INGENIERÍA TÉCNICA
EN INFORMÁTICA DE SISTEMAS

Curso Académico 2009/2010

Proyecto Fin de Carrera

PLANIFICACIÓN CUALITATIVA DE CAMINOS DE ROBOTS MÓVILES EN ENTORNOS POLIGONALES BASADA EN LA TEORÍA DE MATROIDES ORIENTADAS

Autor: Roberto Solana Hernández

Tutor: Ernesto Staffetti

Resumen

La planificación de caminos es uno de los principales problemas de la robótica. En este proyecto fin de carrera se estudia la planificación del camino de uno o varios robots móviles, en el que cada uno tiene que moverse desde una posición inicial a una final, evitando los obstáculos.

Los obstáculos se representan en el plano en que se mueven los robots, mediante polígonos. Las rectas soporte de las aristas de estos polígonos inducen una partición del plano en celdas. Este conjunto de celdas será el espacio de búsqueda de los caminos, que serán secuencias de celdas adyacentes que unen las celdas que contienen las posiciones iniciales y finales.

Para representar el entorno en que los robots se mueven, se usará una estructura llamada matroide orientada, la cuál guarda propiedades combinatorias tales como el orden, la separación, convexidad de los vértices de los polígonos que representan obstáculos.

Esta estructura proporcionará al robot información local y global para la planificación del camino, sin ser necesaria información métrica, como puedan ser las coordenadas de los vértices. Los robots se suponen de tipo uniciclo.

Para resolver el problema se utilizará el Algoritmo A*.

Índice general

Capítulo 1

Introducción

En este proyecto se estudia la planificación del camino que deben realizar uno o varios robots móviles para llegar desde una posición inicial, a una posición final conocidas, evitando los obstáculos. El espacio en el que se mueven los robots será un plano.

El espacio de trabajo estará formado por obstáculos conocidos por el robot. Los obstáculos se modelizan mediante diferentes formas: polígonos cerrados simples, curvas poligonales simples y segmentos.

Los polígonos cerrados simples estarán formados por tantos vértices como aristas, y podrán ser de dos tipos: convexos o no convexos.

Las curvas poligonales simples son un conjunto de segmentos unidos por vértices, pero sin ser cerrado, y sin cruzarse ningún segmento. El número de vértices será uno mayor que la cantidad de aristas.

Aparte pueden encontrarse segmentos libres.

Además de los obstáculos, suponemos que el espacio de los robots contienen landmarks, que son puntos de referencia.

La Figura. 1.1 es un ejemplo de modelización de los obstáculos en el espacio de trabajo del robot.

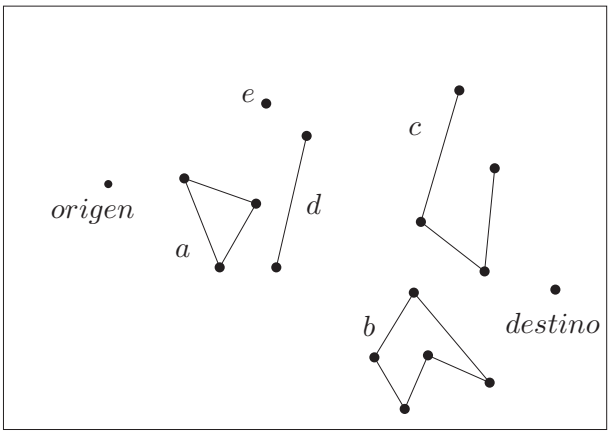

Figura 1.1: Modelización de los obstáculos en el espacio de trabajo del robot. Se puede distinguir diferentes tipos de obstáculos: a) polígono simple convexo, b) polígono simple cóncavo, c) curva poligonal, d) segmento. También se incluyen landmarks (e).

Para la representación matemática del entorno, utilizaremos una representación llamada matroide orientada, la cuál es capaz de almacenar propiedades combinatorias como puedan ser el orden, la separación y la convexidad de los puntos que representen los vértices y landmarks. En particular, se usa una estructura llamada "hyperline sequence"que almacena la posición angular relativa de estos puntos.

A lo largo del problema, no utilizaremos información métrica como puedan ser las coordenadas, sino que utilizaremos solo la información cualitativa almacenada en las hyperline sequences. Con esto se demostrará que con conocer la posición relativa de los objetos del entorno, es suficiente para encontrar un camino que lleve a uno o más robots de una posición inicial a una posición final. Al igual que los humanos, que somos capaces de resolver el mismo problema, con la posición relativa de los objetos que nos rodean, para llegar desde un punto a otro, sin ser necesario saber las coordenadas en las que nos encontramos dentro del espacio.

La estructura hyperline sequences generará una partición del espacio de trabajo del robot en celdas. Estas celdas serán polígonos irregulares convexos con número de aristas y vértices

variable.

