Universidad Zaragoza
1542

Trabajo Fin de Grado

Planificación del movimiento basada en diagramas de Voronoi para equipos de robots móviles

Path planning based on Voronoi diagrams in multirobot systems

Autor:

José Luis Calvo Subirá

Director:

Cristian Mahulea
(Departamento de informática e ingeniería de sistemas)

Escuela de Ingeniería y Arquitectura
Febrero 2019

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. José Luis Calvo Subirá,

con nº de DNI 77134574A 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) Planificación del movimiento basada en diagramas de Voronoi para equipos de robots móviles,

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

Zaragoza, 01/02/2019

Fdo: José Luis Calvo Subirá

RESUMEN

PLANIFICACIÓN DEL MOVIMIENTO BASADA EN DIAGRAMAS DE VORONOI PARA EQUIPOS DE ROBOTS MÓVILES

En este proyecto se considera una estrategia que supervisa la planificación de trayectorias y el control del movimiento de un conjunto de robots que deben ejecutar sus tareas en el mismo entorno. El problema de la navegación de cada uno de los robots se soluciona mediante la abstracción de la situación inicial en una representación basada en sistemas de eventos discretos, y la búsqueda de la solución óptima en un grafo. En este proyecto se asume navegación clásica, en la que cada robot debe alcanzar una configuración final desde una configuración inicial evitando obstáculos.

En primer lugar, se calcula el diagrama de Voronoi del entorno compuesto por los obstáculos y sus límites de contorno. Posteriormente, debido a que este diagrama consiste en un conjunto de curvas continuas, se discretiza considerando solamente algunos puntos del diagrama por razones que se detallan a lo largo del trabajo. Por último, se aplica el algoritmo de Dijkstra en el grafo obtenido a partir de la discretización anterior, con el objetivo de calcular las trayectorias óptimas para cada robot.

Respecto al control del movimiento del equipo de robots, la solución implica detener a determinados robots cuando las regiones por las que deben circular, cuya capacidad está limitada en cuanto al número de robots que pueden ocuparlas al mismo tiempo, están siendo transitadas por otros, de forma que se eviten colisiones. Esto se consigue mediante la construcción de una red de Petri que modela las trayectorias del conjunto de robots usando recursos compartidos, que restringen las capacidades de las regiones críticas (aquellas donde las trayectorias de dos o más robots coinciden, y por lo tanto existe probabilidad de colisión), y técnicas que evitan el bloqueo de la red durante su evolución.

ÍNDICE

1. INTRODUCCIÓN

1.1 OBJETIVOS Y ALCANCE

El objetivo de este TFG consiste en desarrollar un método por el que un conjunto de robots móviles $R = \{R_1, R_2, \dots R_n\}$ puedan trasladarse en un espacio de trabajo rectangular$W$ dado, desde una configuración inicial (q_{init}^{Ri}) hasta una configuración objetivo (q_{goal}^{Ri}), en condiciones de total seguridad:

1. sin colisionar con los obstáculos contenidos en W,
2. sin salirse de los límites de W, y
3. sin colisionar con otros robots $q_t^{Ri} \neq q_t^{Rj}, \forall\, i \neq j\,, R_i, R_j \in R, t \in [init, \dots, goal]$.

Las trayectorias se obtendrán de manera independiente para cada robot móvil mediante un algoritmo de planificación a partir del espacio de trabajo W. La evitación de colisiones se consigue implementando un controlador centralizado que genera modos de espera: determinados robots se detienen en ciertos lugares con el fin de permitir el paso por el siguiente lugar de su trayectoria a otro robot. No obstante, una consecuencia negativa es que tales modos de espera pueden ocasionar bloqueos, por lo que es necesario que el controlador también sea capaz de evitar las situaciones en las que se producirían estos bloqueos.

El método se implementará en C++, haciendo uso de una programación orientada a objetos, y se incluirá en una plataforma multi-robot, producto de las aportaciones de distintos alumnos. En ella se realizarán los experimentos que corroboren el correcto funcionamiento del método. A lo largo del proyecto, se han diseñado e implementado ciertas funciones[1] que contribuyen al desarrollo del método objetivo, las cuales aparecen nombradas en la siguiente lista de acuerdo con su área de implicación, bien sea en la parte de planificación de trayectorias, bien en el controlador centralizado.

PLANIFICACIÓN DE TRAYECTORIAS

- Point.h/Point.cpp:
  - Class *Point2D*
- Segment.h/Segment.cpp:
  - Class *Segment*
- obstacles_filter.cpp:
  - obstacles_filter
- Process.cpp:
  - nearest_vertex
  - check_vertex
  - processVoronoi
- VoronoiDiagram.cpp:
  - check_double_solution
  - intersect_with_inputgeometry
  - inside_obstacle
  - create_segments
  - v_diagram
- Graph.h/Graph.cpp:
  - Class *Graph*:
    - Graph
    - set_weights
    - check_continuity
    - getGraph
    - print_nodes
- Dijkstra.cpp:
  - dijkstra

EVITACIÓN DE COLISIONES Y PREVENCIÓN DE BLOQUEOS

- Petri_Net.h/Petri_Net.cpp:
  - Class Input_S4PR:
    - Input_S4PR
    - r_not_created
    - set_to_null_matrix
    - add_minus_1
    - is_idle
    - get_nprocesses
    - get_dimP
    - get_dimT
    - get_Pre
    - get_Post
    - get_processes
    - get_m_0
    - get_resources
    - printParameters
- control.cpp:
  - shared_resource
  - resource_available
  - search_in_matrix
  - pick_token
  - return_token
  - main ()

Figura 1. Lista de funciones diseñadas e implementadas durante el desarrollo del proyecto.

[1]Se ha creado un repositorio en GitHub para ofrecer acceso a todos los ficheros descritos y a las funciones definidas, así como a la implementación del método desarrollado: https://github.com/jlcalvo/PathPlanning-CollisionAvoidance

1.2 FASES

A continuación, se detallan las fases que este proyecto ha atravesado:

1) La primera parte del trabajo consiste en un breve estudio de los componentes de la plataforma multi-robot, que define el espacio de trabajo de los experimentos que se realizarán. Especialmente, se propone comprender la información que se va a usar como entrada en el método a diseñar.
2) En segundo lugar, se trata la planificación de trayectorias, responsable de garantizar las condiciones de seguridad (1) y (2) descritas en el objetivo del proyecto. En este punto se considera únicamente la existencia de un robot, ya que en el caso multi-robot se calculan las trayectorias de forma independiente.
    A. Se hace un breve repaso a los algoritmos de planificación existentes, divididos en dos grandes clasificaciones: los algoritmos exactos y los basados en muestreo.
    B. Se profundiza en el algoritmo basado en el diagrama de Voronoi, el elegido para proceder con el cálculo de las trayectorias.
    C. Con el objetivo de generar una representación discretizada del espacio de trabajo, se crea un grafo representativo de la red de trayectorias, a partir del cual se puede obtener el conjunto de trayectorias posibles desde la configuración inicial q_{init}^{Ri} hasta la configuración final q_{goal}^{Ri}, ambas representadas por nodos en el grafo.
    D. Sin embargo, de todas las trayectorias posibles se quiere hallar la óptima, para lo que se requiere el uso de algoritmos de búsqueda en grafos.
3) Una vez concluida esta parte del proyecto, se realizan experimentos en la plataforma para comprobar el correcto funcionamiento de la obtención de la trayectoria óptima con un único robot, asegurando el cumplimiento de las condiciones de seguridad (1) y (2).
4) A continuación, se trata la metodología empleada para resolver la evitación de colisiones (condición (3)), que consiste en la abstracción de las trayectorias de los robots en una red de Petri S4PR, y el correspondiente diseño e implementación de un controlador centralizado que además asegure la prevención de bloqueos.
5) Por último, se llevan a cabo experimentos en la plataforma con la participación de dos robots (sistema multi-robot) para valorar el funcionamiento del controlador.

1.3 POSIBLES APLICACIONES

A parte de las bien conocidas aplicaciones en el área de la navegación en robots (controladores autónomos aéreos y otros sistemas autónomos como el *self-driving car*), la planificación de trayectorias con un método integrado de evitación de colisiones tiene otras áreas de aplicación útiles.

VIDEOJUEGOS

En juegos de ordenador y animaciones en general determinados objetos animados han de moverse y encontrar rutas a destinos específicos evitando objetos fijos. En algunos casos, la solución propuesta consiste en proporcionar secuencias predefinidas basadas en puntos de referencia. El problema se complica todavía más cuando estos objetos animados han de

moverse en grupos, comportándose en particular como un grupo real, evitando colisiones entre ellos [1].

Figura 2. Ejemplo de objetos animados pertenecientes a un videojuego, que han de comportarse como un grupo.

INSPECCIÓN Y VIGILANCIA

La vigilancia por robots puede consistir en una estación remota fija que despliega un equipo de robots cuya misión es explorar un espacio determinado y detectar objetos (intrusos, que no forman parte de la configuración inicial del espacio), informando a la estación central de dichas detecciones. Si estos intrusos se suponen fijos, cada robot debe ser enviado a una región diferente, a la cual debe llegar a través del camino más corto (para reducir el tiempo de llegada), y evitando colisiones con obstáculos y otros robots. Si los intrusos se suponen móviles, el problema se complica. En este caso se debe buscar una solución que cubra toda el área disponible del espacio.

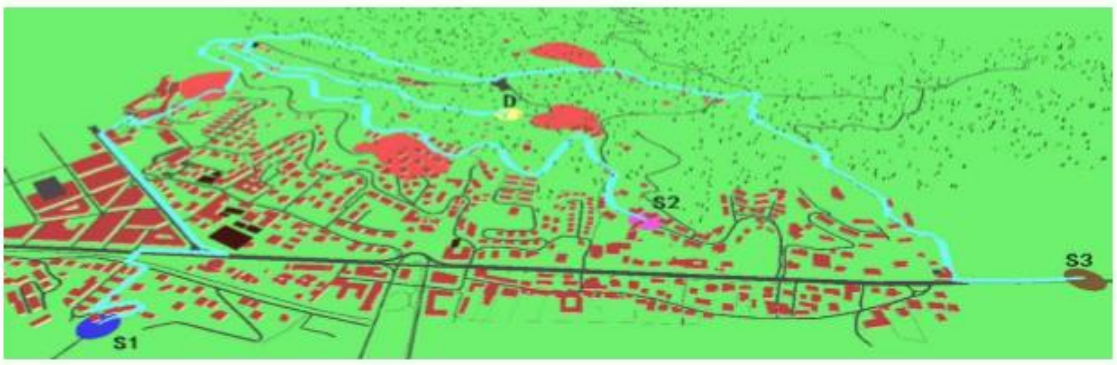

Figura 3. Aplicación de planificación de trayectorias en vigilancia

CIRUGÍA ROBÓTICA

Hoy en día existen numerosas investigaciones para desarrollar robots-asistentes en el campo de la cirugía médica. Algunas tareas repetitivas, como por ejemplo los puntos de sutura o cortes mediante bisturí, son fácilmente realizables por un robot. Para ello es esencial obtener la trayectoria óptima evitando cualquier tipo de colisión. No obstante, existe una dificultad adicional a tener en cuenta, y es que se trata de entornos dinámicos.

Figura 4. Robot asistente

2. PLATAFORMA DE ROBOTS

Antes de comenzar a explicar el algoritmo de planificación con el método de evitación de colisiones, es preciso describir el sistema en el que se implementará dicho método para proceder con los experimentos. Por lo tanto, se analizarán a continuación las partes principales que lo componen, la salida del sistema y la comunicación con los robots.

2.1 PARTES PRINCIPALES

La plataforma está compuesta por cuatro partes principalmente:

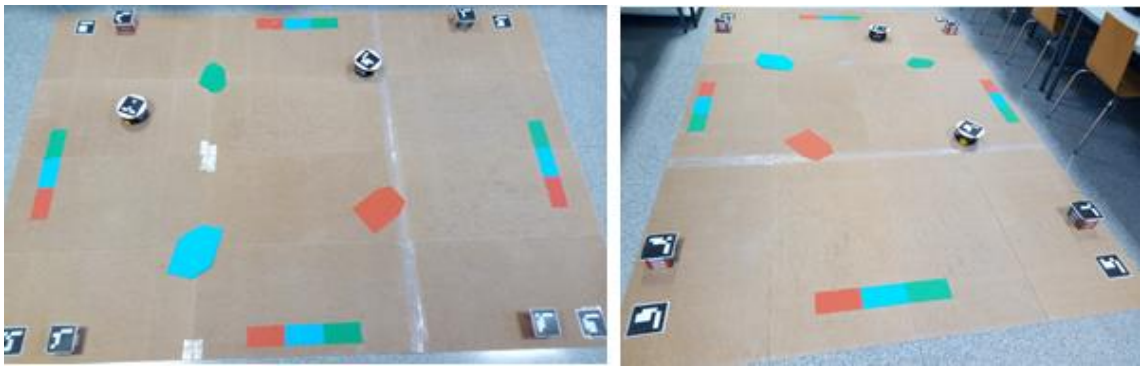

Figura 5. Plataforma multi-robot actual con todos sus componentes.

a) Espacio de trabajo. Se define como el espacio físico que contiene los obstáculos y los robots, de forma geométrica rectangular con unas dimensiones totales de 332 x 234 centímetros. Contiene además un total de 8 marcadores: 4 de ellos situados en las esquinas al nivel del suelo cuyo objetivo es permitir que sistema de visión reconozca los límites del entorno; mientras que los 4 marcadores restantes están situados a una altura igual a la del robot (10 cm), de modo que el sistema de visión pueda localizar la posición de los robots.

En cada lado del rectángulo que forma el espacio de trabajo se disponen tres regiones rectangulares cada una de un color distinto (azul, verde, rojo), las cuales representan los colores que el sistema de visión reconocerá como obstáculos.

b) Robots. Los robots utilizados en los experimentos se han diseñado utilizando un chasis *Turtle 2WD Mobile Platform* y una placa RoMeo V2[R3] [2] con un controlador Arduino Leonardo. Se han utilizado dos robots que portan en la parte superior un marcador que permite al sistema de visión identificarlos y localizar su posición.

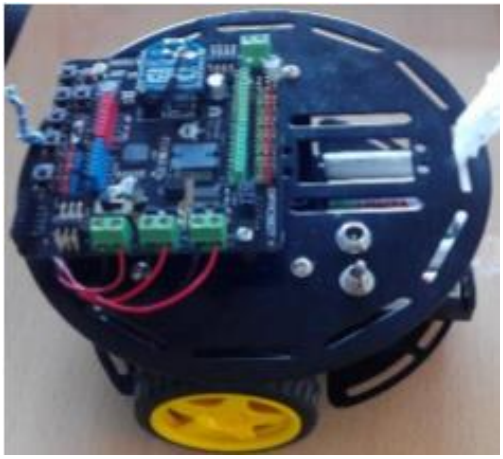

Figura 6. Robot móvil Arduino

c) Obstáculos. Consisten en trozos de cartulina de forma poligonal con sus segmentos lo más rectos posibles (para favorecer su identificación por parte del sistema de visión) y del mismo color que uno de los que representan las regiones rectangulares.

d) Sistema de visión. El sistema de visión consiste en una cámara situada en un plano cenital a una altura aproximada de 3.5 metros, que graba el espacio de trabajo, así como su contenido (marcadores, obstáculos, robots y regiones rectangulares). Estas imágenes son enviadas al ordenador central, el cual reconstruye el entorno de manera virtual haciendo uso de las librerías OpenCV y ArUco. De este modo, el sistema es capaz de localizar los obstáculos, además de reconocer tanto la posición como la orientación de los robots en cada instante.

Para un funcionamiento correcto del sistema de visión, es necesario proporcionar todas las medidas geométricas del espacio de trabajo, por ejemplo: la dimensión total del espacio, la posición y las dimensiones de los marcadores, la posición de las regiones rectangulares que permiten la identificación de obstáculos, etc.

