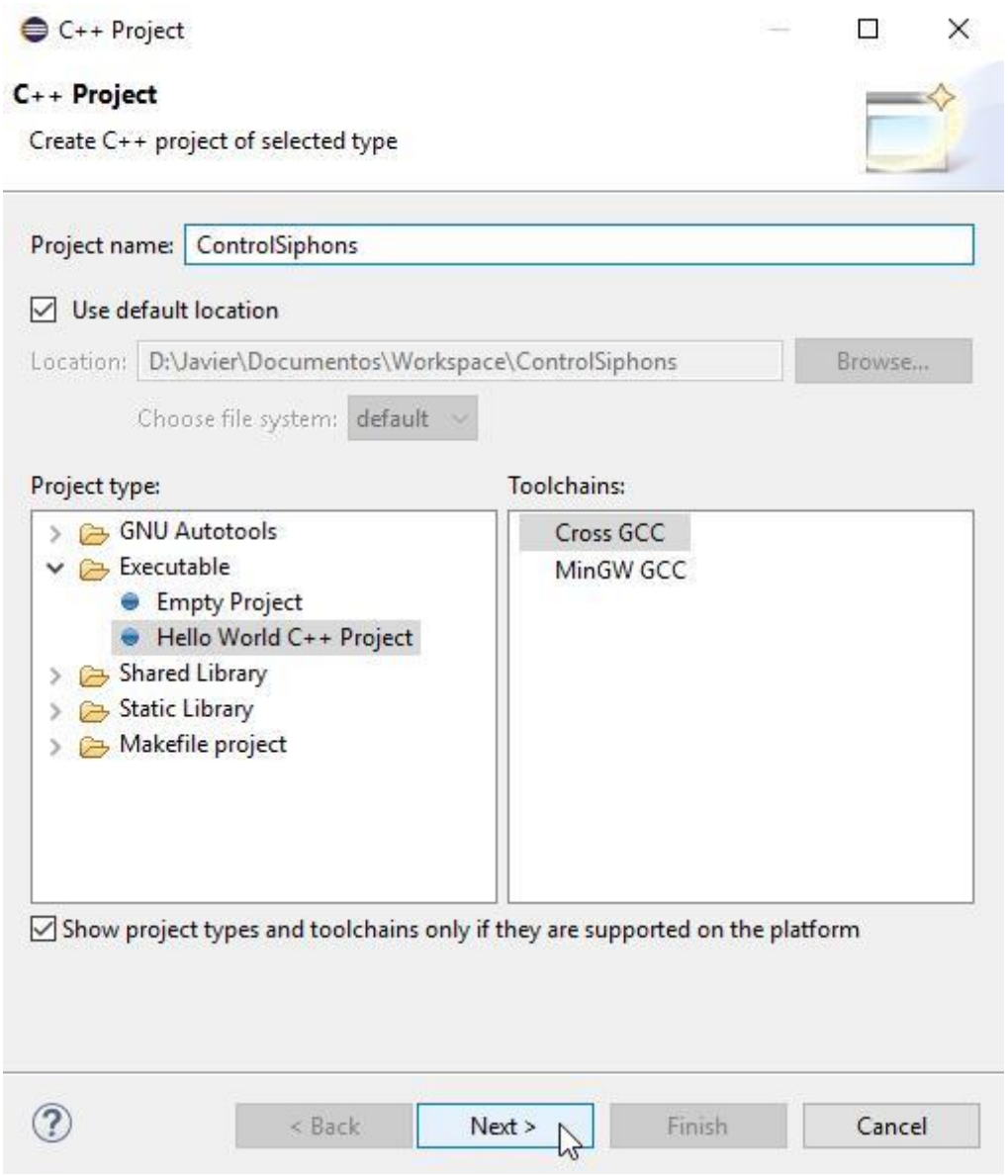


Figura 22. Crear nuevo proyecto.

A continuación se hace clic en *Next* hasta que aparezca el botón *Finish.* Y hacemos clic en *Finish.* Entonces el proyecto ya estará creado, ahora hay que añadir los archivos del trabajo en la carpeta *scr.*