Las rectas que unen a todos los puntos que son vértices de los obstáculos y landmarks entre sí, podrán ser o no soporte de las aristas de algún obstáculo. Si es recta soporte de una arista de un obstáculo, el segmento de la recta que forme parte de la arista de un obstáculo, no podrá ser atravesada. El resto de la recta, se podrá atravesar.

Las rectas las denotaremos mediante los puntos que contiene, es decir la recta que une los punto p_2 y p_3 será la recta $l_{2,3}$.

En el ejemplo de la Figura 1.2 se ve como, el segmento $e_{b,c}$, es un segmento soporte de una arista de un obstáculo. Si un punto, por ejemplo p_2, intenta atravesar el segmento $e_{b,c}$, no podrá, debido a que es un obstáculo. En cambio si el punto p_1 quiere atravesar una de las dos semirectas externas al segmento $e_{b,c}$, este si podrá atravesarla, ya que la semirecta que quiere atravesar no forma parte del segmento $e_{b,c}$.

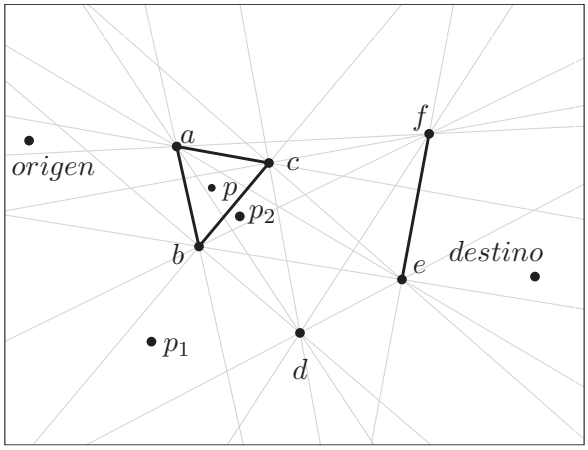

Figura 1.2: En este ejemplo, vemos como queda particionado el plano en celdas, si el espacio poligonal es creado por un polígono cerrado convexo con vértices a, b, c, un segmento con vértices e, f y un landmark representado por d. Se muestran posibles situaciones de diferentes robots en el espacio de búsqueda.

En alguna ocasión, si un punto está situado dentro de una celda que a su vez está dentro de un obstáculo cerrado, este no podrá llegar a un destino que se sitúe fuera del obstáculo. En este caso, si el robot fuese representado por el punto p, este no podría salir del obstáculo formado

por los vértices *abc*.

El conjunto de todas las celdas obtenidas, formarán el espacio de búsqueda del problema.

Una vez se haya particionado el plano, el robot deberá moverse desde la celda en que está situado, a la celda en la que está situado el punto destino. Para llegar hasta el destino, tendrá que moverse de celda en celda, siendo éstas siempre adyacentes y separadas por una pared que pueda ser atravesada.

En nuestro problema trataremos casos en el que no haya una gran cantidad de obstáculos. En el caso de que hubiese muchos obstáculos, la partición del plano, quedaría muy complicada, ya que se crearían gran cantidad de celdas. En ese caso, para disminuir la complejidad del problema, se trabajaría con una representación en mutiresolución del entorno. Dependiendo del movimiento que esté realizando el robot, trabajará a diferentes niveles de resolución en el que cada nivel de resolución contenga solo los elementos del entorno necesarios para ejecutar la tarea asignada. En nuestro problema, solo realizaremos una representación, en la que trataremos todos los obstáculos detalladamente.

1.1. Trabajos previos

Los métodos clásicos para resolver el problema de planificación de caminos de robots móviles, son los métodos basados en el diagrama de Voronoi, en el grafo de visibilidad y en la descomposición vertical.

1.1.1. Método basado en grafo de visibilidad

El método de planificación basado en el grafo de visibilidad se desarrolla en un entorno poligonal. Los vértices de los obstáculos poligonales, la posición inicial del robot (q_{init}), y la posición final del robot (q_{goal}), serán los elementos que generan el espacio de búsqueda, el cuál se creará mediante los segmentos que unen a todos ellos, sin que los segmentos atraviesen ningún obstáculo. El conjunto de estos segmentos, será por donde deba moverse el robot, para ir del punto inicial a su destino. En general se busca el camino mínimo que tenga menor número de segmentos. Para determinarlo se utiliza el algoritmo de Dijkstra.

En la Figura 1.3 se puede observar un ejemplo que se basa en el método de grafo de visibilidad, en el que el camino mínimo es marcado con una línea más gruesa.

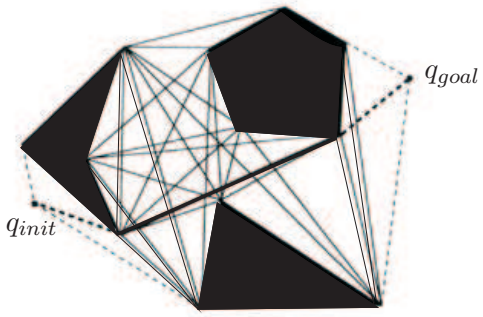

Figura 1.3: Ejemplo de planificación de camino mediante el método del grafo de visibilidad.