2.2 SALIDA DEL SISTEMA DE VISIÓN

Aunque no es importante entrar en detalle en el funcionamiento interno del sistema de visión, sí lo es su salida, puesto que ésta representa la entrada del trabajo realizado en este TFG.

El sistema de visión proporciona la siguiente información, con el origen del sistema de coordenadas 2D en la esquina inferior izquierda del espacio de trabajo.

- Posición (coordenadas del punto central del robot) y orientación (en radianes) de los robots (círculo azul en el dibujo).

- Posición de los obstáculos (polígonos rojos en el dibujo), en particular, las coordenadas de cada una de las esquinas de cada obstáculo.

En cuanto a la orientación, representada por la línea recta de color negro en el interior del robot, se considera 0 si tiene misma dirección y sentido que el eje X.

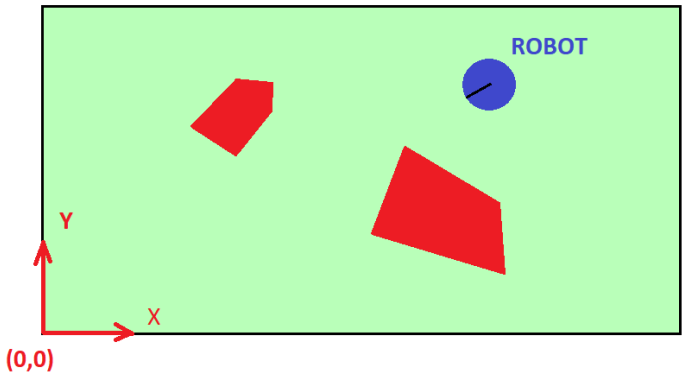

Figura 7. Interpretación gráfica de la salida del sistema de visión.

La siguiente imagen muestra un ejemplo en detalle del tipo de información que proporciona el sistema de visión:

```
Robot[1](x,y,theta):[0.559467, 0.176463, -0.604264]
Robot[2](x,y,theta):[0.67671, 0.815171, 1.34377]

obstáculo[1] esquina[1]: [0.740175, 0]
obstáculo[1] esquina[2]: [0.944732, 0]
obstáculo[1] esquina[3]: [0.742965, 0]
obstáculo[1] esquina[4]: [0.946831, 0]
obstáculo[2] esquina[1]: [0.665548, 0.309521]
obstáculo[2] esquina[2]: [0.720425, 0.444334]
obstáculo[2] esquina[3]: [0.861858, 0.175407]
obstáculo[2] esquina[4]: [0.719949, 0.222388]
obstáculo[2] esquina[5]: [0.891627, 0.237368]
obstáculo[2] esquina[6]: [0.853465, 0.409748]
obstáculo[3] esquina[1]: [1.586, 0.606365]
obstáculo[3] esquina[2]: [1.586, 0.604729]
obstáculo[3] esquina[3]: [1.586, 0.392205]
obstáculo[3] esquina[4]: [1.586, 0.378858]
```

Figura 8. Salida del sistema de visión. El sistema ha detectado dos robots R[1] y R[2], y tres obstáculos, con 4,6 y 4 esquinas respectivamente.

2.3 COMUNICACIÓN CON LOS ROBOTS

El hardware usado para la transmisión de información entre el ordenador y los robots es el XBee Antenna. Este hardware está conectado al ordenador central a través de un USB, usando otro hardware (Waspmote Gateway) para configurar la red de comunicación, creando así una red punto-multipunto para establecer la comunicación.

Hay dos tipos de información que deben ser mandadas desde el ordenador central a los robots:

1. - La trayectoria de cada robot.
2. - La posición y la orientación de cada robot recogidas por la cámara en cada momento.

El cálculo de la información tipo 1 es el objetivo de la primera parte del trabajo desarrollado en este proyecto.

Dado que la estructura de los paquetes de información no ha sido configurada en este TFG, para más información, se recomienda recurrir a [3].

2.4 MEJORAS DE LA PLATAFORMA DEL LABORATORIO

Originalmente la base del espacio de trabajo consistía en un rectángulo de papel de dimensiones 2,2 x 1,6 metros, tal y como muestra la Figura 9. Durante el desarrollo de este TFG se han llevado a cabo varios cambios, los cuales se explican a continuación:

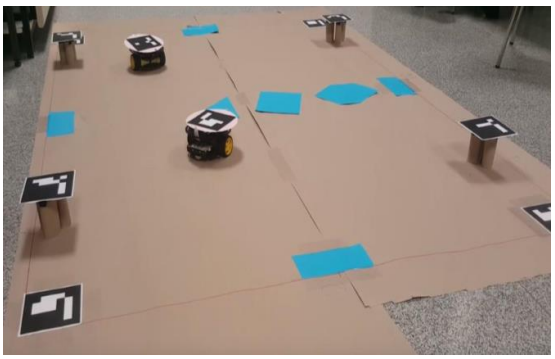

Figura 9. Plataforma multi-robot original.

- Se ha cambiado el material de la base por madera (traseros). Esto supone un aumento de la resistencia de la base y de su fiabilidad, ya que su deterioro es menos probable al guardarse e incluso durante su utilización.
- Se han aumentado las dimensiones de la plataforma, siendo ahora de 332 x 234 centímetros, lo que permite realizar experimentos con mayor número de obstáculos en el espacio de trabajo, y comprobar de este modo el funcionamiento de la implementación en situaciones de mayor complejidad con más robots actuando.
- Se han reemplazado las cartulinas (regiones rectangulares) que permiten al sistema de visión reconocer los obstáculos. En particular se ha reemplazado el color amarillo por el verde: debido a la tonalidad semejante entre la base de la plataforma y el color amarillo, el sistema de visión identificaba en ocasiones obstáculos inexistentes compuestos por una única esquina, esto es, por un único punto.

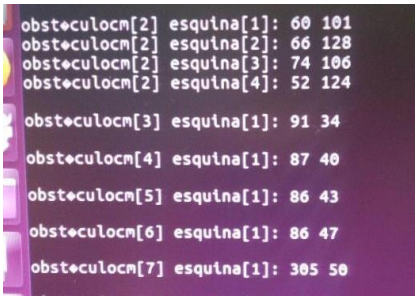

Figura 10. Identificación de obstáculos falsos compuestos por un único punto.

En primer lugar, el hecho de existir un obstáculo de área 0 es una incoherencia. En segundo, al ser identificado dicho punto como obstáculo se alteran las posibles trayectorias del robot con el objetivo de que éste evite dicho punto. El resultado de eliminar el color amarillo implica que el sistema de visión no identifique este color como obstáculo, sino el verde en su lugar, altamente distinto al color de la base. En consecuencia, se elimina la determinación de esos obstáculos falsos.

- Se ha elevado la cámara hasta un total de aproximadamente 3.5 metros, de modo que pueda captar el espacio de trabajo completo.
- Por último, y en concordancia a lo explicado en el punto d) en el apartado 2.1. , se han cambiado, en el programa relativo al sistema de visión, los datos geométricos correspondientes a la dimensión total del espacio y la posición de las regiones rectangulares (los marcadores no han variado su tamaño original), de manera que coincidan con los de la plataforma actual.
- En ocasiones, el sistema de visión detecta un número diferente de esquinas al real. Esto supone un problema porque parte del obstáculo podría quedar fuera si se dan ángulos cóncavos. Ese espacio se estaría considerando libre para el robot cuando en realidad es restringido. Para solucionarlo, se ha diseñado e implementado durante el desarrollo de este TFG una función denominada *obstacles_filter* (definida en el archivo *obstacles_filter.cpp*) que obtiene el convexo de cada obstáculo (*convex_hull* [4]), tal y como se aprecia en la Figura 10, y puede eliminar aquellos que tengan un área menor a un valor a definir en el programa.

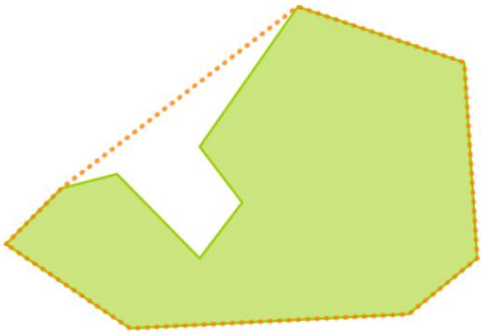

Figura 11. Representación de la obtención de un nuevo polígono formado únicamente por ángulos convexos. [4]

3. PLANIFICACIÓN DE TRAYECTORIAS

Dado un mapa, uno de los principales objetivos de un robot móvilR_i, y del cual trata este capítulo, reside en la capacidad de alcanzar un punto destino (q_{goal}^{Ri}) a partir de un punto inicial (q_{init}^{Ri}) sin colisión alguna. Por lo tanto, es necesario que el sistema planifique trayectorias libres de colisiones con los obstáculos, los cuales son considerados fijos en este trabajo.

De todas las trayectorias posibles, se identificará aquella que suponga un coste menor en cuanto a distancia (en este caso, siendo un sistema discreto, el número mínimo de estados intermediarios).

3.1 ALGORITMOS DE PLANIFICACIÓN

Los algoritmos de planificación se dividen en dos clasificaciones: los métodos exactos y los basados en muestreo. Los primeros construyen representaciones discretas de un entorno dado sin pérdida de información. Su principal ventaja es que son completos, es decir, siempre encuentran una solución si ésta existe en un tiempo finito [5]. Respecto a los segundos, muestrean aleatoriamente el mapa y realizan búsquedas discretas, por lo que se consideran algoritmos probabilísticamente completos, es decir, la probabilidad de encontrar una solución (si existe) converge a 1 cuando el tiempo de ejecución tiende a infinito [5]. Sacrifican la completitud por eficiencia, ya que en algunos casos encuentran la solución (si existe) más rápido que un algoritmo completo [6].

MÉTODOS EXACTOS

Roadmaps: El entorno real se representa a partir de nodos, los cuales corresponden a localizaciones específicas, y curvas unidimensionales, las cuales representan conexiones entre dos localizaciones vecinas. Se crea de este modo una red que habilita la construcción de un camino entre dos nodos cualesquiera. A continuación, se incluyen los puntos inicial y destino en la red, para después obtener una serie de trayectorias libres de colisiones (con obstáculos) que conecten ambos puntos. A esta categoría pertenecen dos de los métodos más usados: Grafo de visibilidad y basado en el Diagrama de Voronoi.

Grafo de visibilidad. La principal diferencia de este algoritmo respecto de los demás, es que sus nodos comparten un camino únicamente si son visibles recíprocamente, es decir, un nodo P se enlaza con un nodo P' si P' es visible desde P.

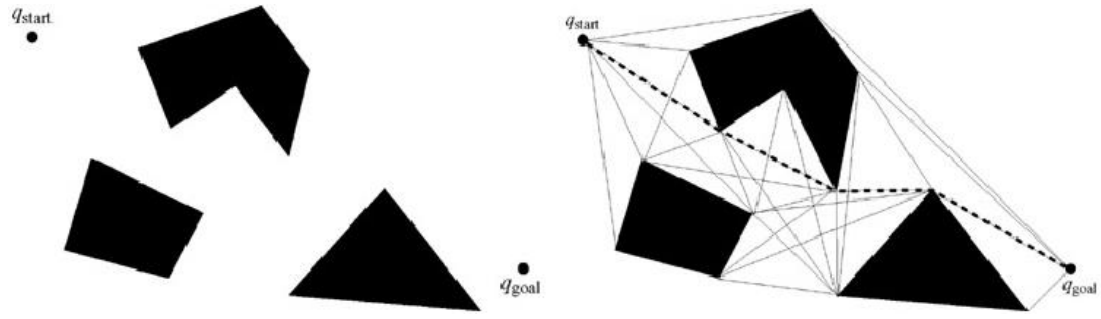

Figura 12. Grafo de visibilidad con obstáculos.

De este modo se obtienen caminos de longitud mínima, debido a que se acercan lo máximo posible a los obstáculos. En el caso que se desarrolla en este trabajo, esto supone un

inconveniente cuya solución implicaría ampliar el área de los obstáculos a modo de coeficiente de seguridad.

Diagrama de Voronoi. Se define como el lugar geométrico de los puntos donde las distancias a los dos obstáculos más cercanos del entorno son iguales. Esto implica que las trayectorias obtenidas se mantendrán lo más alejadas posibles de los obstáculos del espacio de trabajo, por lo que no es necesario aumentar el área de los obstáculos, tal y como sucede para el caso del grafo de visibilidad.

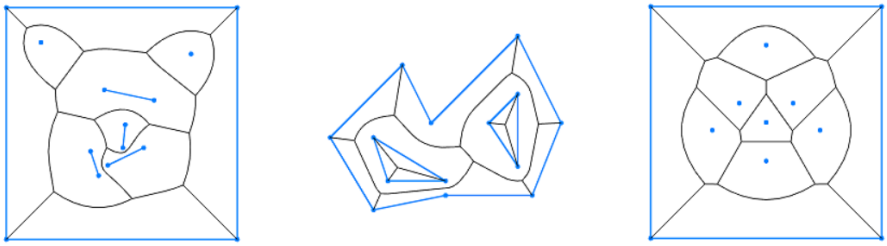

Figura 13. Diagramas de Voronoi, con un conjunto de puntos y segmentos a evitar(1), segmentos a evitar(2), y con puntos a evitar(3). [9]

Particiones en celdas. Estos algoritmos consisten en la partición del entorno en celdas de una misma forma geométrica (triángulos, rectángulos, etc.) y crear una abstracción a un modelo discreto, grafo o red de Petri, en general. A cada celda obtenida por partición se le asocia un nodo en el modelo discreto y las transiciones se construyen utilizando la relación de adyacencia de las celdas. Si se abstrae a un grafo (un autómata finito determinista) la trayectoria se obtiene utilizando algoritmos de búsquedas en grafos [6]. Por contrario, si se utilizan las redes de Petri, los algoritmos de planificación utilizan problemas de programación matemática o de optimización [7].

Es preciso mencionar que los métodos exactos definidos hasta ahora, necesitan complementarse con algoritmos de búsqueda de caminos para generar la trayectoria óptima que conecta series de puntos desde el punto inicial hasta el punto objetivo.

MÉTODOS BASADOS EN MUESTREO

PRM (Probabilistic Roadmap): Se construye un grafo que representa la configuración espacial mediante la generación de configuraciones aleatorias, creando conexiones libres de colisión entre pares de configuraciones cercanas mediante un planificador local.

El mapa de caminos se representa por el grafo $G = (V, E)$. Los nodos en Vson un conjunto de configuraciones del robot elegidas por algún método sobre el espacio libre. Ecorresponde a los caminos libres de colisión conectando dos configuraciones $q1$y$q2$. Dadas una configuración inicial $qinit$y una final $qgoal$,se conectan cada una a dos configuraciones q' y q'',respectivamente. A continuación, el planificador busca en Guna secuencia de caminos en Eque una q' y q''.

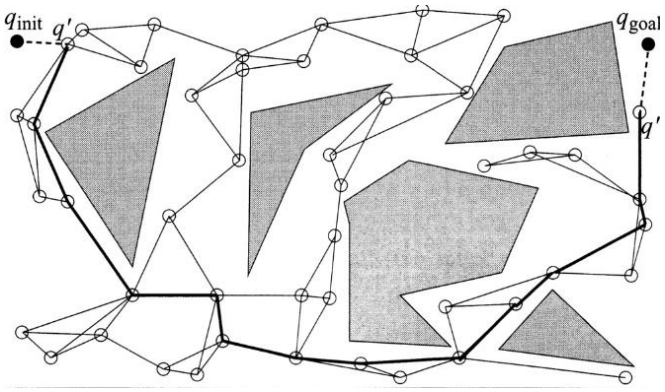

Figura 14. Ejemplo de un PRM para un robot en un espacio de trabajo 2D. El camino más corto aparece denotado por las líneas gruesas. Fuente: [5]

RRT (Rapidly-exploring random tree): La principal ventaja de este método respecto al PRM es que no requiere realizar conexiones entre pares de configuraciones por lo que puede resultar más eficiente para problemas holonómicos [8].

La idea básica de este método consiste en construir un árbol mediante la generación de nuevos estados aleatorios, que pueden verse como nuevas extensiones del árbol original, tal y como muestra la Figura 15.