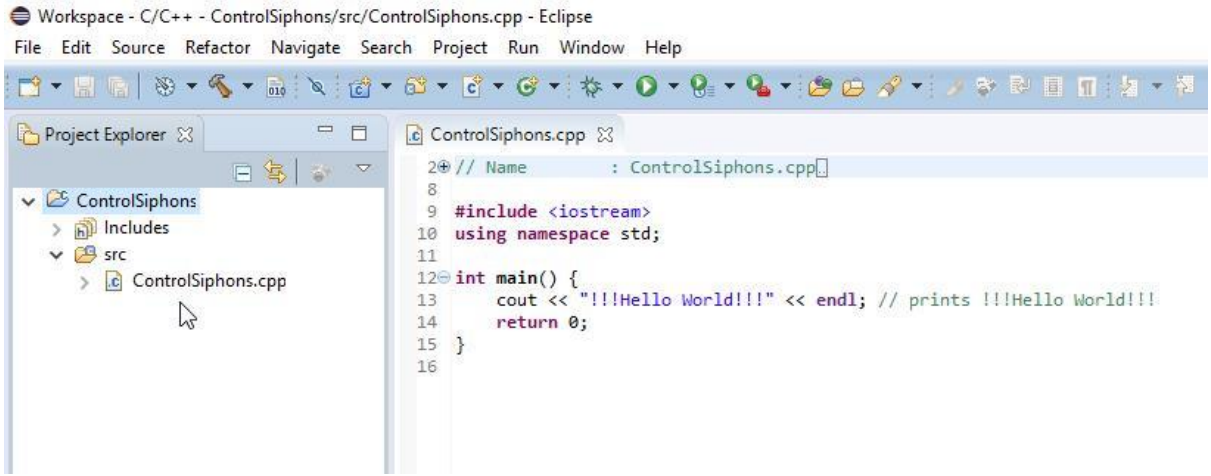


Figura 23. Aspecto del programa tras crear el proyecto C++ en Eclipse.

Una vez copiados los archivos del código (*ControlSiphons.cpp, petri.h, petri.cpp, matrix.h, matrix.cpp y setFunctions.h*) en la carpeta *scr*, obviamente sustituyendo el archivo *ControlSiphons.cpp* que se ha creado con el proyecto por el archivo *ControlSiphons.cpp* con el código, habrá que copiar también los archivos *.txt* con las redes de Petri de entrada. Y tras copiar añadir todos los archivos al proyecto, el proyecto queda así:

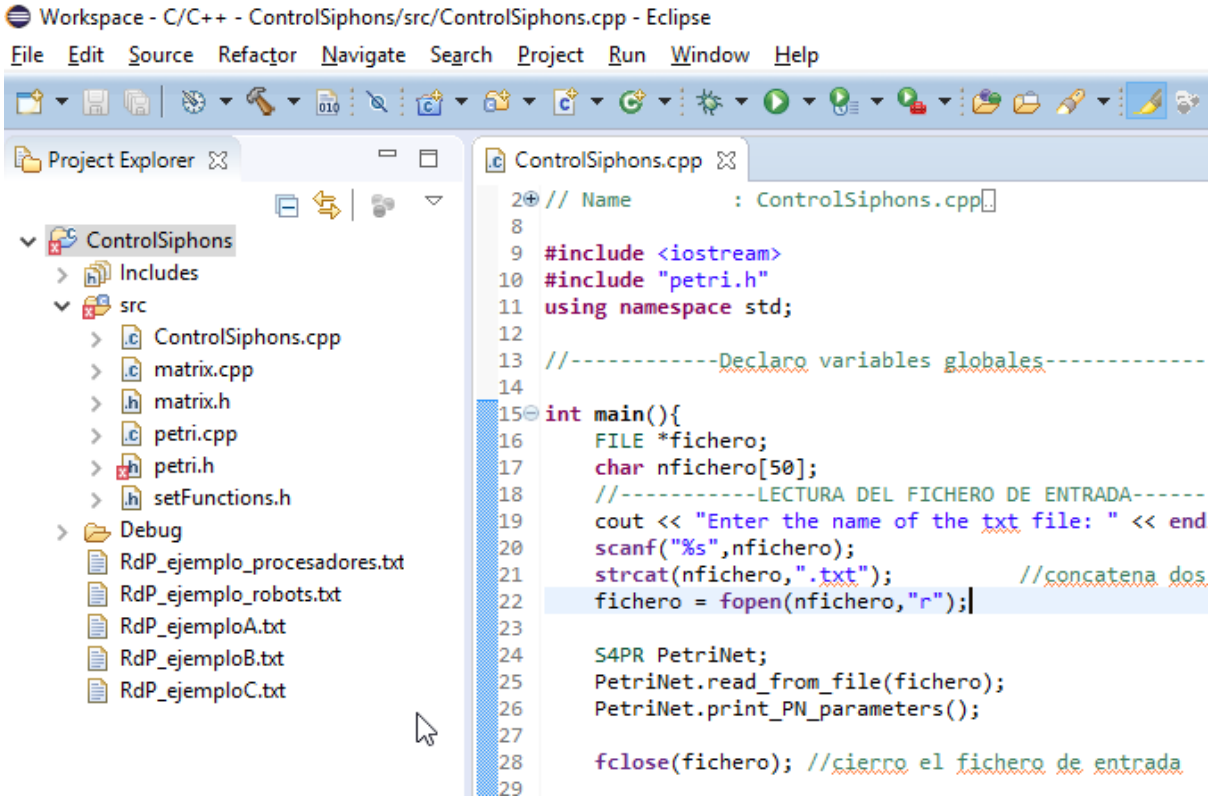


Figura 24. Archivos del proyecto: ControlSiphons.

Por último queda añadir la librería *boost graph*, para ello hay que descargar la librería de la web https://sourceforge.net/projects/boost/files/. Una vez descargado el archivo, hay que extraerlo, y copiar la carpeta *boost* en la ubicación deseada.