Las principales diferencias con nuestro caso, son que en nuestro caso:

- la posición inicial y final del robot, no influyen en la partición del plano,
- las rectas que inducen la partición, atraviesan cualquier obstáculo,
- nuestro espacios de búsqueda se basa en celdas, y no en segmentos, y
- se utiliza el algoritmo A* en lugar del algoritmo de Dijkstra.

1.1.2. Método basado en el diagrama de Voronoi

Dada una configuración de puntos, el diagrama de Voronoi se crea al unir los puntos entre sí, y luego trazando las mediatrices de los segmento que se obtienen. Las intersecciones de estas mediatrices determinan una serie de polígonos en un espacio bidimensional alrededor del conjunto de los puntos, de manera que el perímetro de los polígonos generados, sea equidistante a los puntos vecinos. Estas celdas creadas se llaman polígonos de Thiessen o polígonos de Voronoi.

En la Figura 1.4 se muestra el diagrama de Voronoi, cuando en el plano solo hay puntos.

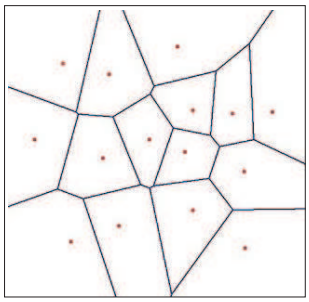

Figura 1.4: Diagrama de Voronoi para una configuración de puntos en el plano.

En la Figura 1.5, se puede ver como también se puede calcular el diagrama de Voronoi para un conjunto de polígonos.

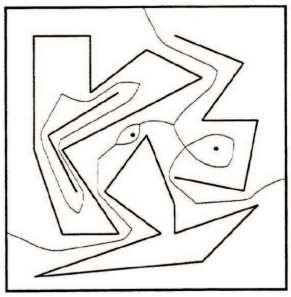

Figura 1.5: Diagrama de Voronoi para un conjunto de polígonos en el plano.

1.1.3. Método de descomposición vertical

El método de la descomposición vertical es un método muy usado en la partición de entornos poligonales, para la planificación de caminos de robots móviles. Para realizar la partición, se traza una recta vertical desde cada vértice de los polígonos. Estas rectas no atravesarán ningún obstáculo. Las celdas resultantes serán de forma triangular o en forma de trapezoide. El robot deberá avanzar de celda en celda, siendo estas adyacentes, para encontrar el camino.

En general, el camino se busca sobre el grafo, y se obtiene uniendo todos los puntos medios de los segmentos de las celdas adyacentes.

En la Figura 1.6 se muestra el camino mínimo para un robot, en un entorno en el que se ha

realizado una descomposición vertical.

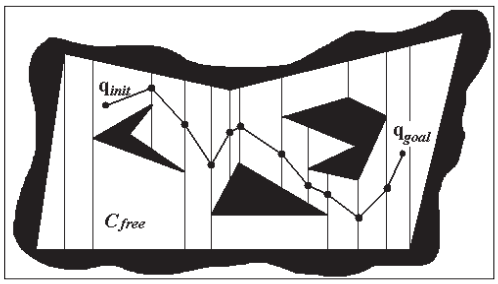

Figura 1.6: Camino mínimo para el robot, en un entorno particionado por el método de descomposición vertical.

Este método tiene dos variaciones conocidas: la descomposición cilíndrica y la descomposición triangular.

Descomposición cilíndrica.

La descomposición cilíndrica es muy similar a la partición vertical, con la única diferencia de que las rectas que se trazan para particionar el plano, no se detienen cuando se encuentran alguna pared de un obstáculo, sino que son infinitas. Se ve que en una misma tira vertical, habrá celdas que podrán ser visitadas, y habrá celdas que pertenezcan a los obstáculos, por lo que no podrán ser tratadas. Esta variación es menos eficiente que la descomposición vertical. En la Figura 1.7 se muestra un ejemplo de partición.

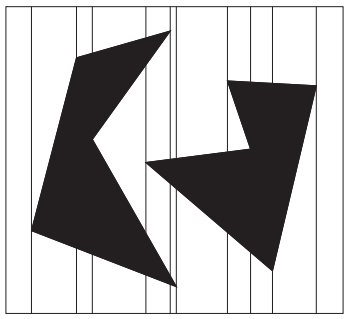

Figura 1.7: Partición del entorno según el método partición cilíndrica.

Descomposición triangular.

Este método trata en descomponer el entorno en celdas triangulares. Hay muchos métodos para poder realizar esta partición.