Sin embargo, este método no obtiene el camino óptimo. Esta restricción se soluciona con la variante RRT*, que garantiza la obtención de la trayectoria óptima cuando se trata de sistemas complejos no-holonómicos.

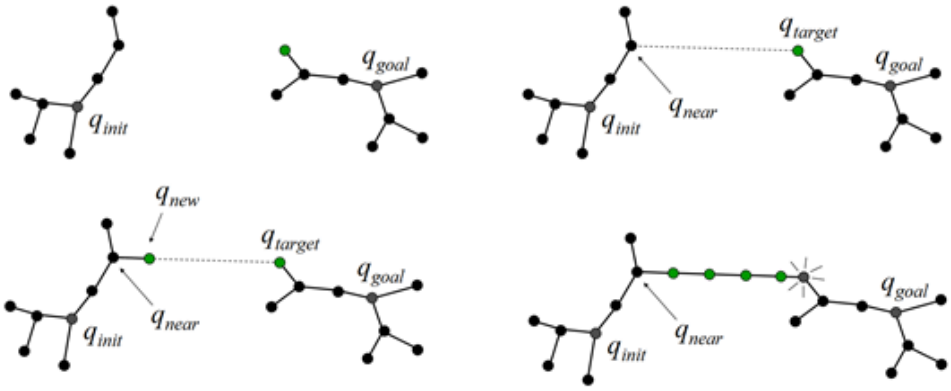

Figura 15. Ejemplo del proceso iterativo de construcción del árbol en el método RRT.

3.2 IMPLEMENTACIÓN DE UN ALGORITMO PARA PLANIFICACIÓN DE TRAYECTORIAS

Para el desarrollo de este TFG se ha decidido hacer uso del algoritmo basado en el diagrama de Voronoi. La principal ventaja de este algoritmo es que no requiere la ampliación del área de los obstáculos, como sucede en el caso del grafo de visibilidad. Esto permite aprovechar al máximo el espacio de trabajo y es más difícil que llegue a producirse una colisión ficticia con un obstáculo (ficticia, porque en realidad pasaría por encima). Dado que éste es limitado, agrandar todavía más los obstáculos supondría reducir excesivamente el espacio libre por el que pueden circular los robots.

DIAGRAMA DE VORONOI

La implementación de este algoritmo en C++ se encuentra definida en el archivo *VoronoiDiagram.cpp*. Las variables procedentes del módulo de visión con las que se va a trabajar en este apartado son:

- la posición de las esquinas de cada obstáculo, y
- las dimensiones del espacio de trabajo (w, h).
- El punto inicial en el que se encuentra el robot.
- El punto objetivo de la trayectoria del robot también es necesario, aunque éste es dado por el usuario.

El cálculo del diagrama de Voronoi se realiza haciendo uso de la librería Boost [9], en concreto las funciones implementadas en el fichero *voronoi.hpp*. Su uso e implementación está explicado en su página web [10]. No obstante, se han diseñado e implementado nuevas funciones adicionales que se detallan más adelante.

En el archivo *voronoi.hpp* se definen dos funciones estáticas que integran la construcción del diagrama de Voronoi:

- *template <typename PointIterator, typename VD>*
  void ***construct_voronoi****(PointIterator first,*
  PointIterator last,
  *VD *vd)*

  Construye el diagrama de Voronoi a partir de un conjunto de puntos.

- *template <typename SegmentIterator, typename VD>*
  void ***construct_voronoi****(SegmentIterator first,*
  SegmentIterator last,
  *VD *vd)*

  Construye el diagrama de Voronoi a partir de un conjunto de segmentos.

La segunda de ellas es la que mejor se adapta a nuestro caso, ya que únicamente es necesario separar los obstáculos en segmentos, mientras que la primera supondría discretizar los segmentos que forman los obstáculos, lo cual resulta más costoso. Para limitar el diagrama de Voronoi al espacio de trabajo, se añadirán los segmentos que definen el rectángulo del espacio de trabajo en sí.

De este modo, sendas variables de entrada (obstáculos y límites del espacio de trabajo) han de convertirse a un vector de segmentos. En el mismo archivo en el que se trabaja en este apartado, se ha diseñado e implementado la función ***create_segments****,* que cumple con este requisito. Con el siguiente fragmento de código se construye el diagrama de Voronoi correspondiente al conjunto de segmentos introducidos:

```
std::vector<Segment> segments = create_segments(w, h, obstacles);
              //w=largo de la plataforma, h=alto de la plataforma, obstacles=salida
              módulo //de visión.
// Construction of the Voronoi Diagram.
voronoi_diagram<double> vd;
construct_voronoi(segments.begin(), segments.end(),&vd);
```

La función utilizada para la construcción del diagrama no tiene salida como tal ya que el último parámetro de entrada es un *pointer* (una dirección de memoria) a la variable *vd* en la cual se guarda el diagrama de Voronoi. De esta forma, los valores de este vector se pueden cambiar en la función y se mantienen guardados también después de la finalización de la función. El uso de *pointers* permite reducir el uso de memoria evitando definir otras variables y además reducir los parámetros de salida de una función. Para visualizar y trabajar con el diagrama de Voronoi obtenido, es preciso recorrer el diagrama y extraer los segmentos que lo forman de la siguiente forma:

```
for (voronoi_diagram<double>::const_edge_iterator it = vd.edges().begin();
it != vd.edges().end(); ++it) {
   if (it->is_primary()) {
        const voronoi_diagram<double>::vertex_type* vertex_init = it->vertex0();
        const voronoi_diagram<double>::vertex_type* vertex_end = it->vertex1();

        ……//Resto de código de lectura, en el que se incluyen las discriminaciones
que se explican a continuación.

}}
```

Las variables y funciones que aparecen en esta parte de código son:

- ***vd.edges()*** representa el conjunto de segmentos que forman el diagrama.
- ***is_primary()*** es una función miembro que devuelve falso si el segmento atraviesa el extremo de un segmento de entrada.
- ***vertex_type*** es una variable miembro que define los vértices de un segmento
- ***vertex0()*** devuelve un puntero al vértice inicial del segmento.
- ***vertex1()*** devuelve un puntero al vértice final del segmento.
- ***const_edge_iterator*** itera sobre el conjunto de segmentos del diagrama de Voronoi ***vd***.

En el proceso de extracción de los segmentos del diagrama de Voronoi anterior, es necesario realizar ciertas discriminaciones de puntos del diagrama no requeridos, debido a los siguientes factores:

1. Cada segmento está definido dos veces: dados dos vértices A y B pertenecientes al diagrama, si existe una línea que los conecta, entonces es posible recorrer dicho segmento desde A hasta B y viceversa. Es decir, como los segmentos son unidireccionales, existen dos de mismo módulo y dirección, y distinto sentido. Con el objetivo de evitar dobles definiciones de los nodos que forman el diagrama, es necesario eliminar uno de ellos, acción que se lleva a cabo gracias al diseño e implementación de la función ***check_double_solution***.
2. Se construyen segmentos pertenecientes al diagrama cuyo origen o fin coinciden con uno de los extremos de los segmentos de entrada que representan los límites del espacio de trabajo. Es preciso no tenerlos en cuenta puesto que conducen al límite del entorno y no suponen la evitación de un obstáculo. La función implementada, también en este TFG, ***intersect_with_inputgeometry*** cumple con este requerimiento.
3. El diagrama de Voronoi también se crea en el interior de los obstáculos, ya que se crea a partir de los segmentos que forman el perímetro de los obstáculos. Es preciso eliminarlos porque resulta incoherente que el robot pueda pasar por encima de los obstáculos, lo que se ha considerado como colisión ficticia anteriormente. Esta discriminación se realiza a través de la función implementada ***inside_obstacle***.

Tras proceder a las discriminaciones, se devuelven al programa principal los segmentos extraídos, a partir del cual se calculan las posibles trayectorias. Para ello, antes es necesario incluir los puntos inicial y objetivo del robot. Se añaden por lo tanto dos nuevos segmentos al conjunto antes mencionado:

1. Un segmento cuyo origen es el punto inicial de la trayectoria del robot y cuyo final es el nodo (vértice) más cercano perteneciente al diagrama de Voronoi.
2. Un segmento cuyo origen es el punto objetivo del robot y cuyo final es el nodo (vértice) más cercano perteneciente al diagrama de Voronoi.

Ambas acciones se realizan gracias a la función implementada en este TFG ***nearest_vertex***.

EJEMPLO OBTENCIÓN DIAGRAMA VORONOI

El siguiente ejemplo muestra el procedimiento seguido para discretizar el entorno según el diagrama de Voronoi:

1. Se consideran las siguientes variables, procedentes del sistema de visión:
    - Largo total (width) = 30
    - Alto total (height) = 20
    - Coordenadas de las esquinas del obstáculo: [(13,9) ; (16,14) ; (19,10) ; (15,8)]
2. Se convierte la información anterior a un conjunto de segmentos.
3. El diagrama de Voronoi resultante, producto del constructor especificado anteriormente, consiste en lo mostrado por el siguiente gráfico. Este es el diagrama completo, pero tal y como se ha comentado, existen algunos puntos que no se requieren en este trabajo, bien porque no son necesarios, o bien porque derivarían en una colisión del robot.

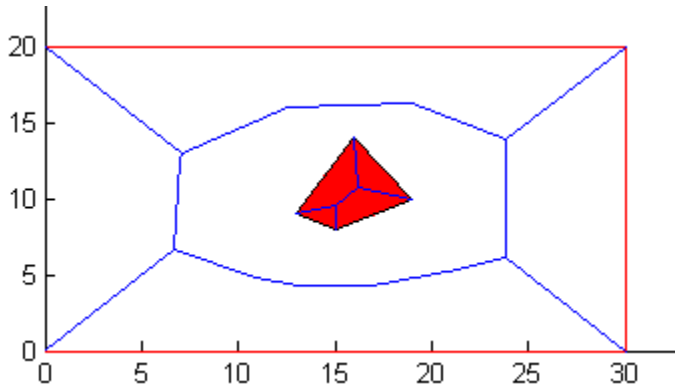

Figura 16. Diagrama de Voronoi del espacio de trabajo definido por las líneas de contorno en color rojo y que contiene un obstáculo, también de color rojo. En azul, los segmentos del diagrama resultante. Fuente: Matlab.

4. Por consiguiente, se procede a las discriminaciones requeridas:
    - ***check_double_solution****:* los segmentos pasan a estar definidos una única vez.
    - ***intersect_with_inputgeometry****:* desaparecen los segmentos que intersectan o bien con los límites del entorno, o bien con los obstáculos.

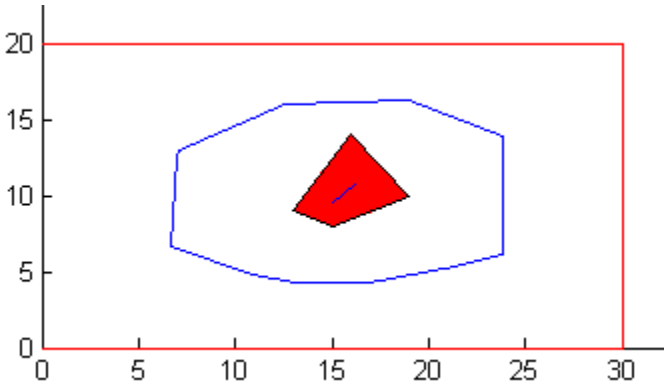

Figura 17. Diagrama obtenido tras proceder a la discriminación número 2. Se aprecia la desaparición de 8 segmentos en total con respecto a la Figura 16. Fuente: Matlab.

- ***inside_obstacle**:*

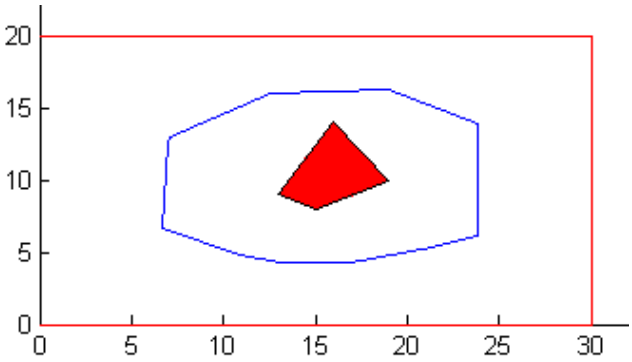

Figura 18. Diagrama obtenido tras las tres discriminaciones. En este caso, se discrimina un único obstáculo. Fuente: Matlab.

- Esta última eliminación no es indispensable, puesto que al crear el posterior grafo dicho segmento aparecería aislado, por lo que el robot no llegaría a circular por él. Sin embargo, se lleva a cabo para completar la operación.

5. Por último, se añaden dos segmentos que incorporen las configuraciones inicial q_{init}^{Ri} y final q_{goal}^{Ri}.

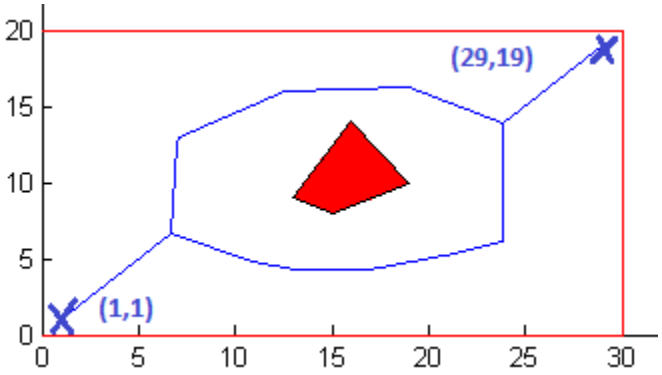

Figura 19. Adición de los puntos inicial (29,19) y objetivo (1,1) de un robot R1, mediante la creación de segmentos que los unen a los vértices más cercanos, respectivamente. Fuente: Matlab.

3.3 GRAFO REPRESENTATIVO DE LA RED DE TRAYECTORIAS

Con el fin de calcular de una forma eficiente todas las posibles rutas desde una configuración inicial hasta una final, se opta por crear un grafo en el que los nodos son los vértices de los segmentos del diagrama resultante de las operaciones realizadas en el apartado anterior, además de los puntos inicial y objetivo del robot. En consecuencia, dos nodos están unidos si existe un segmento que los une en el diagrama.

La clase Graph, diseñada e implementada en este TFG y definida en los archivos *Graph.h* y *Graph.cpp*, obtiene de forma automática el grafo representativo de los segmentos extraídos en la planificación de trayectorias. Se ha hecho uso también en este apartado de una clase grafo de la librería Boost [11] (ver inclusión de librerías externas en Visual Studio en el Anexo 3).

En primer lugar, se ha de generar un grafo de n nodos, con n el número total de vértices pertenecientes al diagrama de Voronoi. Después se han de añadir progresivamente las líneas de unión entre ellos, las cuales corresponden a los segmentos obtenidos en el apartado anterior. Paralelamente se añade un valor, denominado peso o coste, relativo a cada línea de unión. En este caso se considera la distancia de dicho segmento, la cual se calcula como la distancia euclídea entre sus dos vértices o nodos.

```
typedef adjacency_list < listS, vecS, directedS,
             no_property, property < edge_weight_t, double > > graph_t;
int count = vertices_.size();
graph_t g(count); //esta función hace que a cada vértice le corresponda un
entero, que actuará a partir de ahora como su marcador.
```

Este fragmento de código genera un grafo con número de nodos igual al tamaño de un vector que contiene todos los vértices del diagrama de Voronoi. La siguiente función *add_edge*, perteneciente a la librería Boost, crea las conexiones entre nodos con sus respectivos pesos:

```
add_edge(a, b, weight, g);
```

Donde:

- *a* representa el nodo inicial.
- *b* representa el nodo final
- *weight* es el peso que se le asigna a la unión entre *a* y *b*, el cual se calcula gracias a la función implementada en este trabajo ***set_weights***.
- *g* es el grafo al que pertenece la unión a generar.