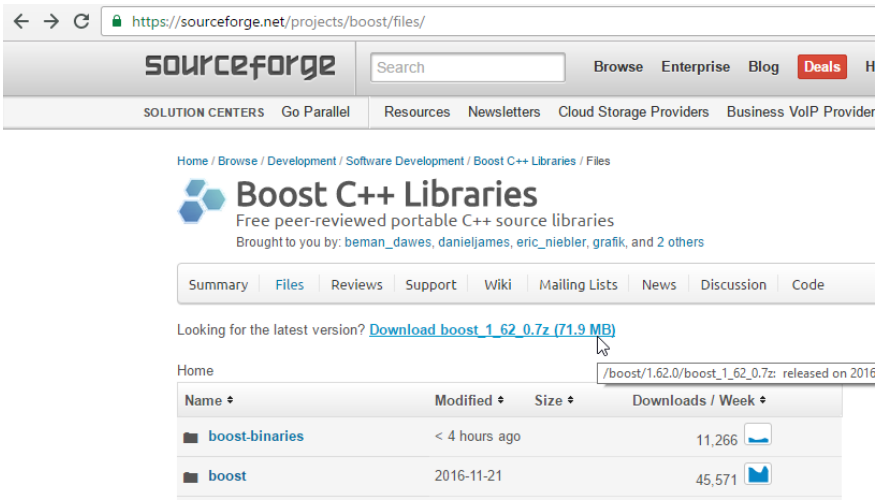


Figura 25. Descarga de la librería boost para tebajar con grafos.

A continuación hay que ir a Eclipse e ir al menú superior *Project → Properties* y aparecerá la ventana que se muestra en la *Figura 24* Allí hay que ir a *C/C++ Bluid → Settings → Gross GCC Compiler → Includes* y ahí en *Include paths (-I)* hacer clic en el icono con un símbolo '+' para añadir una nueva ruta de librería, donde añadiremos la ruta de donde se encuentre la carpeta *boost* que hemos descomprimido previamente. Después hay que repetir la misma operación y copiar también la misma ruta en los *include paths (-I)* de *C/C++ Bluid → Settings → Gross G++ Compiler → Includes.*

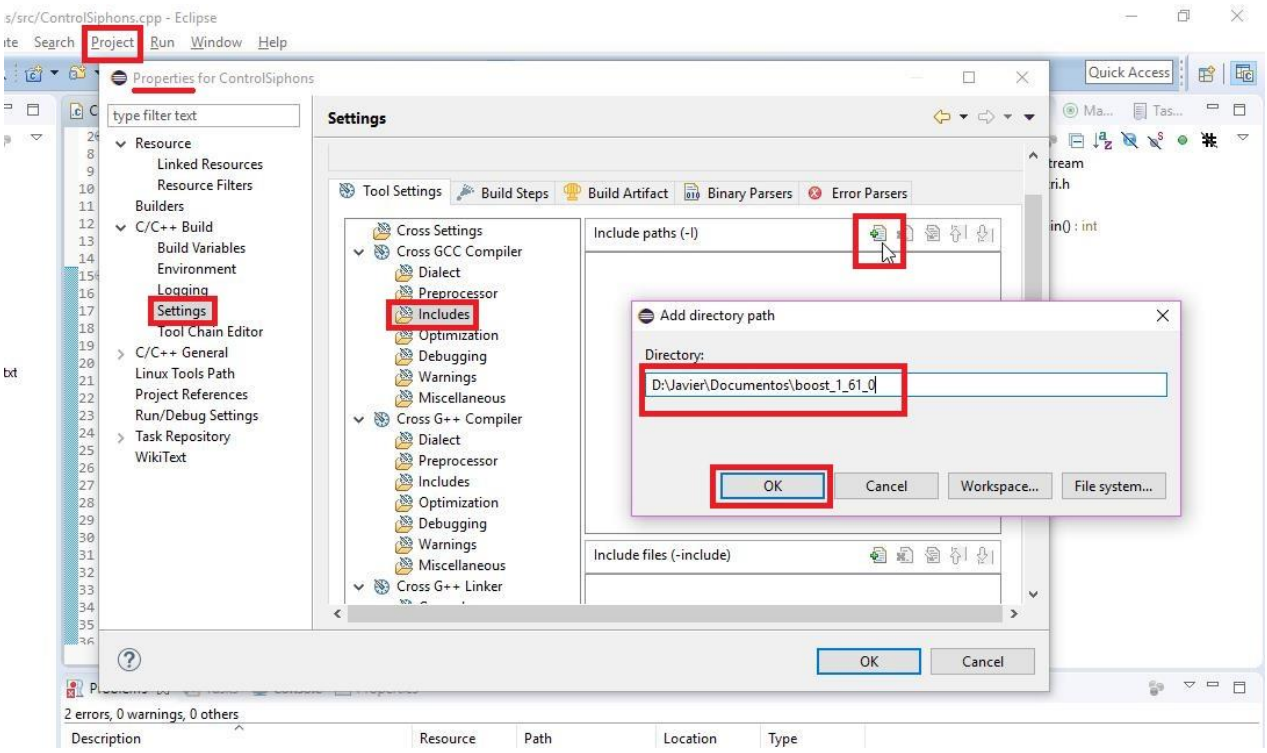


Figura 26. Insertar ubicación de la librería boost en los includes del proyecto.