En la Figura 1.8 se muestra un caso, en el que se forma un grafo, usando los baricentros de las celdas triangulares, y el punto medio de las aristas de los triángulos. Sobre este grafo se busca el camino mínimo.

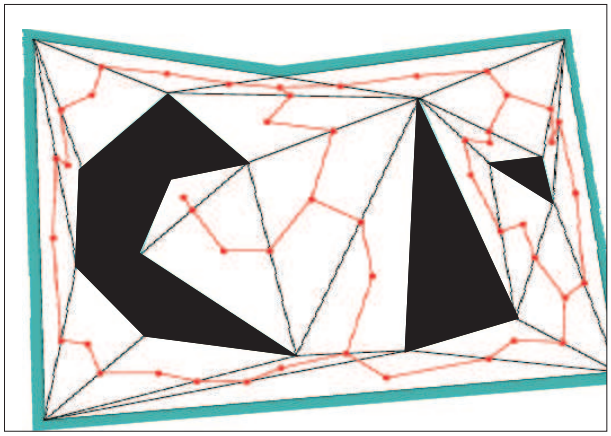

Figura 1.8: Partición del entorno utilizando el método de descomposición triangular.

Como hemos visto, el tema de la planificación de caminos para robots móviles, ya ha sido estudiado en la literatura científica, pero ninguno de ellos se ha realizado a través de la teoría de matroides orientadas.

Los dos libros más importantes relacionados con la planificación de caminos en la robótica, son el libro de Steven La Valle [2] y el libro de Jean-Claude Latombe [3].

Un planteamiento similar al nuestro, se describe en [5], el cuál estudia el problema de la planificación de un robot móvil, sin usar información métrica. En este trabajo se introduce el concepto del orden cíclico teóricamente, pero a diferencia de nuestro caso, no utilizan la teoría de las matroides orientadas para así poder obtener una representación matemática del entorno. Al no utilizar una estructura matemática, el razonamiento espacial está basado en reglas, por lo que no podrán llegar a obtener información de forma sistemática, como la que nos ofrece las

hyperline sequences (cuales son la paredes de una celda, si las paredes pueden ser obstáculo o no, cuántos movimientos serían necesarios para ir de una celda a otra, etc...)

La teoría de matroides orientadas, ha sido tratada en proyectos fin de carrera anteriores realizados por alumnos de la Universidad Rey Juan Carlos, [1], [4], en ambos casos aplicada al problema del reconocimiento de objetos en visión por computador.

Capítulo 2

Objetivos

En esta sección se describe el problema estudiado de encontrar un camino libre de obstáculos para uno o varios robots dentro de un entorno poligonal, que estará formado por obstáculos y landmarks. Además, suponemos que estos obstáculos serán conocidos por todos los robots involucrados. El objetivo es poder hacerlo utilizando solo información cualitativa, sin llegar a utilizar información métrica.

Los robots no tendrán en cuenta las posiciones de los demás robots que estén planificando el camino, por lo que el tratamiento del problema es el mismo, independientemente del número de robots involucrados en la planificación.

Durante la descripción del problema y de su método de resolución, los puntos representan varios elementos como los vértices de los obstáculos, los landmarks y la posición del robot.

Tendremos que almacenar matemáticamente nuestra configuración de puntos, usando la estructura de hyperline sequences.

Una vez calculada, ignoraremos la información métrica, y simplemente trataremos la información cualitativa almacenada en las hyperline sequences.

Esta estructura de datos, generará una partición del plano en celdas poligonales convexas, las cuáles formarán el espacio de búsqueda. El punto que representa al robot, se irá moviéndo de celda en celda, siempre que estas sean adyacentes y no estén separadas por una pared que sea un obstáculo.

El objetivo es que el punto que representa al robot encuentre un camino hasta el destino,

que esté libre de obstáculos, y que tenga el menor número de celdas, independientemente del tamaño de éstas. Para ir de una celda a otra celda adyacente, el punto que representa al robot, tendrá que cruzar la recta que las separa, por lo tanto, el camino mínimo se representará como la secuencia de las rectas que ha tenido que atravesar el robot.

2.1. Representación matemática

Para estudiar el problema, necesitamos representar matemáticamente el espacio poligonal que representa el entorno en que se mueve el robot. Para ello utilizaremos la teoría de matroides orientadas, usando la estructura de datos llamada hyperline sequences.

2.1.1. Matroide orientada e hyperline sequences

Una matroide orientada es una representación sin coordenadas de una configuración planar de puntos, que contiene propiedades combinatorias como puedan ser el orden, la convexidad y la separación de los puntos de la configuración.

En esta sección, las matroides orientadas serán introducidas utilizando la estructura de datos hyperline sequences, que representa la principal herramienta para traducir problemas geométricos en este formalismo.

Una hyperline sequence es un vector que almacena la posición angular relativa de un conjunto de puntos, con respecto al punto que se esté tratando, y cuya longitud será el número de puntos de nuestra configuración.