3.4 ALGORITMO DE BÚSQUEDA DE CAMINOS

Una vez definido el grafo representativo de la red de trayectorias, es preciso implementar un método mediante el que se obtenga la ruta óptima que conecte una serie de puntos entre el punto inicial y el punto objetivo, la cual se enviará al robot.

La trayectoria que se busca es la más corta, es decir, aquella que suponga el mínimo coste posible en términos de distancia física. Por consiguiente, el algoritmo a utilizar debe cumplir dos requisitos:

- Detectar todos los caminos posibles entre dos nodos del grafo.
- Seleccionar el óptimo en términos de mínima distancia a recorrer.

El algoritmo de Dijkstra representa una gran opción para el caso de redes estáticas en las que el peso de los arcos es conocido y no-negativo.

DESCRIPCIÓN

Para un nodo perteneciente al grafo definido, este algoritmo busca y encuentra paso a paso las rutas más cortas entre dicho nodo y el resto de ellos.

1. Se comienza seleccionando el nodo inicial, a partir del cual se calcularán las distancias del resto de los nodos.

2. Se marca el nodo inicial como actual, y el resto se incluyen en un conjunto compuesto por los nodos no visitados.

3. Se asignan ciertos valores a cada uno de los nodos: 0 al nodo inicial, infinito al resto, ya que todavía no han sido visitados.

4. Para el nodo actual, se calculan las distancias a todos sus nodos vecinos (pertenecientes al conjunto de los no visitados), y se asigna a cada uno de ellos la suma del valor del nodo actual (si es el inicial, esto es 0) más la longitud del camino que los une al nodo inicial, solamente si la cantidad es menor que la previamente asignada.
5. De todos los nodos con conexión directa al actual se selecciona aquel situado a una distancia menor, se elimina del conjunto de los nodos no visitados, y se marca como actual. En el caso de que existan dos o más distancias de igual valor, y este sea el mínimo, no importa cual se marca primero como actual.
6. Si el conjunto de nodos no visitados no está vacío, se retrocede al punto 4. De lo contrario, el algoritmo ha terminado.

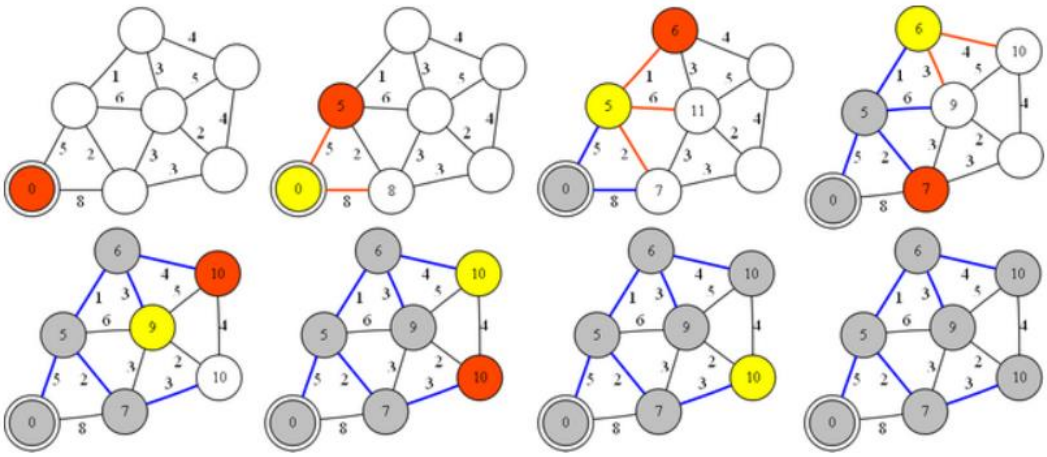

Figura 20. Ejemplo del proceso iterativo del algoritmo Dijkstra para obtener los caminos óptimos (de menor peso).[2]

[2] *https://steemit.com/popularscience/@krishtopa/dijkstra-s-algorithm-of-finding-optimal-paths*

IMPLEMENTACIÓN

El algoritmo de Dijkstra se ha implementado en este TFG como una función que devuelve una trayectoria de puntos, estableciendo como entrada un grafo y los puntos inicial y final del robot, haciendo uso del fichero *dijkstra_shortest_paths* perteneciente a la librería Boost [12]. Dicha función puede encontrarse en el archivo *Dijkstra.cpp*. La función miembro que obtiene la ruta más corta desde el nodo inicial hasta cada uno de los nodos restantes es la siguiente:

```
typedef adjacency_list < listS, vecS, directedS,
        no_property, property < edge_weight_t, double > > graph_t;

typedef graph_traits < graph_t >::vertex_descriptor vertex_descriptor;

dijkstra_shortest_paths(g, start,
                        predecessor_map(boost::make_iterator_property_map(p.begin(),
                        get(boost::vertex_index, g))).
                        distance_map(boost::make_iterator_property_map(d.begin(),
                        get(boost::vertex_index, g))));
```

Donde "*g*" es la variable representativa del grafo (tipo *graph_t*), y "*start*" la del punto inicial (tipo *vertex_descriptor*). Para obtener la trayectoria entre el punto inicial y el punto final, se implementa el siguiente fragmento de código:

```
std::vector< vertex_descriptor > path;
vertex_descriptor current = target;
while (current != start) {
        path.push_back(current);
        current = p[current];
}
path.push_back(start);
```

Donde p[] es el "*mapa predecesor*" obtenido a través de la función miembro *dijkstra_shortest_paths*, el cual guarda las conexiones en la red de mínima expansión (esto es, la red de caminos más cortos). Es decir, se recorre la red en sentido inverso: desde el punto final al punto inicial, pasando por los nodos intermedios que conforman la trayectoria óptima.

4. EVITACIÓN DE COLISIONES Y PREVENCIÓN DE BLOQUEOS

A lo largo del apartado 3 de este trabajo, se han explicado los fundamentos prácticos y teóricos para el cálculo de la trayectoria de un robot en la plataforma. Sin embargo, el objetivo último es crear un sistema multi-robot en el que un conjunto de robots pueda cumplir con sus tareas bajo condiciones de total seguridad.

Los algoritmos diseñados relativos a la planificación de trayectorias garantizan el cumplimiento de las condiciones de seguridad (1) y (2), establecidas en la introducción de este TFG. Respecto a la condición número (3), dado que las trayectorias de cada robot se calculan de forma individual ignorando al resto de ellos, se requiere un mecanismo adicional que garantice la evitación de colisiones.

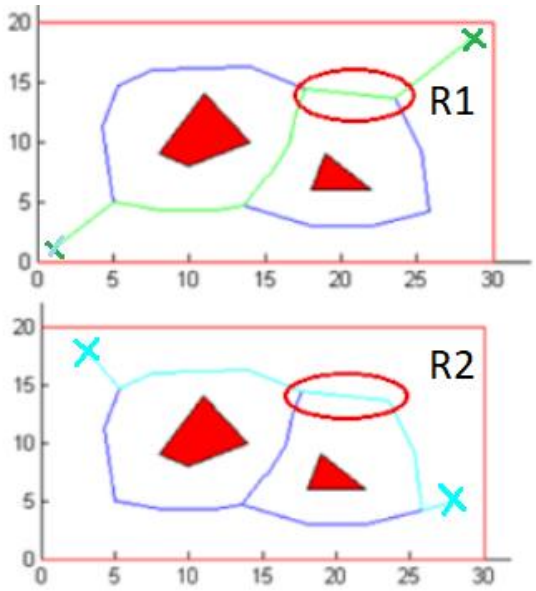

Figura 21. Situación de posible colisión en la que un robot R1 (trayectoria en color verde) y un robot R2 (trayectoria en color cian) presentes en el mismo espacio de trabajo, comparten trazado (redondeado en rojo) en su ruta hacia sus respectivos puntos objetivos. En rojo, los obstáculos a evitar. En azul los segmentos obtenidos a partir del diagrama de Voronoi. Fuente: Matlab.

4.1 CONTROLADOR CENTRALIZADO BASADO EN REDES DE PETRI

A) EVITACIÓN DE COLISIONES

Es muy importante asegurar la ausencia de posibles colisiones entre robots durante la ejecución de las trayectorias, por lo que esta parte del trabajo está dedicada a la implementación de un mecanismo que lo consiga.

Para explicar su importancia, veamos el siguiente ejemplo. Supongamos la existencia de dos robots, *R1* y *R2* en el espacio de trabajo. Dicho espacio está modelado por un grafo *G* y el conjunto de trayectorias de los robots son *r1* y *r2*, respectivamente. Las componentes de las trayectorias se denominan a partir de ahora regiones y serán consideradas de capacidad unitaria (es decir, solo un robot puede atravesarlas al mismo tiempo). Su denotación consistirá en la letra "c" (del inglés *cell*), seguida de un número que las identifica frente al resto. A modo de ejemplo, se supone que las trayectorias son las siguientes:

- r1 = {c1 c3 c4 c5 c6 c7}
- r2 = {c2 c8 c4 c3 c5 c9}

Las regiones objetivo para R1 y R2 son respectivamente c7 y c9, mientras que las iniciales son c1 y c2. Tal y como se puede observar, ambas trayectorias coinciden en tres regiones (c3, c4, c5), por lo que los robots podrían colisionar. Sin embargo, los movimientos de un robot en particular pueden entenderse como un sistema basado en secuencias. Dicho de otro modo, un conjunto de *n* robots puede verse como *n* sistemas secuenciales ejecutándose paralelamente formando un sistema concurrente. Este sistema de eventos discretos puede representarse con una red de Petri S4PR [13] (ver definición en el Anexo 1).

Las regiones que recorrer están representadas dentro de los procesos por lugares, que son variables de estado y se marcarán solamente cuando el robot correspondiente al proceso se encuentre en ese estado. La capacidad de una región se modela usando un lugar recurso, de

modo que el recurso se cede cuando el robot entra en la región (ejecutando la transición de entrada) y es liberado al abandonarla (ejecutando la transición de salida). Las regiones c1 y c2 son los lugares de reposo (idle) de los procesos o de inicio de los robots, por eso contienen una marca cada uno al inicio.

Para evitar las posibles colisiones, se introducen modos de espera, es decir, se impone que por las regiones comunes no pueda pasar más de un robot en el mismo instante de tiempo, de modo que el segundo en llegar a esa región deberá esperar a que el primero la abandone.

El requisito de *modo de espera* en una red de Petri se soluciona introduciendo los lugares de capacidad mencionados antes. Si las trayectorias pasan por una misma región, el lugar que modela la capacidad de la región será un recurso compartido que los dos procesos (las dos trayectorias) tienen que asignar y liberar para poder continuar. Estos lugares de capacidad contienen inicialmente tantas marcas como capacidad tenga la región, en este caso se asignará un único recurso a los lugares correspondientes a regiones comunes con capacidad unitaria. Esta marca se consume cuando se dispara 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. Los lugares de recurso que modelan capacidades de regiones que no son comunes son recursos privados y si su marcado inicial es mayor que 1 son lugares implícitos[3]. Estos lugares se añaden solo para asegurar que la clase de red de Petri que se obtiene a modelar el conjunto de trayectorias pertenece a la clase S4PR y utilizar los resultados de esta clase para evitar los bloqueos.

En resumen, al generar modos de espera, se garantiza que nunca coincidan los dos robots en las regiones con capacidad restringida.

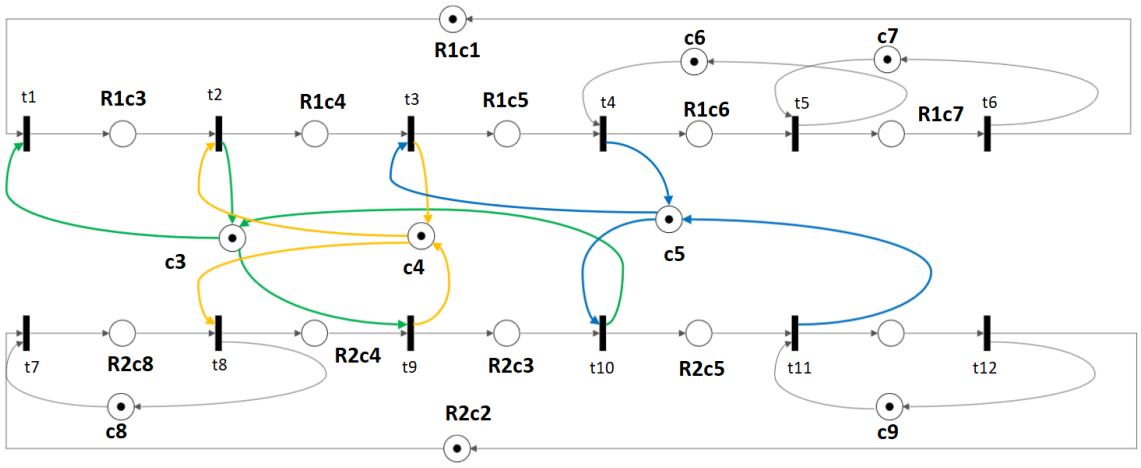

Figura 22. Red de Petri abstraída de las trayectorias r1 y r2 de los robots R1 y R2. Los recursos compartidos se representan en color, mientras que los privados en gris. Marcado inicial en los lugares idle y en los recursos. Fuente: smartdraw[4]

[3] En redes de Petri, un lugar implícito es un lugar que no cambia la dinámica del sistema a ser eliminado.
[4] smartdraw-Diagram Software: https://www.smartdraw.com/

IMPLEMENTACIÓN

La estrategia anterior debe trasladarse ahora a la realidad. La salida obtenida en la parte de planificación de trayectorias, esto es, los caminos de los dos robots desde sus puntos iniciales hasta sus puntos objetivo, debe modelarse como una red de Petri para hallar los recursos compartidos y así poder implementar un controlador centralizado que garantice la evitación de colisiones en la plataforma multi-robot.

Se ha implementado en este TFG una clase en C++, denominada ***Input_S4PR*** y definida en el archivo *Petri_Net.h,* que obtiene una red de Petri S4PR que modela el conjunto de trayectorias de los robots, estableciendo los recursos compartidos (correspondientes a las regiones comunes de las trayectorias) y su marcado inicial, dependiendo este último de las restricciones de capacidad de cada lugar de la red, que se suponen 1.

Su utilización es la siguiente. Existen dos constructores diferentes, cuyo uso depende del formato usado durante la planificación de trayectorias.

- El primer constructor tiene como parámetros de entrada el conjunto de trayectorias de los robots (un vector de dos dimensiones), cuando estas están definidas por regiones, y un vector de tipo *int* que representa las capacidades de cada región. Es decir, si en todas las regiones la capacidad máxima es de un robot, dicho vector solo contiene unos. Si, por ejemplo, la región c3 puede albergar dos robots al mismo tiempo, en la posición dos del vector habrá un 2.

```
Input_S4PR(std::vector<std::vector<int>> paths, std::vector<std::vector<int>>
capacities);
```

- El otro tiene como parámetros de entrada el conjunto de trayectorias de los robots, cuando estas están definidas por puntos, por lo que el vector es ahora de tres dimensiones ya que cada trayectoria se representa como se explica en el apartado 3.4. Este constructor está diseñado para la planificación basada en el diagrama de Voronoi. No se requiere un vector capacidad porque al tratarse de puntos del plano 2D, se sobreentiende que dos robots no pueden situarse en el mismo punto en un mismo instante de tiempo t, luego se restringen automáticamente todos los lugares a capacidad 1.

```
Input_S4PR(std::vector<std::vector<std::vector<double>>> paths);
```

Siguiendo con el ejemplo anterior, los parámetros obtenidos de la red de Petri que modela el sistema, suponiendo que la capacidad de las regiones es de un solo robot al mismo tiempo, son los presentados en la siguiente imagen. Descripción detallada de la obtención de estos parámetros en el Anexo 2.