Una vez copiadas las rutas de las librerías, se hace clic en *OK* y ya está listo el programa para ser compilado y ejecutado desde *Eclipse.* Para ello, hay que ir a *Project → Build All*, para compilar el programa y después hay que apretar el botón *Run*. El programa pedirá por pantalla al usuario que introduzca el nombre del fichero de entrada y a continuación se leerá la red de Petri del fichero indicado y se realizarán los cálculos.

ANEXO 5: EJEMPLOS DE CÁLCULO DE SIFONES EN DIFERENTES REDES S^{4}PR

En este anexo se van a mostrar varios ejemplos de redes de Petri de la subclase S4PR. Estos ejemplos no van a corresponder a ninguna abstracción proveniente de trayectorias de un sistema multi-robot, sino que tan solo van a ser redes de distintas tipologías, es decir, con distinto número de lugares, de procesos y de recursos compartidos.

La estructura en la que se mostrará cada ejemplo será la siguiente:

- Imagen de la red de Petri a analizar junto a una breve descripción de ella.
- Conjunto de sifones obtenidos mediante la herramienta *TimeNET*.
- Conjunto de sifones obtenidos mediante la función *getSiphons()* dentro del programa implementado en este trabajo.
- Comparación de los tiempos de cálculo de cada uno de los métodos.
- Conjunto de sifones obtenidos mediante la función *getBadSiphons()* dentro del programa implementado en este trabajo.

El código que se ha ejecutado para calcular los sifones mediante la función *getSiphons()* y *getBadSiphons()* en el programa principal es el siguiente:

```
#include <iostream>
#include "petri.h"
using namespace std;

int main(){
 FILE *fichero;
 char nfichero[50];
       cout << "Enter the name of the txt file: " << endl;
       scanf("%s",nfichero);
       strcat(nfichero,".txt");
      fichero = fopen(nfichero,"r");

 S4PR PetriNet;
 PetriNet.read_from_file(fichero);
 PetriNet.print_PN_parameters();
       fclose(fichero);

 clock_t t_start;
      t_start= clock();
 std::vector<std::set<int> > Siphons=PetriNet.getSiphons();
 double processing_time=((double)clock() - t_start) / CLOCKS_PER_SEC;
      cout<<"Execution time of the function getSiphons() --> ";
      cout<<processing_time<<" secs"<<endl;

 PetriNet.printSiphons();

 PetriNet.printBadSiphons();

    return 0;
}
```

Con estos ejemplos tan sólo se quiere demostrar que la función *getSiphons()* de la plataforma desarrollada en este proyecto supone una importante mejora frente a otras herramientas existentes para el cálculo de sifones de redes S4PR, en términos de velocidad de cómputo y requerimientos de memoria.

También con estos ejemplos se demuestra el funcionamiento para distintos tipos de redes independientemente del tamaño de las mismas.

A5.1. EJEMPLO A

Comenzamos con un ejemplo de una red sencilla con dos procesos paralelos concurrentes y además añadimos recursos limitados.

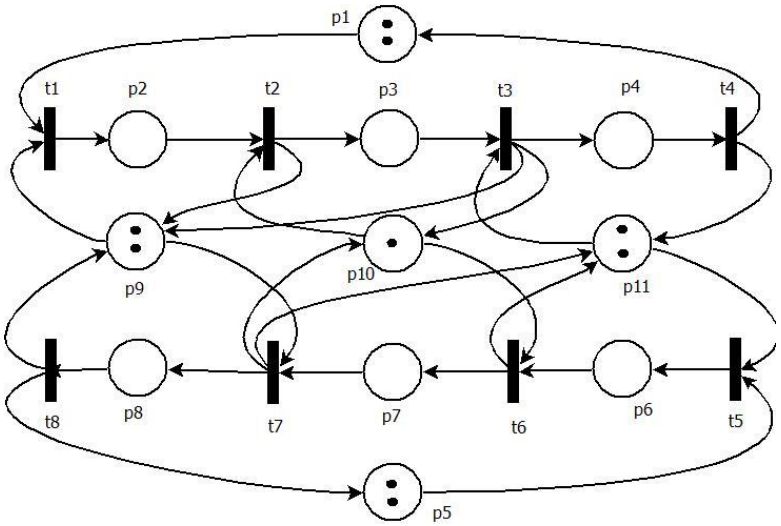


Figura 27. Red de Petri Ejemplo A con 9 sifones.

Los sifones mínimos de esta red obtenidos mediante TimeNET y mediante el algoritmo del grafo de poda implementado en este trabajo, son los mismos y son los siguientes:

{P4, P6, P7, P11}
{P3, P7, P10}
{P2, P3, P8, P9}
{P4, P8, P9, P10, P11}
{P4, P7, P10, P11}
{P3, P8, P9, P10}
{P2, P4, P6, P8, P9, P11}
{P1, P2, P3, P4}
{P5, P6, P7, P8}.

Siendo los dos últimos sifones los que corresponden a los P-semiflujos correspondientes a cada uno de los dos procesos de la red. El tiempo de cálculo con TimeNET es de 0,03 segundos y con la función *getSiphons()* del programa desarrollado en este trabajo es de 0,04 segundos. Por último, el único sifón malo de la red se ha calculado con la función *getBadSiphons()* y es {P4, P8, P9, P10, P11}.