Nuestra configuración geométrica estará formado por todos los puntos que representen los vértices de los obstáculos, los landmarks que forman el espacio de trabajo y la posición actual del robot.

El etiquetado de estos puntos será un etiquetado aleatorio, con la única restricción de que las etiquetas de los vértices de los obstáculos sean consecutivos lexicográficamente. Se empezará a etiquetar desde el número p_1, en orden ascendente.

En la Figura 2.1 podemos ver un ejemplo de espacio de trabajo, y la configuración de puntos resultante.

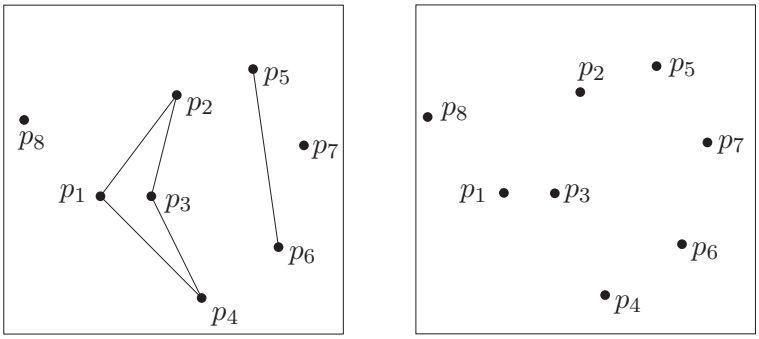

Figura 2.1: A la izquierda, espacio poligonal creado por un polígono convexo con vértices p_1, p_2, p_3, p_4, un segmento con vértices p_5 y p_6, y un landmarks representado por p_7, y la posición del robot representada por p_8. A la derecha, representación de la configuración de puntos.

La hyperline sequences será una matriz de $N \times N$, siendo N el número de puntos que haya en la configuración. La hyperline sequences será el conjunto de hyperline sequence individual de cada punto.

A continuación se explica cómo obtener la hyperline sequence de cada punto.

Una vez tengamos la configuración de puntos de la Figura 2.1, tenemos que diferenciar dos casos: el cálculo de la hyperline sequence del punto p_1, y el cálculo de la hyperline sequence de cualquier otro punto distinto de p_1.

Caso 1: Cálculo de la hyperline sequence del punto p_1.

Para calcular la hyperline sequence del punto p_1, trazaremos una recta que una el punto p_1 con el punto p_2. La semirecta que va desde el punto p_1 al punto p_2, tiene orientación positiva, mientras que la otra semirecta tiene orientación negativa.

Una vez que tengamos la recta que une ambos puntos, la giraremos con eje de giro en p_1, en sentido contrario a las agujas del reloj, y se irán añadiendo al vector todos los puntos que se vaya encontrando la recta. Si la recta se encuentra con un punto con la parte positiva, este es almacenado en el vector con una etiqueta positiva, en cambio, si son encontrados con la parte negativa de la recta, se les etiquetará con un signo negativo delante.

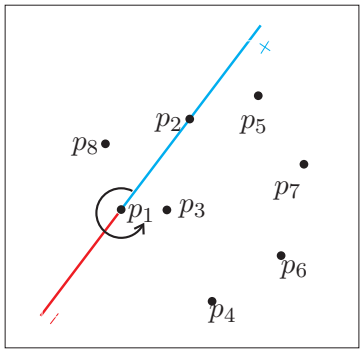

Figura 2.2: Método para calcular la hyperline sequence del punto p_1.

La hyperline sequence del punto p_1 obtenida de la Figura 2.2 será:

1	2	8	-4	-6	-3	-7	-5

Caso 2: Cálculo de la hyperline sequence de un punto distinto de p_1.

Para calcular la hyperline sequence de cualquier punto que no sea p_1, trazaremos una recta desde el punto del que se quiere calcular la hyperline sequence al punto p_1. Una vez tengamos la recta, realizaremos el mismo paso que en el caso anterior, con el cambio de que el eje de giro de la recta será el punto del cual se quiere calcular la hypeline sequence.

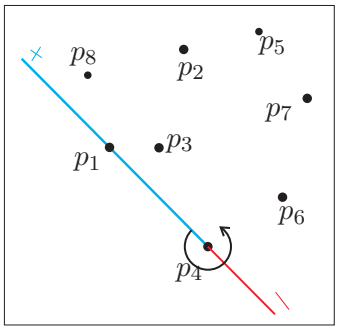

Figura 2.3: Método para calcular la hyperline sequence del punto p_4.

En la Figura 2.3 se muestra el caso de como calcular la hyperline sequence del punto p_4.