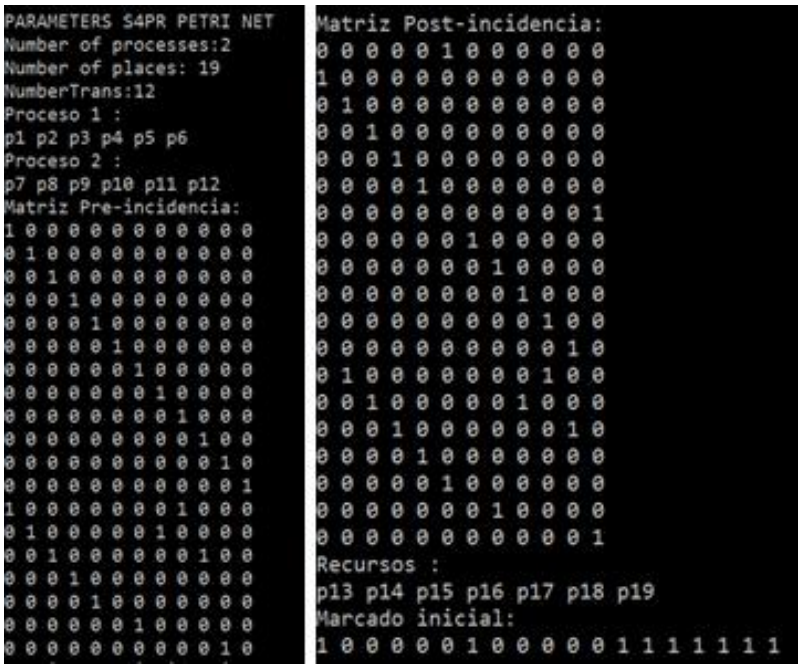

Figura 23. Parámetros de la red de Petri que modela las trayectorias de los robots, obtenidos a partir de la clase Input_S4PR.

Tal y como se puede apreciar en la imagen, la red de Petri obtenida tiene un total de 19 lugares y 12 transiciones. El proceso 1, compuesto por los lugares p1, p2, p3, p4, p5, p6, corresponde a la trayectoria obtenida para el primer robot: {c1 c3 c4 c5 c6 c7}. El proceso número 2, compuesto por los lugares p7, p8, p9, p10, p11, p12, corresponde a la trayectoria obtenida para el segundo robot: {c2 c8 c4 c3 c5 c9}. Los lugares de p13 a p19 modelan los lugares de capacidad, de cuales los p13, p14 y p15 corresponden a recursos compartidos, mientras que el resto son recursos privados. Las matrices de incidencia Pre y Post se representan en la Figura 23.

A pesar de evitar colisiones entre los robots, la inclusión de los modos de espera puede derivar en un bloqueo de la red (*deadlock*), ya que se puede llegar a una espera circular entre un conjunto de robots que implícitamente lleva a bloqueo. En el ejemplo actual, un bloqueo se produce cuando el primer robot se encuentra en R1c3 (p2) y tiene que seguir hacia R1c4 (p4), pero tiene que esperar al segundo robot que se encuentra en R2c4 (p9) y debe avanzar a R2c3 (p10).

En redes de Petri, el bloqueo está relacionado con algunos elementos estructurales llamados sifones y en la clase S4PR los sifones mínimos malos (definidos en la siguiente sección) pueden llevar el sistema a bloqueo si se quedan sin marcas [14]. Se entiende como bloqueo total o *deadlock* toda aquella situación que surge durante la evolución del sistema en la red de Petri y que se 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.

B) PREVENCIÓN DE BLOQUEOS

Por consiguiente, se plantea un segundo objetivo del mecanismo a implementar: evitar esta situación mediante la generación de lugares de control. Estos lugares, también denominados lugares monitor en términos de redes de Petri, limitan el número de robots en un

conjunto dado de regiones impidiendo el vaciado completo de los *sifones malos.* Esta solución es válida porque para una red S4PR, como es nuestro caso, toda situación de bloqueo está relacionada con al menos un sifón mínimo vacío [15].

En redes de Petri, un *sifón* es un subconjunto de lugares tal que su conjunto de transiciones de entrada está contenido en el conjunto de transiciones de salida: $S \subseteq P$ es un sifón si $\bullet S \subseteq S\bullet$.

En el ejemplo correspondiente a la figura 21, el conjunto de lugares $S = \{R_1c_3, R_2c_3, c_3\}$ es un sifón ya que el conjunto de transiciones de entrada

$$\bullet S = \bullet R_1c_3 \cup \bullet R_2c_3 \cup \bullet c_3 = \{t_1\} \cup \{t_9\} \cup \{t_2, t_{10}\} = \{t_1, t_2, t_9, t_{10}\}$$

Está incluido en el conjunto de transiciones de salida:

$$S \bullet = R_1c_3 \bullet \cup R_2c_3 \bullet \cup c_3 \bullet = \{t_2\} \cup \{t_{10}\} \cup \{t_1, t_9\} = \{t_1, t_2, t_9, t_{10}\}$$

Si los lugares pertenecientes a un sifón se vacían en un momento dado, es decir, se quedan sin marcas durante la evolución del sistema, no es posible marcarlos de nuevo, ya que el conjunto de transiciones de entrada está incluido en el conjunto de transiciones de salida.

Por otro lado, una *trampa* es un subconjunto de lugares tal que su conjunto de transiciones de salida está contenido en el conjunto de transiciones de entrada: $D \subseteq P$ es una trampa si $D\bullet \subseteq \bullet D$.

En el ejemplo correspondiente a la Figura 21, el conjunto de lugares $D = \{R_1c_7, c_7\}$ es una trampa ya que el conjunto de transiciones de salida

$$D \bullet = R_1c_7 \bullet \cup c_7 \bullet = \{t_6\} \cup \{t_5\} = \{t_5, t_6\}$$

Está incluido en el conjunto de transiciones de entrada:

$$\bullet D = \bullet R_1c_7 \cup \bullet c_7 = \{t_5\} \cup \{t_6\} = \{t_5, t_6\}$$

La propiedad de las trampas es que se mantienen marcadas a lo largo de la evolución del sistema si en el marcado inicial existe, al menos, una marca en un lugar de la trampa. En este caso, como c_7 contiene una marca inicialmente, D siempre contendrá una marca con independencia de las transiciones que se disparen.

Sin embargo, no todos los sifones conducen a situaciones de bloqueo. Todos aquellos sifones que contienen trampas no llegaran a vaciarse puesto que una trampa siempre mantiene al menos una marca en su conjunto, son los denominados como sifones *buenos*. Es decir, un sifón $S \subseteq P$ se considera malo si no existe una trampa $D \subseteq P$ tal que $D \subseteq S$. Los sifones malos son los que pueden vaciarse durante la evolución del sistema y, en consecuencia, producir un bloqueo.

Para evitar el vaciado de un *sifón malo* S_k se añade un lugar p_k, denominado lugar monitor, conectado de la siguiente forma:

1) Respecto a la matriz de incidencias:

$$C[p_k, t_j] = \sum_{p_l \in S_k} C[p_l, t_j];$$

2) Respecto al marcado inicial:

$$m_0[p_k] = \left(\sum_{p_l \in S_k} m_0[p_l] \right) - 1;$$

Donde $t_j \in T, p_l \in S_k$, siendo $S_k = \{p_1, \ldots, p_n\}$ uno de los sifones malos.

En el caso que se desarrolla en este TFG únicamente se consideran dos robots, por lo que cualquier lugar monitor p_k tal que $m_0[p_k] >= 2$ puede ignorarse. Por ejemplo, en el caso del sifón $S = \{R_1c_4, R_2c_3, c_3, c_4\}$ del ejemplo que se ha estado desarrollando en este apartado, el lugar monitor PM1 controla el disparo de las transiciones de salida, evitando que S se vacíe:

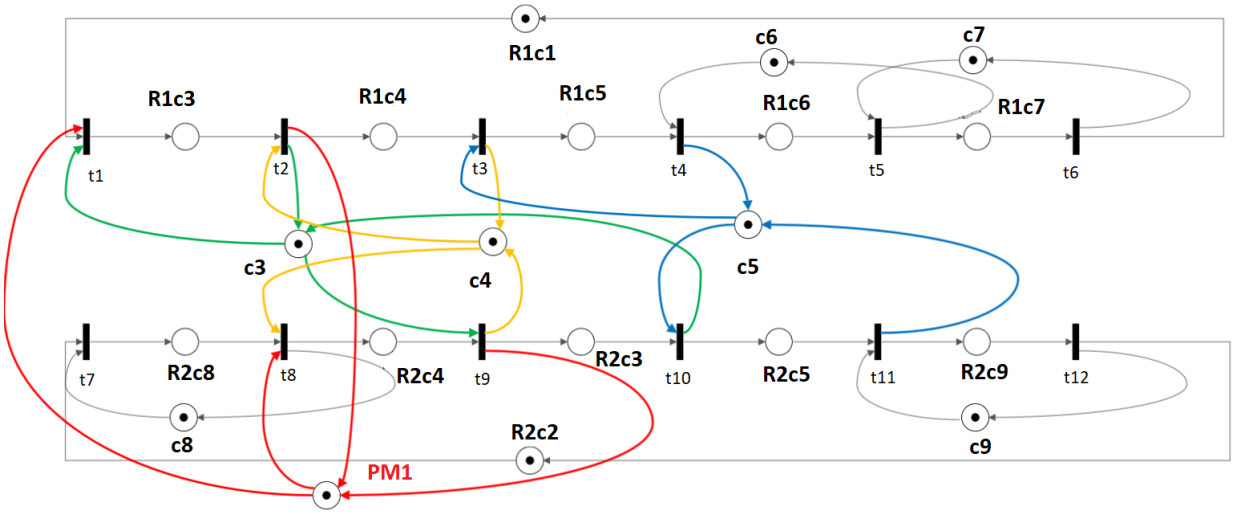

Figura 24. Red de Petri correspondiente al ejemplo detallado en este apartado, con la introducción de un lugar monitor que previene el vaciado de un sifón malo. Fuente: smartdraw[5]

La situación de bloqueo comentada anteriormente[6], ya no llegaría a producirse gracias a la inclusión del lugar monitor PM1, el cual impide que R1 y R2 se encuentren simultáneamente en R1c3 y R2c4, respectivamente.

OBTENCIÓN DE LUGARES MONITOR

Para la obtención de los lugares monitor, se ha usado parte del trabajo realizado por Javier Oroz [16], quien implementó en C++ una clase para las redes de Petri S4PR que permite obtener los sifones malos y los lugares monitor que impiden el bloqueo total a partir de los parámetros que definen la red. Respecto al código original, se ha optado por cambiar la implementación de la función *getControlSiphons(),* responsable de la detección de los sifones *malos*, de forma que la creación de un lugar monitor se condiciona a que su marcado inicial tenga un valor máximo igual al número de robots menos 1. El razonamiento se basa en que un marcado inicial igual o mayor al número de robots, no restringiría el número de marcas en el conjunto de lugares que es necesario controlar para evitar el bloqueo del sistema.

[5] smartdraw-Diagram Software: https://www.smartdraw.com/

[6] *"...un bloqueo se produce cuando el primer robot se encuentra en R1c3 (p2) y tiene que seguir hacia R1c4 (p4), pero tiene que esperar al segundo robot que se encuentra en R2c4 (p9) y debe avanzar a R2c3 (p10)."*

La clase ***Input_S4PR*,** que se ha diseñado e implementado en este TFG, funciona como interfaz entre la salida del algoritmo de planificación y el cálculo de lugares monitor para la prevención de bloqueos, además de actuar como mecanismo para evitar colisiones, como se ha explicado anteriormente.

Siguiendo con el ejemplo, en el apartado anterior se han calculado los lugares de recursos compartidos para evitar colisiones, no obstante, el sistema no está libre de bloqueos (*deadlock*). Se obtiene, por lo tanto, el siguiente conjunto de sifones de la red de Petri, el conjunto de sifones malos, y los lugares monitor necesarios.

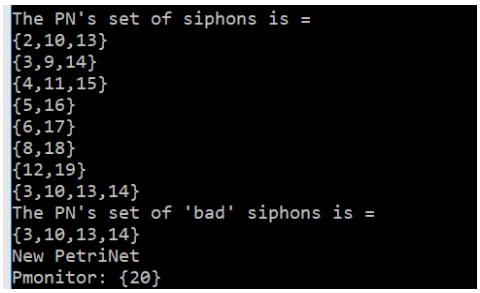

Figura 25. Conjunto de sifones y de sifones malos, obtenidos gracias al trabajo de Javier Oroz.

En consecuencia, los parámetros de la red de Petri quedan alterados de acuerdo a las expresiones 1) y 2) relativas a la adición de lugares monitor descritas en el apartado anterior. Las matrices de incidencia de la Figura 23 son ahora las representadas en la siguiente imagen:

Figura 26. Alteración de los parámetros de la red de Petri debido a la introducción del lugar monitor, correspondiente a la red representada en la Figura 24.

4.2 DISEÑO E IMPLEMENTACIÓN DEL CONTROLADOR CENTRALIZADO

Para diseño del controlador centralizado se tienen en cuenta fundamentalmente dos parámetros:

1. Los lugares de recursos compartidos para evitar que los robots colisionen.
2. Los lugares monitor para prevenir situaciones de *deadlock*.

La idea principal consiste en que, una vez abstraídas las trayectorias de los robots, en una red de Petri S4PR y obtenidos los lugares monitor, el controlador centralizado se ejecute en bucle cerrado hasta que todos los robots hayan alcanzado su configuración final q_{goal}^{Ri} .

Puesto que no existe comunicación entre los robots, el controlador es el responsable de permitir el avance de cada uno de ellos, o bien de establecerlos en modo de espera. Para que este procedimiento funcione correctamente, se propone particionar las trayectorias a enviar a cada robot, según el siguiente planteamiento:

Supongamos dos robots R_1y R_2:

1. En la primera iteración del bucle del controlador, se enviará a cada robot una trayectoria compuesta por:
    a) Su lugar de comienzo (lugar *idle* en la red de Petri)
    b) La secuencia de la trayectoria desde a) hasta el primer lugar restringido, sin incluir este último, para que no se produzcan pérdidas de tiempo. Por el contrario, se establecería en modo de espera en el lugar previo de su trayectoria, esperando a que su próximo destino quede libre.
2. Para el resto de las iteraciones, se enviará a cada robot una trayectoria compuesta por:
    a) El lugar objetivo de la secuencia enviada en la trayectoria anterior, es decir, el lugar donde se encuentra en el momento del envío actual.
    b) Misma secuencia que en 1b).

Por supuesto, el controlador centralizado únicamente estudia la disponibilidad de las próximas regiones por las que debe avanzar cada robot si éstos han llegado a sus respectivos puntos objetivo de las últimas secuencias enviadas, lo que se verifica con el sistema de visión. Por lo tanto, se clasifican los robots en cada iteración en tres estados diferentes:

- Robot en *modo activo*: el robot todavía no ha alcanzado el punto objetivo de la última secuencia enviada, por lo que el controlador ni siquiera ejecuta la parte del algoritmo destinada a generar la siguiente secuencia para no reservar una región por la que podrían circular otros robots.
- Robot en *modo espera:* no se le ha enviado ninguna trayectoria ya que el siguiente punto al que debe avanzar está ocupado por otro robot. Se le permite acceso constantemente a la generación de secuencias para que cuando la región por la que debe circular quede libre se envíe al robot su siguiente destino.
- Robot *recién llegado*: el robot acaba de llegar a su punto objetivo de la última secuencia. El controlador permite acceso a la evaluación de las siguientes regiones para comprobar su disponibilidad, mediante la ejecución del algoritmo de generación de secuencias.

La función clave del controlador es garantizar que los dos robots no avancen hacia/por la misma región al mismo tiempo. El procedimiento consiste en evaluar en cada iteración si la siguiente región está libre, en cuyo caso el controlador envía tal destino al robot, o está ocupada. El controlador toma esta decisión de la siguiente forma, haciendo uso de los parámetros de la red S4PR.

1. Sea la posición actual de $R_i$la región c_i. La región posterior en la trayectoria completa es c_{i+1}.
2. El controlador busca la existencia de lugares de recursos r_m o monitor p_k que restringe la capacidad de c_{i+1}.