A5.2. EJEMPLO B

La siguiente red es más compleja y tiene bastantes más lugares, consta de 4 procesos (rodeados en gris) y de 22 lugares, de los cuales, 5 son recursos compartidos.

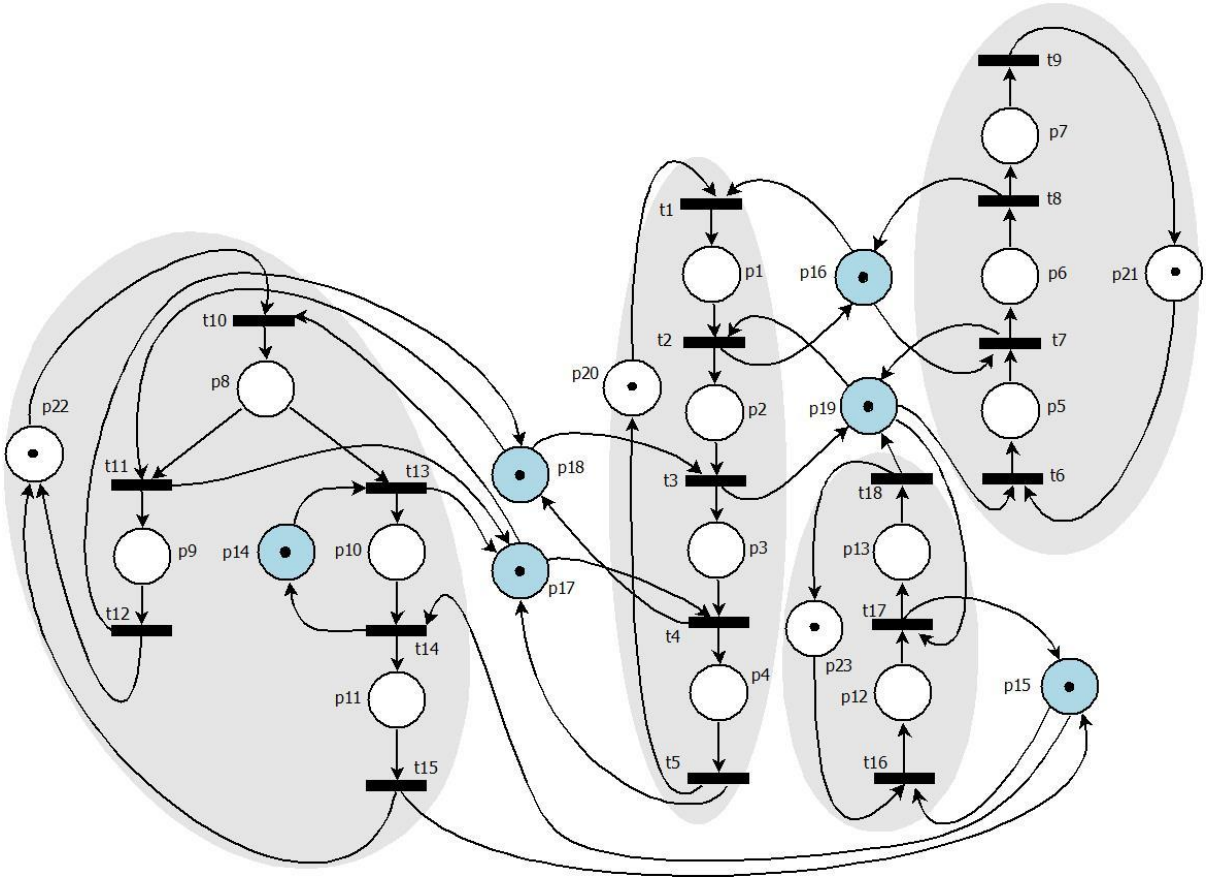


Figura 28.Red de Petri del ejemplo B con 4 procesos y 6 recursos

En este caso para el cálculo de los sifones mínimos con TimeNET surgen problemas ya que la herramienta se bloqueo y finalmente no devuelve todos los sifones mínimos, sólo devuelve algunos de ellos. Y el tiempo de cómputo tarda 0,80 segundos. Sin embargo con la función *getSiphons()* se tarda tan sólo 0,046 segundos en el cómputo de los sifones, y se obtienen todos los sifones mínimos de la red y éstos son los 10 siguientes:

{P10 P14}
{P7 P11 P12 P15}
{P1 P6 P16}
{P4 P8 P17}
{P2 P3 P9 P18}
{P2 P5 P13 P19}
{P2 P7 P11 P13 P15 P16 P19}
{P2 P6 P13 P16 P19}
{P4 P5 P7 P9 P11 P13 P14 P15 P17 P18 P19}
{P4 P7 P9 P11 P13 P14 P15 P16 P17 P18 P19}

Los sifones que corresponden a los p-semiflujos son: {P1 P2 P3 P4 P20}, {P5 P6 P7 P21}, {P8 P9 P10 P11 P22} y {P12 P13 P23} y éstos no los calcula getSiphons() ya que son bastante evidentes y son los lugares correspondientes a los 4 procesos rodeados en gris en la *Figura 28.*