La hyperline sequence del punto p_4 obtenida de la Figura 2.3 será:

4	1	-6	-7	-5	-2	-3	-8

De esta manera todos los puntos, excepto el punto p_1, al primer punto que almacena en su hyperline sequence será p_1. Esto quiere decir que, en todas las hyperline sequence (excepto en la hs_1), el segundo elemento del vector será el punto p_1.

La hyperline sequences será el conjunto de las hyperline sequence individual de todos los puntos. Se tendría que aplicar el segundo caso a todos los puntos excepto al punto p_1, quedando la hyperline sequences de la siguiente forma:

Hyperlines sequences							
1	2	8	-4	-6	-3	-7	-5
2	1	3	4	6	7	-8	5
3	1	-7	-5	-2	4	-8	6
4	1	-6	-7	-5	-2	-3	-8
5	1	3	4	6	7	-8	-2
6	1	4	-7	-5	-2	-8	-3
7	1	3	4	6	-5	-2	-8
8	1	4	3	6	7	2	5

Cuadro 2.1: Hyperline sequence de la configuración planar de los puntos representada en la Figura 2.1.

2.2. Espacio de búsqueda

Una vez se haya representado matemáticamente en las hyperline sequences la configuración de puntos que representan vértices y landmarks, se generará una partición del espacio en celdas. Esta partición se producirá por el cruce de las rectas que unan todos los vértices de los obstáculos y landmarks entre sí.

Las rectas serán etiquetadas, usando los dos puntos que contiene, siendo el primero el que tenga una etiqueta menor, es decir, la recta que una los puntos p_1 y p_4 se etiquetará como $l_{1,4}$.

Las celdas serán polígonos convexos, de un tamaño variable, y con un número variable de aristas que llamamos paredes. El conjunto de todas las celdas, será el espacio de búsqueda.

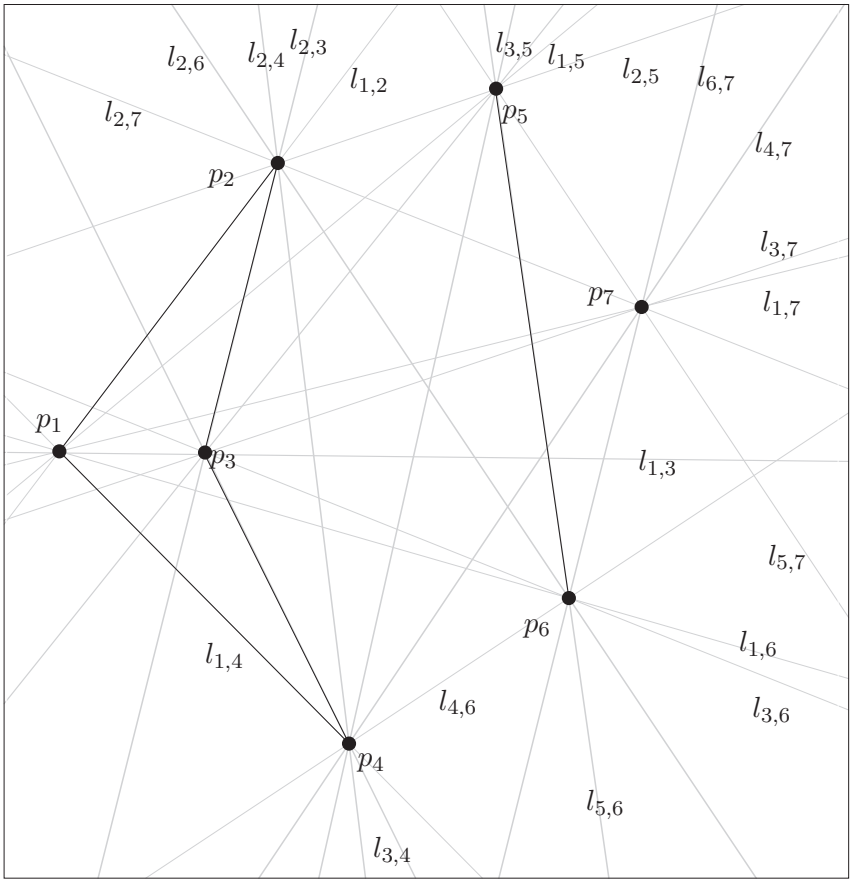

Figura 2.4: Partición del plano en celdas.

2.3. Características de las celdas de la partición

Para el algoritmo es necesario saber cuáles son las paredes que delimitan a la celda que contiene la posición del robot, y sus celdas adyacentes. Gracias a los valores almacenados en la hyperline sequences, podremos saber cuáles son dichas paredes.

Se ha dicho anteriormente que las hyperline sequences representan cualitativamente la información geométrica del problema. La posición exacta en que estén situados los puntos de nuestra configuración, no es importante. Lo importante es la posición relativa del conjunto de puntos, almacenado en la hyperline sequences. Si la posición relativa no cambia, los valores de la hyperline sequences no cambian.