3. Una vez seleccionados los recursos o monitores que restringen la capacidad de c_{i+1}se comprueba si éstos están disponibles.
    a. Si todos los recursos involucrados disponen de al menos una marca, el controlador permite el acceso de R_i a c_{i+1}(configurándose en *modo activo*), y reduce en uno el número de marcas de tales lugares.
    b. Si, por el contrario, el marcado de alguno de los lugares de recurso o monitor que restringen la capacidad de c_{i+1} es cero, el controlador entiende que la región está siendo atravesada por otro robot R_j o es inminente una situación de bloqueo si avanza hasta esta región y configura a R_i en *modo espera*.
4. El controlador envía los paquetes de trayectorias generados.
5. Se supone ahora que R_i avanza por c_{i+1} , restringida por el el lugar r_m. Una vez que el robot abandone la región (información proporcionada por el sistema de visión), el recurso r_m debe ser liberado para que el otro robot $R_j$pueda atravesar c_{i+1}.
6. Los recursos se liberan aumentando en uno el número de marcas de los lugares que los modelan. El robot $R_i$ha completado su trayectoria y se establece en modo *recién llegado.*

Es necesario resaltar que, para un conjunto de robots $R = \{R_1, R_2, \ldots, R_n\}$ la prioridad de acceso a una región se otorga en función del orden de su definición. Por ejemplo, dado el caso en el que R_1y R_2 requieren acceso para atravesar una región c al mismo tiempo, el controlador centralizado concede prioridad a R_1y establece a R_2 en modo de espera. Otras políticas de prioridad se pueden establecer.

PSEUDOCÓDIGO DEL CONTROLADOR CENTRALIZADO

Entrada: Conjunto de trayectorias $Q = \{q^{R_1}, q^{R_2}, \ldots, q^{R_n}\}$ del conjunto de robots $R = \{R_1, R_2, \ldots, R_n\}$, con $q^{R_i} = \{q_{init}^{Ri}, q_x^{Ri}, q_{x+1}^{Ri}, \ldots, q_{goal}^{Ri}\}\ \forall\ 1 \leq i \leq n, x \in [init, goal]$

Definiciones:

- Q_{send} conjunto de trayectorias que se envían a los robots, de la misma estructura que Q:
    - $Q_{send} = \{q_{send}{}^{R_1}, q_{send}{}^{R_2}, \ldots, q_{send}{}^{R_n}\}$
    - $q_{send}{}^{R_i} = \{q_{send}{}_{(init)}^{Ri}, q_{send}{}_{(x)}^{Ri}, q_{send}{}_{(x+1)}^{Ri}, \ldots, q_{send}{}_{(goal)}^{Ri}\}$
- $Q \rightarrow N = \{P, T, F, W, M_0\}, P = P_0 \cup P_S \cup P_R \cup P_M;\ P_M = \{p_k\}.$
- $P_{R,involved}$ →conjunto de recursos (en los que se incluyen lugares monitor) que restringen la capacidad de un lugar de proceso $P_{S_j} \in P_S$.
- $controlAccess = \{controlAccess_{R1}, \ldots, controlAccess_{Rn}\}$, $\forall\ controlAccess_{Ri} = \{0/1\}$. Permite al controlador evaluar el siguiente punto de la trayectoria de un robot si éste ha alcanzado el último punto de la anterior secuencia enviada.
    - R_i en modo activo ⇒ $controlAccess_{Ri} = false$;
    - R_i en modo espera ⇒ $controlAccess_{Ri} = true$;
    - R_i en modo recién llegado ⇒ $controlAccess_{Ri} = true$;

1 **forall** $R_i \in R$ **do {** //q_{init}^{Ri} son los puntos iniciales, por lo que se consideran alcanzados.

2 $q_{send}{}^{Ri} = q_{send}{}^{Ri} \cup q_{init}^{Ri}$;

3 R_i en modo espera ⇒ $controlAccess_{Ri} = true$**; }**

4 **while** ($q_x^{Ri} \neq q_{goal}^{Ri} \forall\ i = 1, \ldots, n$) **do** {

5 **forall** $R_i \in R$ **do {**

6 **if (**$controlAccess_{Ri}$**) {**

7 $q^{Ri}: q_x^{Ri} \rightarrow P_{S_j} \quad \in P_S$

8 **forall** $r_m \in (P_R \cup P_M)$ **do {**

9 $P_{R,involved} = P_{R,involved} \cup r_m$; **}**

10 **forall** $r_m \in P_{R,involved}$ **do {**

11 **if** ($M_0: P \rightarrow \{r_m\} \geq 1$) **{**

12 **pick_token**: $M_0(\ r_m\ ) = M_0(\ r_m\ ) - 1$;

13 $q_{send}{}^{Ri} = q_{send}{}^{Ri} \cup q_x^{Ri}$;

14 $R_i$modo activo;

15 **} else {**R_i modo de espera; **} }** // end forall

16 **}**// end if **}** // end forall

17 **if** ! Q_{send}.empty () **{** send_packet (Q_{send})**;}**

18 **forall** $R_i \in R$ **do {**

19 **if** ($R_i$ha atravesado q_x^{Ri}) **{**

20 **forall** $r_m \in P_{R,involved}$ **do {**

21 **return_token**: $M_0(\ r_m\ ) = M_0(\ r_m\ ) + 1$**;}**

22 **clear** ($q_{send}{}^{Ri}$);

23 $q_x^{Ri} = q_{x+1}^{Ri}$;

24 $R_i$en modo recién llegado ⇒ $controlAccess_{Ri} = true$**;**

25 **} else if (!**Ri en modo de espera;**) {** //si no está en espera, está en activo

26 $R_i$en modo activo ⇒ $controlAccess_{Ri} = false$;

27 **clear** ($q_{send}{}^{Ri}$); **}** // Se elimina la trayectoria porque ya se ha mandado.

28 **}** end forall **} // end while**

5. SIMULACIONES Y EXPERIMENTOS

La planificación de trayectorias y el método de evitación de colisiones y prevención de bloqueos han sido puestos a prueba en la plataforma multi-robot del laboratorio. En primer lugar, se ha comprobado el funcionamiento de la parte de planificación con un único robot, para posteriormente proceder al caso multi-robot, incluyendo el controlador centralizado.

A) PLANIFICACIÓN DE TRAYECTORIAS

Dado el espacio de trabajo[7] mostrado en la Figura 27, el robot R1 debe alcanzar el punto del espacio (280,100), marcado en la imagen con una X en color negro, evitando los obstáculos presentes y ejecutando la ruta óptima en términos de distancia.

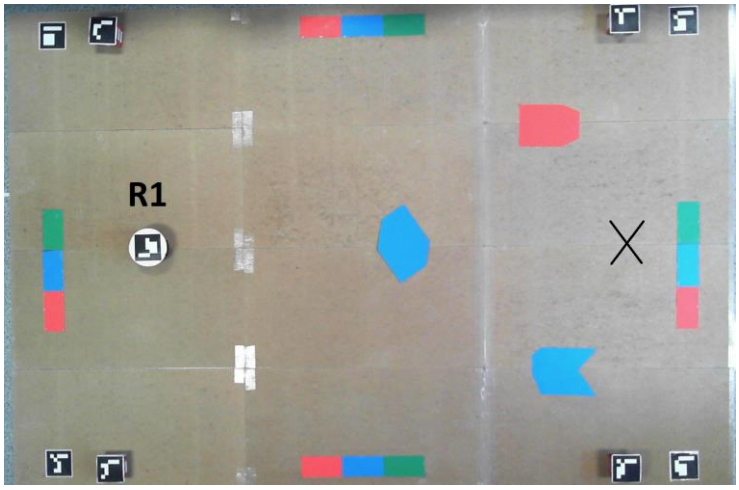

Figura 27. Experimento propuesto en la plataforma multi-robot del laboratorio. Objetivo: R1 debe alcanzar el punto marcado con una X desde su situación inicial.

Tras ejecutar la planificación de trayectorias, se obtiene el siguiente conjunto de posibles caminos representado en la Figura 28, siendo el más corto el de color verde. Tal y como se puede observar, el robot alcanza el punto final introducido evitando los obstáculos.

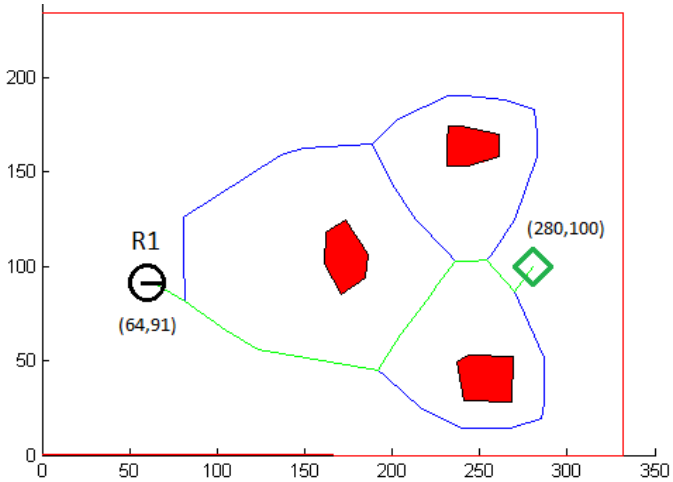

Figura 28. Resultado de la planificación de trayectorias basada en el diagrama de Voronoi. En rojo, los obstáculos y los límites del espacio de trabajo. En azul, el conjunto de posibles caminos para el robot R1. En verde, la trayectoria óptima. Nota: el punto inicial del robot R1 es proporcionado por el sistema de visión, así como la orientación, representada por la línea recta en color negro. Fuente: Matlab.

[7] Componentes de la plataforma multi-robot explicados en el apartado 2 de este trabajo.

B) PLANIFICACIÓN DE TRAYECTORIAS Y EVITACIÓN DE COLISIONES

Dado el espacio de trabajo mostrado en la Figura 29, el robot R1 debe alcanzar el punto (280,20), marcado con una X verde en la imagen, mientras que el robot R2 debe llegar hasta el (20,20), marcado con una X en morado.

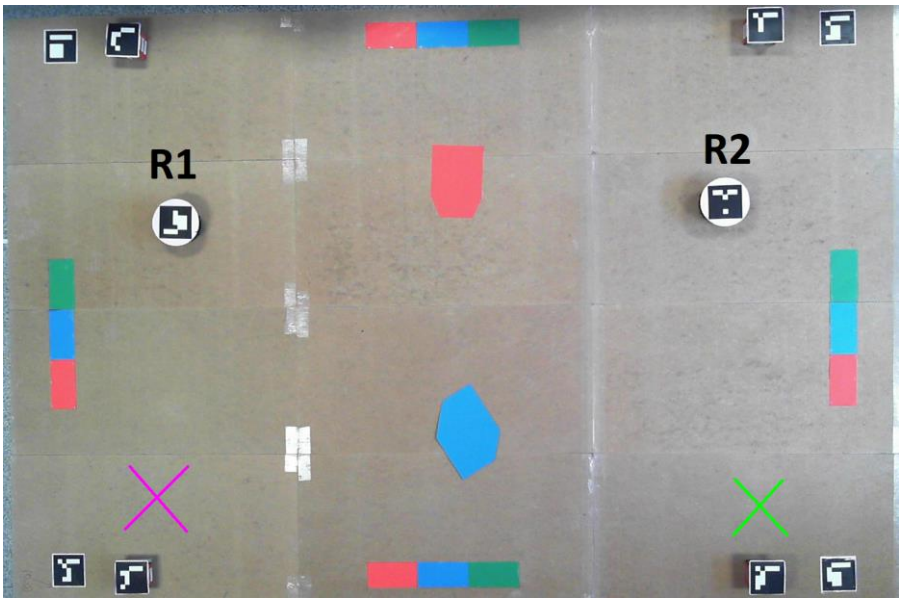

Figura 29. Experimento propuesto en la plataforma multi-robot del laboratorio. Objetivo: R1 debe alcanzar el punto marcado con una X en verde desde su situación inicial. R2 debe alcanzar el punto marcado con una X en mordo desde su situación inicial.

Tras ejecutar la planificación de trayectorias, se obtiene el siguiente conjunto de posibles caminos mostrado en la Figura 30, entre los cuales se destacan las trayectorias óptimas para R1 y R2. Ambos robots alcanzarían su destino evitando los obstáculos del entorno.

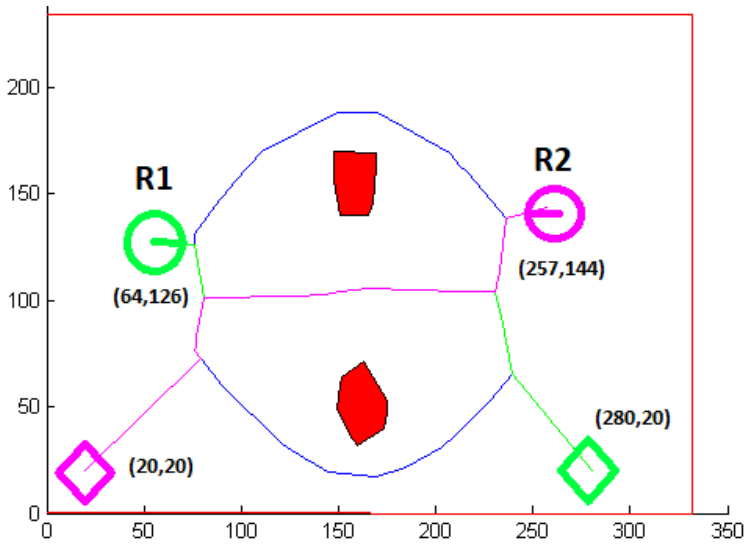

Figura 30. Resultado de la planificación de trayectorias basada en el diagrama de Voronoi. En rojo, los obstáculos y los límites del espacio de trabajo. En azul, el conjunto de posibles caminos para los robots. En verde, la trayectoria óptima para R1. En morado, la trayectoria para R2. Nota: los puntos iniciales de los robots son proporcionados por el sistema de visión, así como las orientaciones, representadas por las líneas rectas en el color correspondiente a cada robot.

Debido a que ambas trayectorias coinciden en varios puntos, el controlador centralizado debe actuar con el objetivo de evitar colisiones entre R1 y R2. Se modelan las trayectorias según la siguiente red de Petri S4PR:

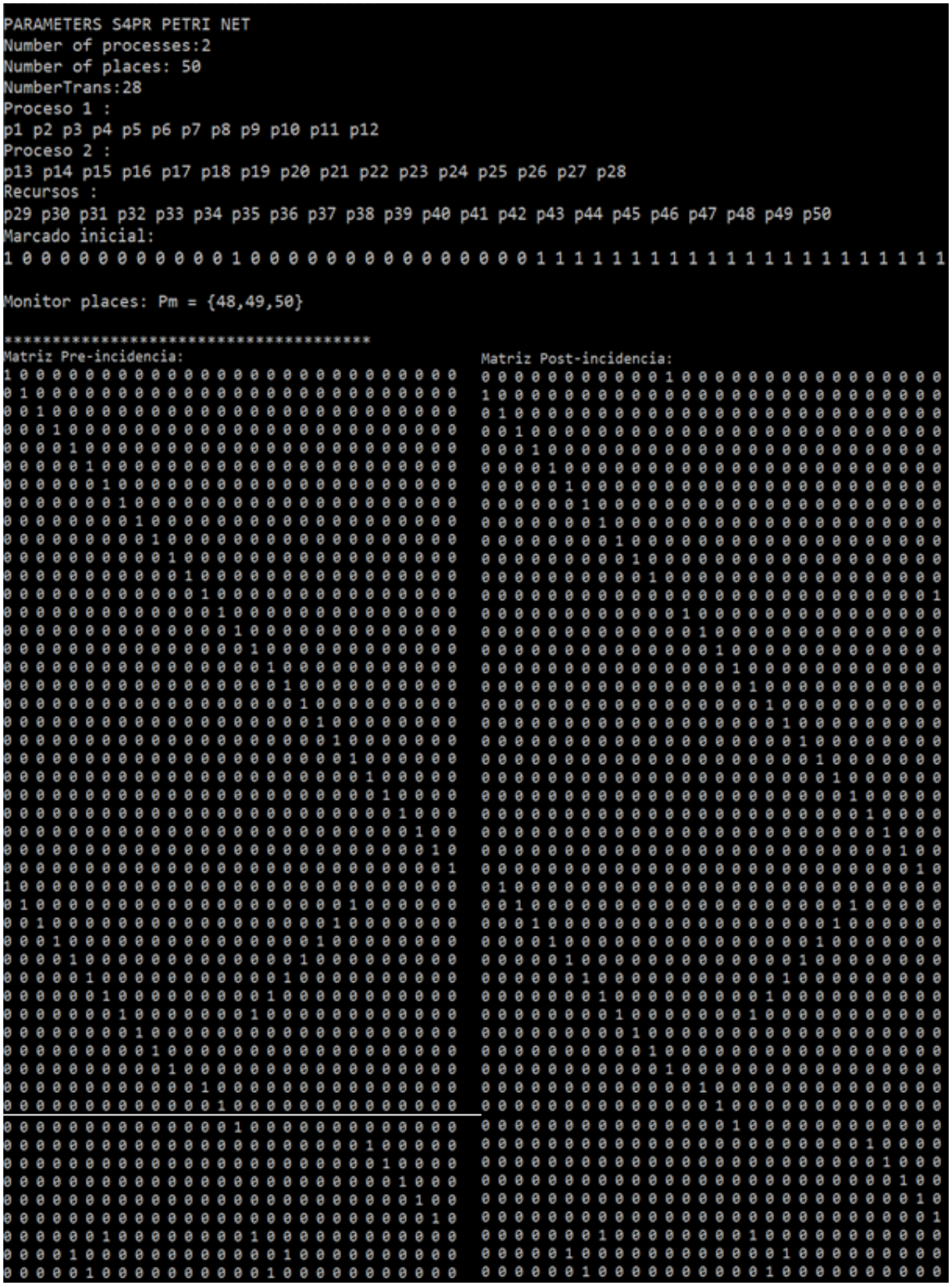

Figura 31. Abstracción del conjunto de trayectorias en una red de Petri S4PR.