Por último los sifones malos de la red calculados con la función *getBadSiphons()* son:

{4,7,9,11,13,14,15,16,17,18,19}
{2,7,11,13,15,16,19}
{2,6,13,16,19}
{4,5,7,9,11,13,14,15,17,18,19}

Cabe destacar que como es una red no muy compleja y cada recurso es usado como máximo por dos o tres procesos, el tiempo de cálculo con TimeNET no es muy superior al tiempo empleado por el programa desarrollado en este proyecto, pero a continuación veremos en los siguientes ejemplos que conforme aumentamos el número de lugares de la red, el número de lugares de recurso y sobre todo, la cantidad de lugares que utilizan cada recurso, el tiempo de cálculo empleado por TimeNET para el cálculo de los sifones se dispara.

A5.3. EJEMPLO C

A continuación se presenta una red de Petri S4PR con 4 procesos y 43 lugares, de los cuales 3 son recursos compartidos (ver *Figura 29*).

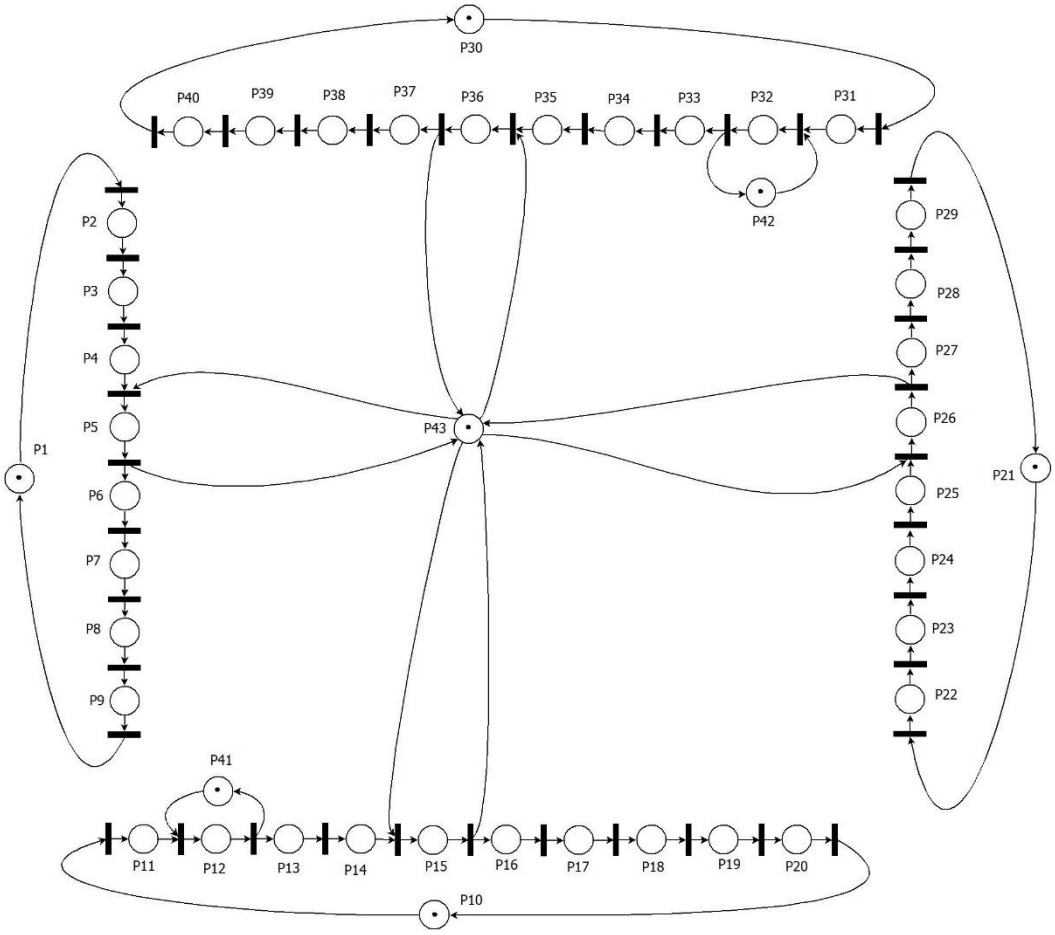


Figura 29. Red de Petri del ejemplo C.

Los sifones mínimos de la red son los que se muestran a continuación:

{P12, P41}
{P32, P42}
{P5, P15, P26, P36, P43}

Los sifones correspondientes a los 4 procesos de la red son:

{P1, P2, P3, P4, P5, P6, P7, P8, P9}

{P21, P22, P23, P24, P25, P26, P27, P28, P29}

{P10, P11, P12, P13, P14, P15, P16, P17, P18, P19, P20}

{P30, P31, P32, P33, P34, P35, P36, P37, P38, P39, P40}

En este caso la herramienta TimeNET tarda más que en los casos anteriores pese a tener tan sólo tres recursos, concretamente tarda 15,50 segundos. Esto se debe a que uno de los recursos (P43) está siendo usado por cuatro procesos y aunque haya menos recursos que en los casos anteriores, lleva más tiempo calcular un sifón si su recurso está siendo usado por muchos procesos. Sin embargo con el programa desarrollado en este trabajo, el tiempo utilizado en el cómputo de los sifones no es tan elevado, es de tan solo 1,40 segundos.