Es decir, cada punto puede moverse sin que los valores de la hyperline sequences cambien. Con estos movimientos, aparecerían y desaparecerían celdas en el espacio de búsqueda. Además, la forma de muchas celdas variaría, apareciendo nuevas paredes, o desapareciendo algunas paredes que ya tenía.

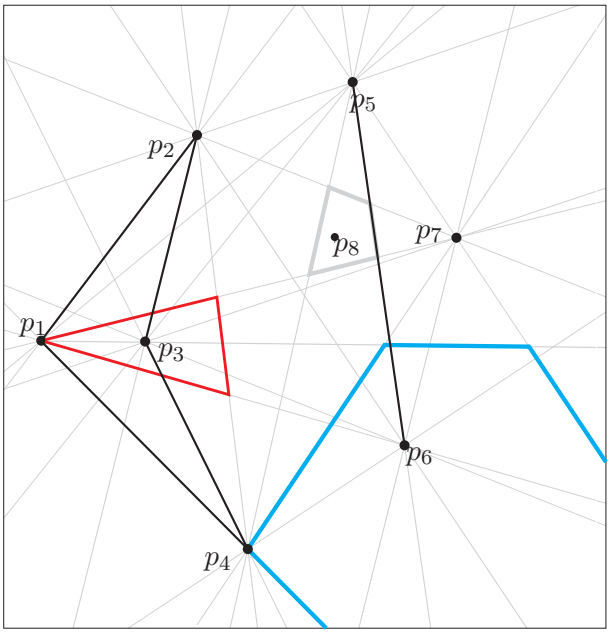

Figura 2.5: Límites en los que se pueden desplazar los vértices de los obstáculos y los landmarks, sin que cambie la representación en la hyperline sequences.

El límite del movimiento de un punto sin que cambien las hyperline sequences, es la celda en la que este situado, como se muestra en la Figura 2.5. Además, se ve que la celda donde está situado p_8, está formada por las cuatro paredes $l_{1,7}$, $l_{2,7}$, $l_{4,5}$, $l_{5,6}$.

En la Figura 2.6, el punto p_6 se ha movido con respecto a la Figura 2.5, sin producir cambios en los valores de las hyperline sequences. Se puede observar que el número de paredes de la celda donde está situado p_8 ha variado, apareciendo una nueva pared $e_{2,6}$. Además han aparecido nuevas celdas, como la celda de color rojo.

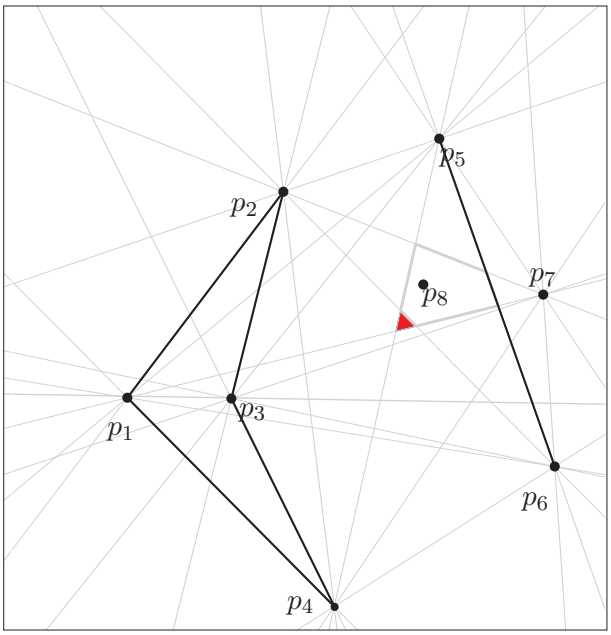

Figura 2.6: Espacio de búsqueda creado tras un cambio de posición del punto p_6 con respecto a la Figura 2.5, sin provocar cambio en las hyperline sequences, pero con la aparición de nuevas celdas en el espacio de búsqueda y de nuevas paredes en la celda de p_8.

2.4. Actualización de la hyperline sequences

Cuando el robot atraviesa una recta para cambiar de celda, tendremos que actualizar la hyperline sequences que representa el entorno.

Supongamos que el robot está representado por el punto p_k, y que p_i y p_j son dos puntos del entorno que definen la recta $l_{i,j}$.

Una vez que p_k atraviese la recta $l_{i,j}$ para avanzar a alguna celda adyacente, la representación mediante hyperline sequences sufrirá cambios en las tres hyperline sequence hs_i,hs_j,hs_k.

El concepto clave para realizar los cambios, es saber que una hyperline sequence, por como está construida, es una estructura circular.

El tratamiento en todas las hyperline sequence será igual, excepto para la hyperline sequence del punto p_1. Podrán darse varios casos dependiendo si la recta que atraviesa contiene a p_1 o no le contiene.

Caso 1: Cuando la recta que atraviesa no contiene al punto p_1.