Con esta información el controlador centralizado ejecuta el algoritmo detallado en el apartado 4.3 de este trabajo, hasta que todos los robots alcancen sus configuraciones finales, asegurando que no se produzca ninguna colisión entre ellos e impidiendo situaciones de bloqueo durante la evolución del sistema.

6. CONCLUSIONES

El trabajo realizado supone una solución que permite la ejecución de ciertas tareas desarrolladas por equipos de robots móviles que se comunican con una estación central. La solución propuesta abarca desde la discretización del espacio de trabajo para calcular las trayectorias de los robots, hasta el diseño e implementación de un controlador centralizado cuyo fin es evitar las colisiones entre los miembros del equipo.

Aunque la idea original requiere la combinación de ambos métodos (planificación y control), ya que el objetivo principal era garantizar las tres condiciones de seguridad explicadas en la introducción, su uso es posible por separado. Por ejemplo, el controlador centralizado actúa sobre un conjunto de trayectorias que pueden ser calculadas con otros algoritmos de planificación tales como la descomposición en celdas.

Se ha mejorado la plataforma de robots, permitiendo realizar experimentos y simulaciones con mayor precisión y variedad que ponen a prueba el funcionamiento de los métodos implementados. Los experimentos realizados con la presencia de un equipo formado por dos robots móviles fueron satisfactorios. Aunque teóricamente debería funcionar también para un conjunto compuesto por más de dos robots, al no haberse comprobado en la plataforma no se puede afirmar con total seguridad.

Este trabajo supone un complemento a un proyecto mayor como es el desarrollo de la plataforma multi-robot, al que varios alumnos han contribuido estos últimos años. El siguiente punto de partida, que supondría una mejora respecto a este trabajo, podría consistir en la panificación de alto nivel en la cual la especificación para el equipo de robots es una tarea más complicada y no solo de alcanzabilidad de algunas regiones.

7. BIBLIOGRAFÍA

[1] Mark H. Overmars. Path Planning for Games. Institute of Information and Computing Sciences. Utrecht University, The Netherlands.

[2] https://www.dfrobot.com/wiki/index.php/Romeo_V2-All_in_one_Controller_(R3)_(SKU:DFR0225); https://www.dfrobot.com/product-844.html

[3] Emanuele Vitolo. Multi-robot platform for path planning and control using high-level specifications: Implementation and experiments. TFM, University of Salerno, 2016.

[4] Funcionamiento teórico de la función convex_hull, Boost Geometry Library C++ https://www.boost.org/doc/libs/1_55_0/libs/geometry/doc/html/geometry/reference/algorithms/convex_hull.htm

[5] H. Choset, K. M. Lynch, S. Hutchinson, G. Kantor, W. Burgard, L. E. Kavraki, and S. Thrun. Principles of Robot Motion: Theory, Algorithms, and Implementations. MIT Press, Boston, 2005. Page 520.

[6] S. M. LaValle. Planning Algorithms. Cambridge, 2006. ISBN 0521862051

[7] C. Mahulea and M. Kloetzer, "Robot Planning based on Boolean Specifications using Petri Net Models, "IEEE Transactions on Automatic Control, 63(7): 2218-2225, July 2018.

[8] S. M. LaValle. Rapidly-Exploring Random Trees: A New Tool for Path Planning. Technical Report TR 98-11, Computer Science Dept. Iowa State University, October 1998.

[9] Boost Library C++: https://www.boost.org/

[10] Librería para la construcción en C++ del diagrama de Voronoi, Boost Polygon Library C++: https://www.boost.org/doc/libs/1_69_0/libs/polygon/doc/voronoi_diagram.htm

[11] Librería para el uso de grafos en C++, Boost Graph Library C++: https://www.boost.org/doc/libs/1_34_0/libs/graph/doc/graph_theory_review.html#sec:shortest-paths-algorithms

[12] Librería para el uso de algoritmos de búsqueda en grafos en C++, Boost Graph Library C++: https://www.boost.org/doc/libs/1_42_0/libs/graph/doc/dijkstra_shortest_paths.html

[13] M. Kloetzer, C. Mahulea and J.M. Colom, "Petri net approach for deadlock prevention in robot planning," In ETFA'2013: 18th IEEE International Conference on Emerging Technologies and Factory Automation, Cagliari, Italy, September 2013.

[14] J. Ezpeleta, F. García-Vallés, and J.-M. Colom, “A class of well-structured petri nets for flexible manufacturing systems,”in ICATPN, ser. Lecture Notes in Computer Science, J. Desel and M. Silva, Eds., vol. 1420. Springer, 1998, pp. 64–83.

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

[16] Javier Oroz, Diseño e implementación de un método para evitar colisiones en un sistema multi-robot. TFG en el Grado de Tecnologías Industriales, Universidad de Zaragoza, Noviembre 2016.

[17] 4.3 Ing Control-SED_RdP. Asignatura Ingeniería de Control. M.Silva. 2017

ANEXOS

ANEXO 1: FUNDAMENTOS DE REDES DE PETRI

Red de Petri [15]**:** Una red de Petri es una terna $\mathcal{N}$ = {P, T, F, W, M_0}, donde:

1) P = {$p1$, $p2$,...,pm} es un conjunto finito de lugares;
2) T = {$t1$, $t2$,...,tn} es un conjunto finito de transiciones;
3) $F \subseteq (P \times T) \cup (T \times P)$ es un conjunto de arcos;
4) W: $F \rightarrow$ {1, 2, 3,...} es la función de ponderación de los arcos;
5) M_0: $P \rightarrow$ {0, 1, 2, 3,...} es la función del marcado inicial.

El marcado de una red de Petri define el estado de la red y cambia según ley de ejecución de transición (o de sensibilización y disparo):

- Se dice que una transición t está sensibilizada 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.
- Una transición sensibilizada se puede o no se puede disparar dependiendo de si el evento externo relacionado ocurre o no.
- El disparo de una transición elimina $W(p,t)$ marcas de cada lugar de entrada p de t, y añade W(t,p) marcas en cada lugar de salida p de t.

Matriz Pre: matriz de pre-incidencias de dimensión P x T, tal que $Pre[p,t] = W(p,t)$. Por ejemplo, si $Pre[i,j] = 1 \Rightarrow \exists$ un arco de peso 1 que conecta el lugar p_i con la transición t_j.

Matriz Post: matriz de post-incidencias de dimensión P x T, tal que $Post[p,t] = W(t,p)$. Si $Post[i,j] = 1 \Rightarrow \exists$ un arco de peso 1 que conecta la transición t_j con el lugar p_i.

Red S4PR: Subclase de las redes de Petri. [15]

Siendo I_N = {1,2,...,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 $\mathcal{N}$ = ⟨P,T,$\boldsymbol{C}$⟩ donde:

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

Los recursos compartidos pertenecen al conjunto $P\mathrm{r}$ y se les denomina con la letra r.

ANEXO 2: CLASE INPUT_S4PR

La clase Input_S4PR se encuentra definida en los archivos Petri_Net.h y Petri_Net.cpp. Su implementación en C++ tiene dos objetivos principales:

1. Obtener el modelado de las trayectorias de los robots calculadas en la parte de planificación. En concreto se obtienen los parámetros de la red de Petri, destacando la creación de recursos compartidos que son los responsables de la evitación de colisiones.
2. Actuar como interfaz entre el programa de la plataforma multi-robot y la clase S4PR creada por Javier Oroz en su TFG [16], que permite la obtención de lugares monitor para prevenir situaciones de bloqueo.

DESCRIPCIÓN DE FUNCIONES Y VARIABLES TIPO MIEMBRO:

VARIABLES:

- *num_procesos:* número de procesos de la red de Petri, que será igual al número de trayectorias procedentes de la parte de planificación (una por robot).
- *num_lugares:* número total de lugares. Representa también el número de filas de las matrices Pre y Post.
- *num_transiciones:* número total de transiciones. Representa también el número de columnas de las matrices Pre y Post.
- *processes:* vector de dos dimensiones de tipo entero. Almacena los lugares de cada uno de los procesos de una forma ordenada, según su secuencia.
- *m_0:* marcado inicial de la red de Petri. Es un vector de tipo entero, en el que un 0 implica que el lugar correspondiente no está marcado inicialmente, y un valor n implica que el lugar correspondiente tiene n marcas al inicio.
- *pre:* matriz de dimensión num_lugares x num_transiciones en la que cada elemento representa el peso de los arcos que conectan los lugares con las transiciones.
- *post:* matriz de dimensión num_lugares x num_transiciones en la que cada elemento representa el peso de los arcos que conectan las transiciones con los lugares.
- *lugar_1*: Esta variable indica el número total de lugares definidos hasta el momento. Su valor se inicializa a 0.
- *resources:* vector tipo *int* que almacena los lugares recurso.
- *regiones:* vector de tipo *int* que almacena las regiones identificadas a lo largo del programa. Es de gran utilidad a la hora de obtener los lugares de recursos compartidos
- *capacity_by_region:* representa la capacidad a la que se restringe cada región. Al tratarse también de un vector de tipo *int*, a la región que ocupa la posición i en el vector regiones, le corresponde la capacidad que se guarda en la posición i en este vector.
- *idle:* vector que guarda los lugares idle de la red de Petri.

FUNCIONES

- *Constructores:* descritos en el apartado 4.4.
- *r_not_created:* devuelve una variable de tipo booleano, y se utiliza para la generación de recursos compartidos. Devuelve *true* si el recurso todavía no ha sido creado, y *false* si ya lo ha sido. Si se supone que una región no puede aparecer más de dos veces entre las dos trayectorias calculadas, entonces esta función es innecesaria. Sin embargo, se ha

implementado con el objetivo de que esta clase sirva también para casos en los que los robots deban regresar a su posición inicial tras alcanzar los puntos objetivo.

- *set_to_null_matrix():* inicializa las matrices Pre y Post a 0, es decir, todos sus elementos con valor 0.
- *add_minus_1:* esta función se ha implementado para cumplir con los requerimientos de los parámetros de entrada de la clase S4PR, la cual exige que el vector que almacena los recursos y el que almacena los procesos, tengan el valor -1 en su última posición.
- *is_idle():* devuelve *true* si se trata de un lugar idle, y *false* en caso contrario. Es preciso diferenciar este tipo de lugares de los demás porque su marcado inicial es 1, y no precisan de lugares recurso ya que son lugares de reposo.
- get_nprocesses(): devuelve el valor de la variable *num_procesos.*
- get_dimP(): devuelve el valor de la variable *num_lugares.*
- get_dimT(): devuelve el valor de la variable *num_transiciones.*
- get_Pre(): devuelve el valor de la variable *pre.*
- get_Post(): devuelve el valor de la variable *post.*
- get_processes(): devuelve el valor de la variable *processes.*
- get_m_0(): devuelve el valor de la variable *m_0.*
- get_resources(): devuelve el valor de la variable *resources.*
- printParameters(): escribe por pantalla los parámetros de la red de Petri.

FUNCIONAMIENTO

Antes de proceder a la implementación de esta clase en el programa principal de la plataforma, se ha comprobado su funcionamiento de manera independiente.

Para su compilación se ha descargado el compilador MinGW, dado que para esta parte el programa usado inicialmente (Microsoft Visual Studio) generaba algunos errores incomprensibles, mientras que para su ejecución se ha usado la consola de Windows (Símbolo de escritorio).

En primer lugar, se ha de crear una carpeta que incluya todos los archivos necesarios para la ejecución del programa, al menos uno de ellos debe contener la función *main* para que pueda ejecutarse. En este caso, la carpeta se denomina PetriNetCPP y contiene los siguientes archivos:

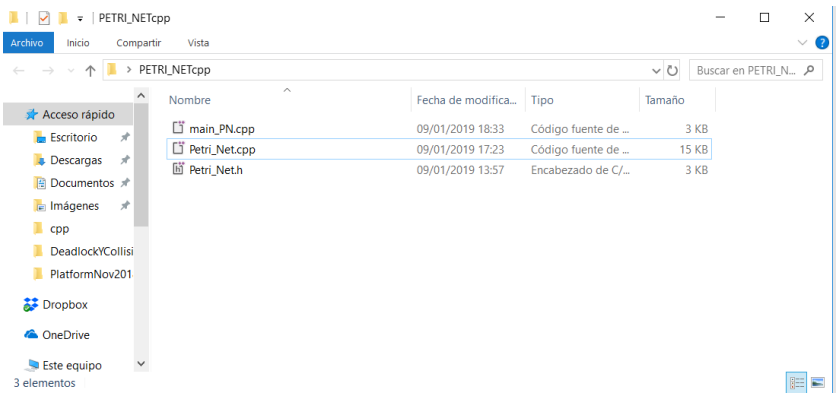

Figura 32. Carpeta que contiene los ficheros implementados.

- *Petri_Net.h* y *Petri_Net.cpp* corresponden a la definición de la clase *Input_S4PR*

- *Main_PN.cpp* contiene la función *main*.

En la consola de Windows cambiamos el directorio de trabajo a la carpeta que se acaba de crear, y compilamos los archivos con la orden que aparece en la imagen siguiente.

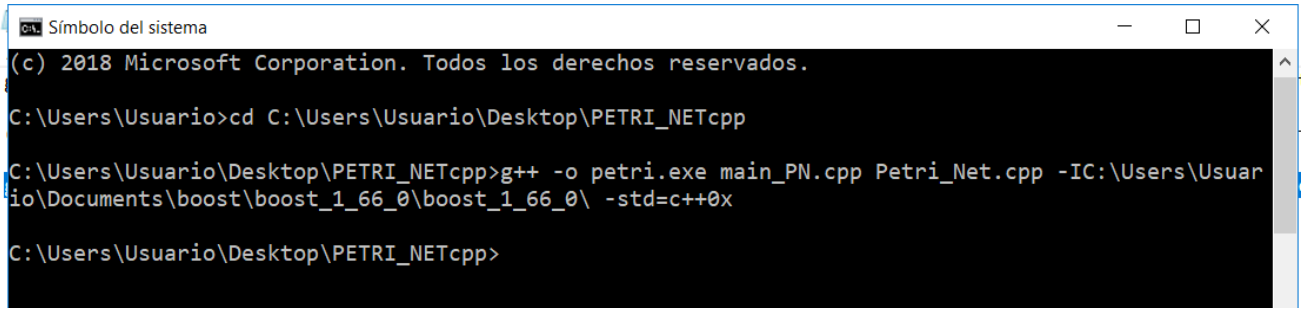

Figura 33. Comando de compilación.