Cabe destacar que la presencia de recursos altamente utilizados influye más que el número de lugares totales en el tiempo requerido para el cálculo de sifones con TimeNET. Tanto es así que si tomamos la misma red que en la *Figura 29* y le añadimos un recurso más utilizado por los cuatro procesos (obteniendo la red de la *Figura* 30), el tiempo de cómputo aumenta drásticamente hasta los 237,56 segundos, aumentando también la cantidad de memoria requerida.

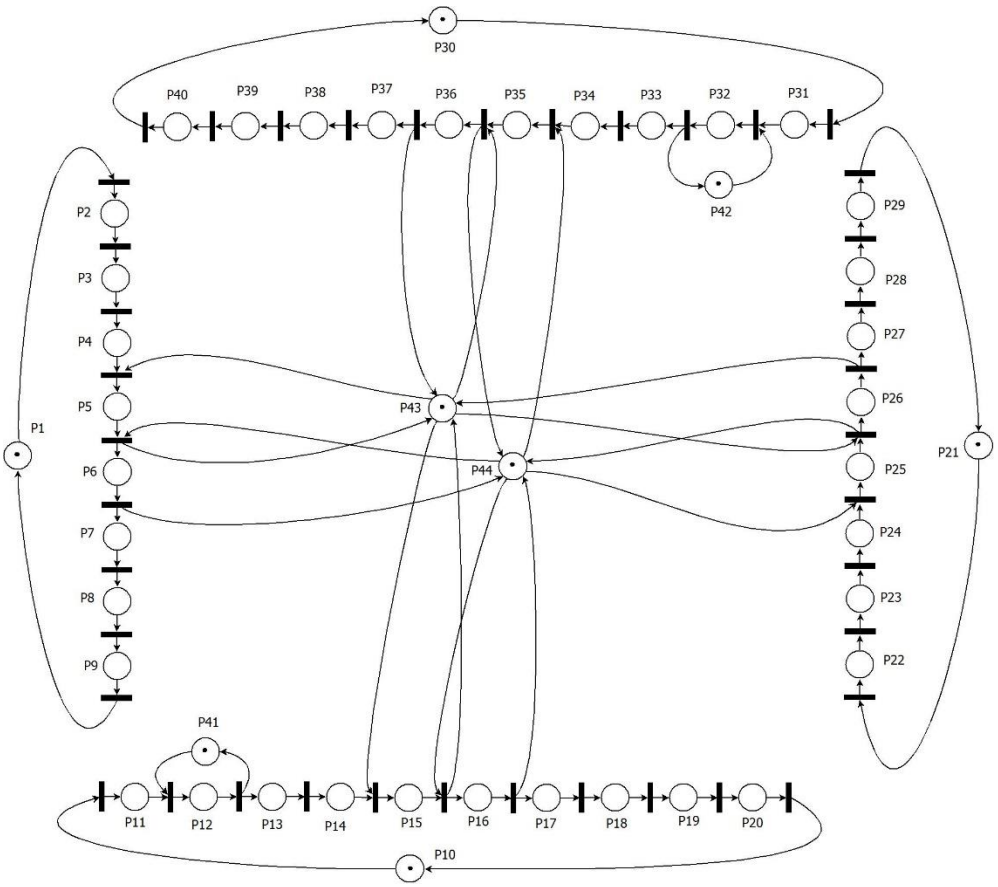


Figura 30. Red del ejemplo C con un recurso extra P44.

A5.4. EJEMPLO D

La siguiente red es más compleja y tiene bastantes más lugares, consta de 4 procesos y de 50 lugares, de los cuales, 10 son recursos compartidos (ver *Figura 31*).