- g++ es la llamada al compilador.
- petri.exe es el nombre del archivo ejecutable que se crea en la carpeta sobre la que se está trabajando.
- A continuación, se incluyen todos los archivos .cpp que se deseen compilar.
- Por último, se incluye la dirección de las librerías externas de las que se hace uso en los archivos.

Por último, se ejecuta el programa como se detalla a continuación:

1. Se escribe el nombre del archivo ejecutable: petri.exe
2. A continuación, se escribe la letra "D". La idea es poder utilizar diferentes tipos de entrada a la función, cada uno de ellos se identifica con una letra mayúscula.
3. Ahora se escribe la secuencia ordenada de la primera trayectoria sin espacios. Cada región se escribe con la letra "c" seguida del número que la identifica.
4. La siguiente cadena de caracteres a introducir son las capacidades de las regiones de la trayectoria anterior, de nuevo sin espacios. Debe ser de igual tamaño que la trayectoria a la que corresponde. Ejemplo:
   c1c2c3c4 1112 significa que las regiones c1, c2, c3 tienen capacidad 1, mientras que c4 tiene capacidad 2.
5. Se introduce la siguiente trayectoria, seguido de las capacidades correspondientes, tal y como se indica en los dos puntos 3 Y 4.

Para el caso del ejemplo desarrollado en el apartado 4.1:

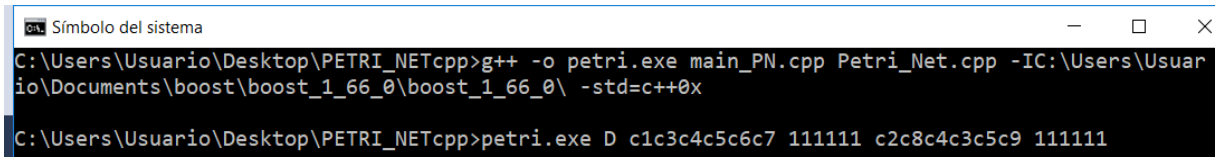

Figura 34. Ejemplo de introducción de dos trayectorias con las capacidades que restringen las regiones.

Al ejecutarlo, se obtienen los siguientes parámetros de la red de Petri:

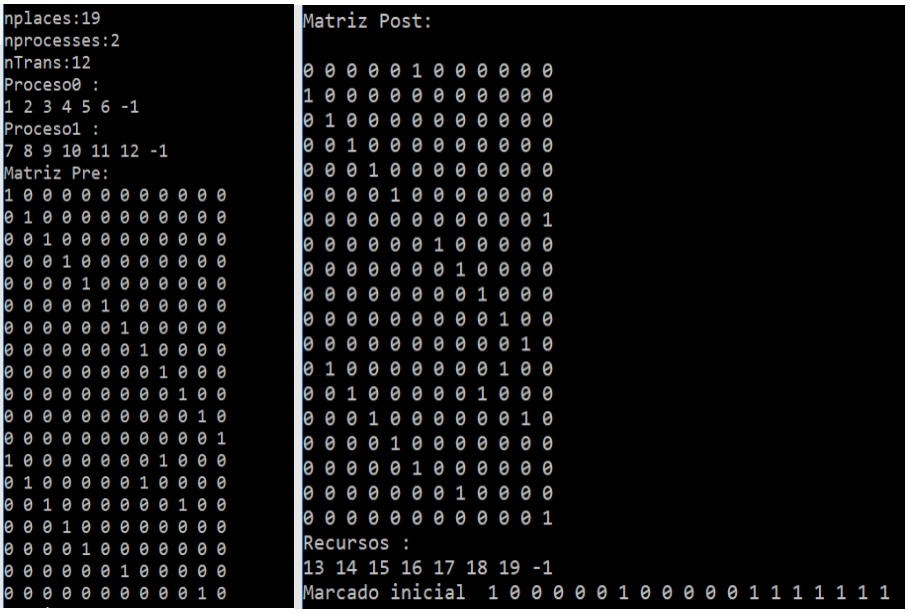

Figura 35. Resultado obtenido de la ejecución del programa.

NOTAS:

- Si se introducen de forma incorrecta los datos, el sistema lanzará una excepción.
- Si por confusión, se introducen dos capacidades diferentes para una misma región, una vez llamado el constructor de la clase *Input_S4PR,* se asignará el máximo de las dos. Por ejemplo:
  Sean dos trayectorias y sus vectores capacidad:
  1. c1c3c4c5 1111
  2. c2c4c3c6 1121

Se asignará a la región 3 capacidad 2.

ANEXO 3: INSTALACIÓN DE MICROSOFT VISUAL STUDIO, CREACIÓN DE UN NUEVO PROYECTO E INCLUSIÓN DE LIBRERÍAS

En primer lugar, se descarga el software que servirá como entorno de programación: Microsoft Visual Studio. En la página https://visualstudio.microsoft.com/es/, en *Descargas,* se selecciona la versión correspondiente a IDE Visual Studio Community 2017.

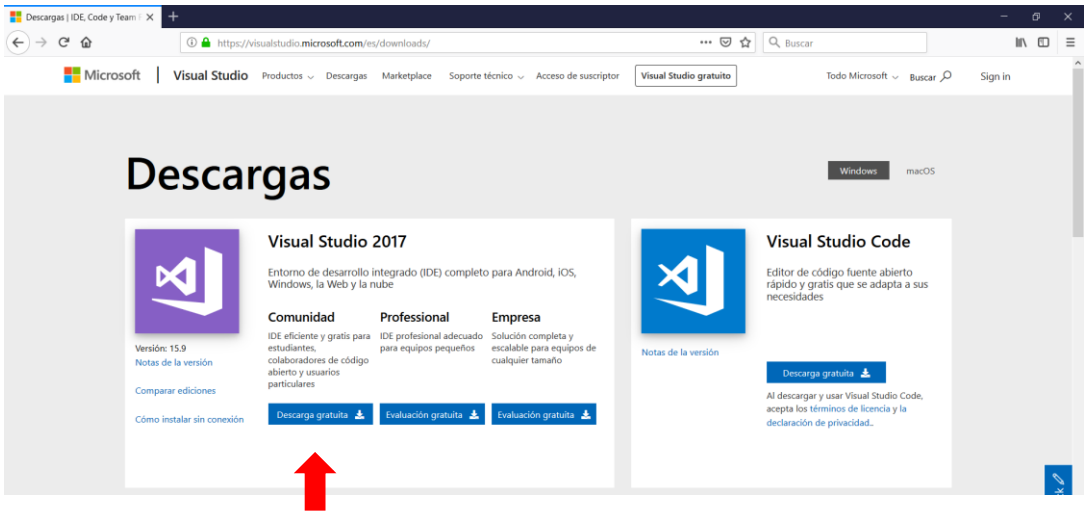

Figura 36. Descarga de Microsoft Visual Studio.

A continuación, es necesario guardar y ejecutar el archivo de programa previo para instalar el Instalador de Visual Studio: *vs_community__699115128.1538063275.exe*

Previamente a proceder a la instalación completa, el programa ofrece seleccionar las cargas de trabajo requeridas. En el tutorial de C++ [XX], al que se ha acudido varias veces a lo largo del desarrollo de este trabajo, se recomienda la siguiente personalización:

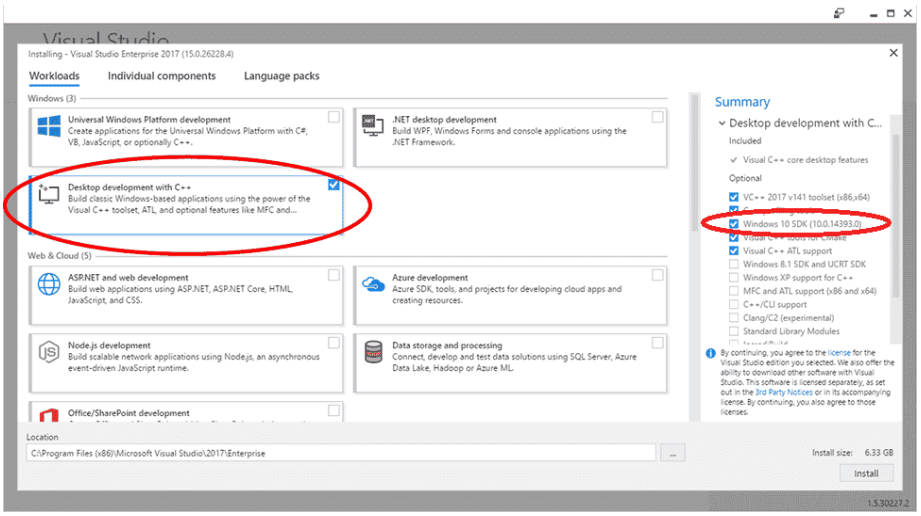

Figura 37. Instalación de Microsoft Visual Studio.

CREACIÓN DE UN NUEVO PROYECTO:

Tras ejecutar el programa, se seleccionan las siguientes opciones en la barra de herramientas:

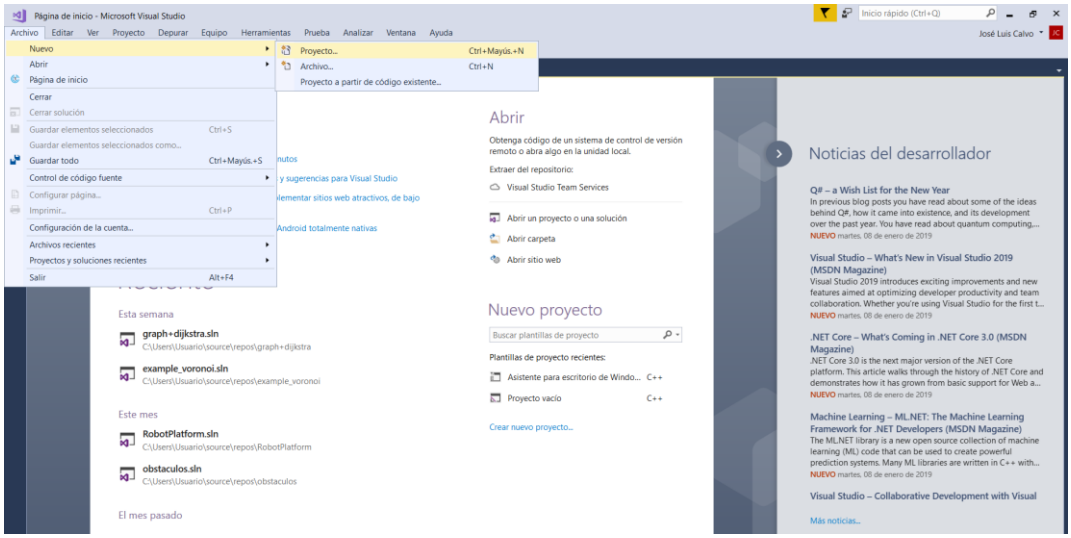

Figura 38. Creación de nuevo proyecto en Microsoft Visual Studio.

Se introduce el nombre del proyecto y se selecciona "Asistente para escritorio de Windows".

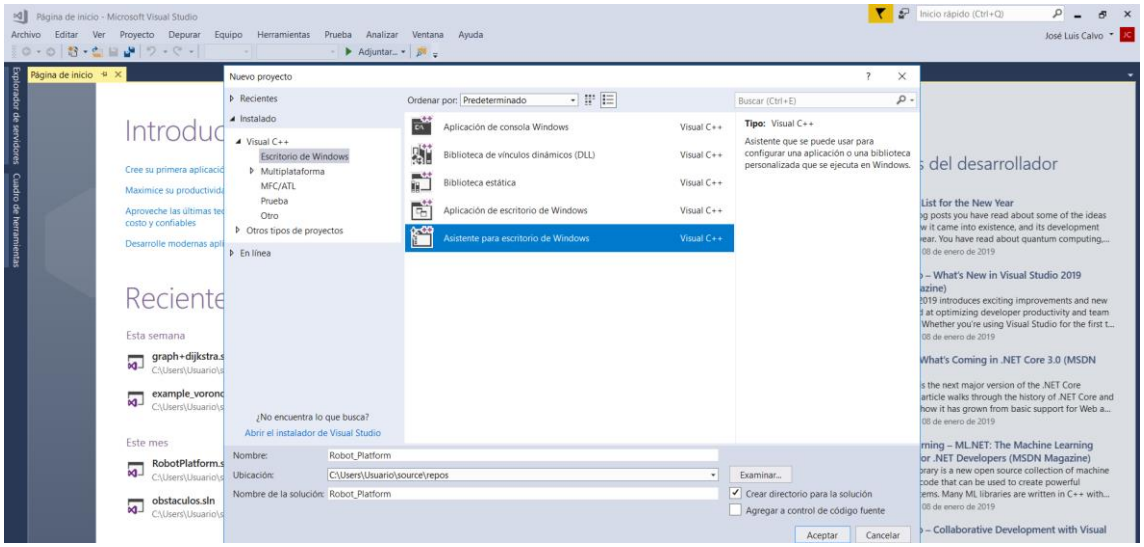

Figura 39. Creación de nuevo proyecto en Microsoft Visual Studio.

El proyecto ya está creado, con su respectiva carpeta dentro del directorio de trabajo. En ella, por el momento, únicamente existe un archivo .cpp, de mismo nombre que el proyecto, donde se encuentra la función *main*. Si se desea introducir archivos existentes, es necesario copiarlos en dicha carpeta.

INCLUSIÓN DE LIBRERÍAS

Para incluir el uso de librerías, se selecciona en primer lugar “Propiedades del proyecto”. En “Propiedades” de configuración, en el subapartado “C/C++”, se selecciona “General”.

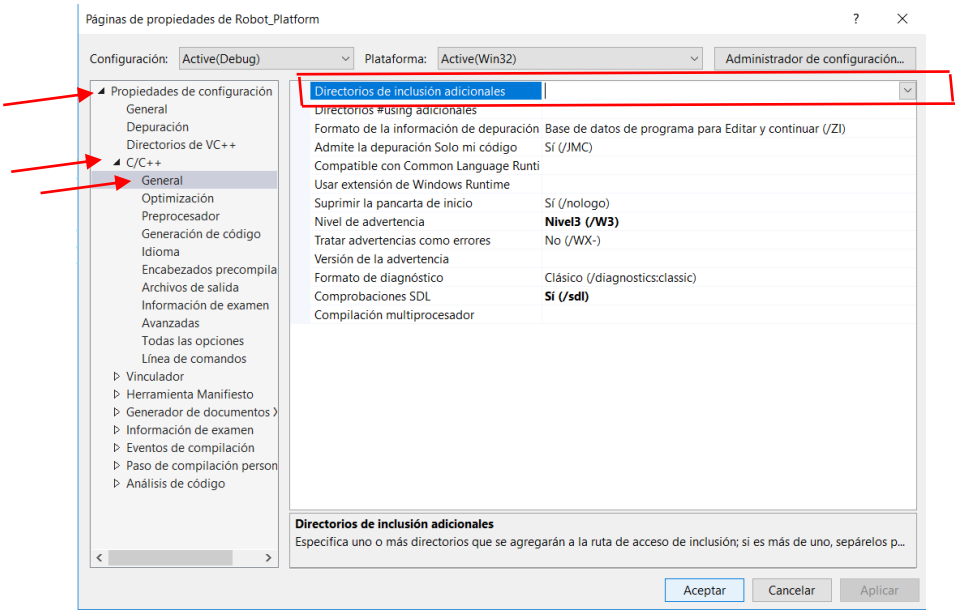

Figura 40. Inclusión de librerías en un proyecto en Microsoft Visual Studio.

En la línea Directorios de inclusión adicionales, se introduce la dirección donde está guardada la librería en el equipo.