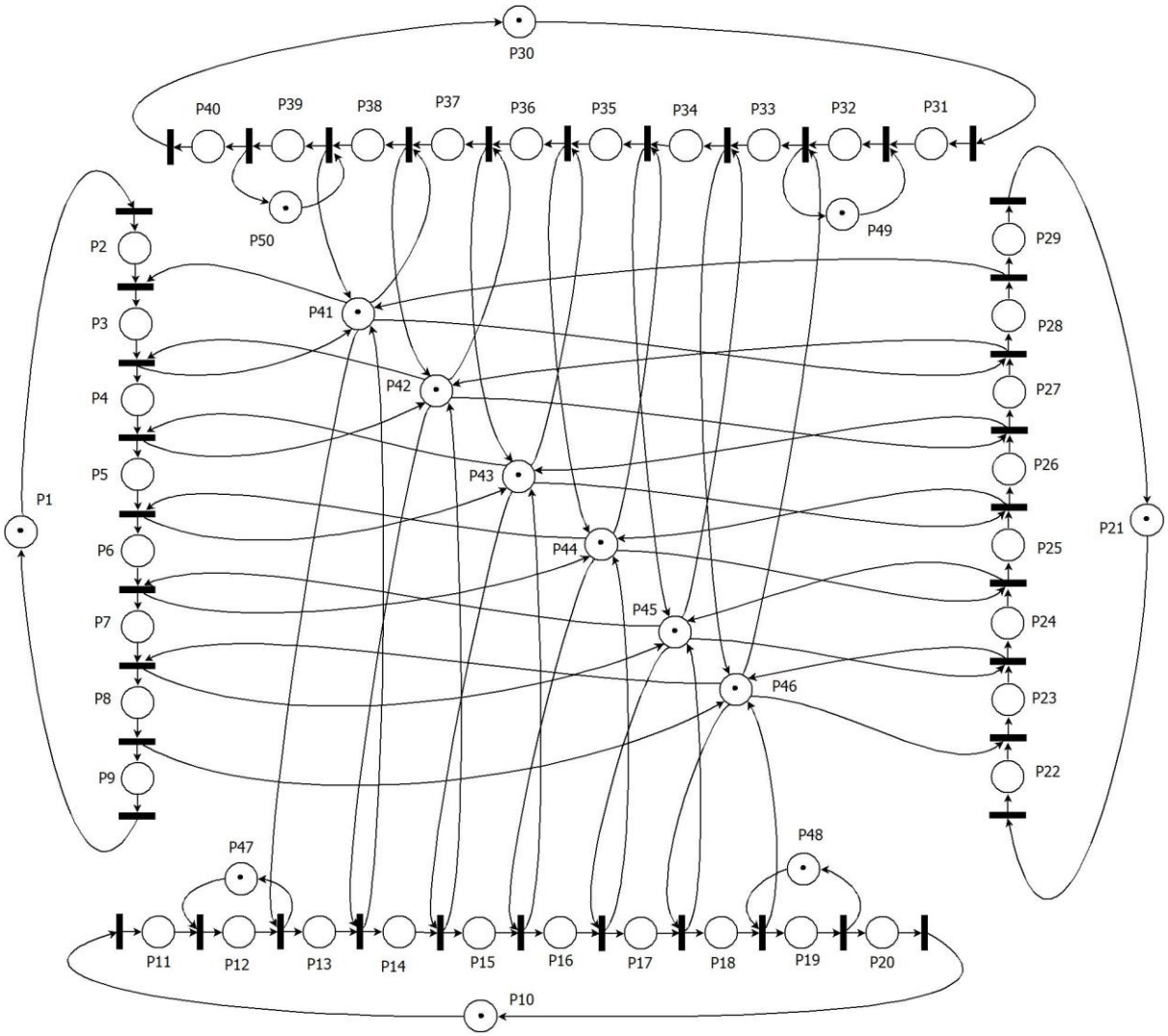


Figura 31. Red de Petri del Ejemplo D

Para esta red, debido a su dimensión mayor que las anteriores, con la herramienta TimeNET no se ha podido realizar el cálculo de los sifones ya que tras tardar varios minutos, se supera la memoria disponible y finalmente el programa devuelve:

```
RESIZEROWS: no more memory available!
RESIZEROWS: no more memory available!
RESIZEROWS: no more memory available!
Error! an error occured
Removing temporary files
Calculation of Siphons:
Time passed with computation:  5055.70 s
 There are no siphons
```

Por otro lado, con la herramienta desarrollada en este proyecto, al utilizar la función *getSiphons()*, es cierto que tarda más tiempo que en los casos anteriores, pero se obtiene un resultado bueno y el tiempo que tarda en el cálculo es de 8,86 segundos. Y los 22 sifones mínimos de la red obtenidos son:

{3,13,28,38,41}
{4,14,27,37,42}
{5,15,26,36,43}
{6,16,25,35,44}
{7,17,24,34,45}
{8,18,23,33,46}
{12,47}
{19,48}
{32,49}
{39,50}
{8,18,28,38,41,42,43,44,45,46}
{8,18,27,37,42,43,44,45,46}
{8,18,26,36,43,44,45,46}
{8,18,25,35,44,45,46}
{8,18,24,34,45,46}
{6,16,28,38,41,42,43,44}
{7,17,28,38,41,42,43,44,45}
{6,16,27,37,42,43,44}
{7,17,27,37,42,43,44,45}
{6,16,26,36,43,44}
{7,17,26,36,43,44,45}
{7,17,25,35,44,45}

Faltan los sifones correspondientes a los 4 procesos de la red y que ya hemos visto que nos son calculados por este programa, pero sabemos que éstos son: {1, 2, 3, 4, 5, 6, 7, 8, 9}, {10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20}, {21, 22, 23, 24, 25, 26, 27, 28, 29} y {30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40}.

Y al ejecutar la función getBadSiphons() se obtienen los siguientes 12 sifones malos:

{8,18,28,38,41,42,43,44,45,46}
{8,18,27,37,42,43,44,45,46}
{8,18,26,36,43,44,45,46}
{8,18,25,35,44,45,46}
{8,18,24,34,45,46}
{6,16,28,38,41,42,43,44}
{7,17,28,38,41,42,43,44,45}
{6,16,27,37,42,43,44}
{7,17,27,37,42,43,44,45}
{6,16,26,36,43,44}
{7,17,26,36,43,44,45}
{7,17,25,35,44,45}