Si p_k ha atravesado una recta $l_{i,j}$, los cambios que se deben hacer en la hyperline sequences sería:

- En hs_i los puntos p_j y p_k intercambian su posición.

- En hs_j los puntos p_i y p_k intercambian su posición.

- En hs_k los puntos p_i y p_j intercambian su posición.

En el ejemplo de la Figura 2.6, el punto p_8 va a avanzar hacia la celda adyacente que les separa la recta $l_{4,5}$, por lo que en la hyperline sequences sufriremos cambios hs_4,hs_5 y hs_8. En este caso, dichas hyperline sequence antes de producirse el cambio eran:

4	1	-6	-7	-8	-5	-2	-3

5	1	3	4	8	6	7	-2

8	1	3	4	-5	6	-2	7

y tras el cambio pasan a ser:

4	1	-6	-7	-5	-8	-2	-3

5	1	3	8	4	6	7	-2

8	1	3	-5	4	6	-2	7

Caso 2: Cuando la recta que se atraviesa contiene al punto p_1.

Supongamos que tengamos p_1, p_j, y p_k. La hs_1 se trataría de la siguiente manera:

- Si ninguno de los otros dos puntos implicados p_j y p_k coinciden con p_2, en la hs_1 se realizará un intercambio de posiciones de los puntos p_j y p_k, como en el caso anterior.
- Si p_j o p_k coinciden con p_2, se trataría de la misma forma que en los siguientes casos.

Caso 2.1: Cuando la recta que se atraviesa contiene al punto p_1, y p_k se encuentra en la última posición de la hs_j.

Este caso se aplicaría en la Figura 2.5 en la que el punto p_8 se dispone a atravesar la recta $l_{1,7}$.

Si calculamos las hyperline sequences para los tres puntos, tendríamos que:

- Para la hs_7 , el punto p_8 estaría en la última posición, con orientación negativa.
- Para la hs_8, el punto p_7, estaría en la última posición, con orientación positiva.
- Para la hs_1, los puntos p_7 y p_8, estarían en posiciones contiguas.

Las hyperline sequence hs_1, hs_7 e hs_8, serían:

1	2	-4	-6	-3	-7	-8	-5

7	1	3	4	6	-5	-2	-8

8	1	3	4	-5	6	-2	7

Una vez realizado la transición, vemos como quedan posicionados todos los puntos en la imagen derecha de la Figura 2.7 .

Como vemos, en la hs_7 y en la hs_8, ahora el punto situado en última posición, pasa a la primera posición, pero cambiando de orientación. El resto de puntos en la hyperline sequence, seguirá con el mismo orden, pero habiéndose desplazado todos una posición a la derecha.

Para la hs_1, simplemente se produce un intercambio de posiciones entre p_7 y p_8.

Las hyperline sequences resultantes serían estas:

1	2	-4	-6	-3	-8	-7	-5

7	1	8	3	4	6	-5	-2

8	1	-7	3	4	-5	6	-2

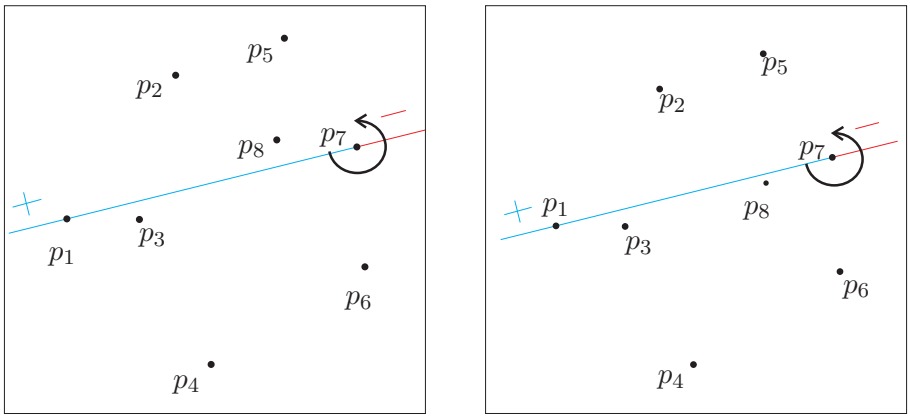

Figura 2.7: Posición de los puntos antes y después de que el punto p_8 atraviese la recta $l_{1,7}$.

Caso 2.2 : Cuando la recta que se atraviesa contiene al punto p_1, y p_k se encuentra en la primera posición de la hs_j.

En este ejemplo, la posición inicial será la imagen derecha de la Figura 2.7, en la que p_8 se dispone a atravesar la recta $l_{1,7}$ para llegar a la posición de la imagen izquierda de la Figura 2.7.

Este caso es el inverso del caso 2.1, es decir, que p_k está en la primera posición de la hs_j. Una vez realice la transición, este pasará a estar en la última posición, cambiado de signo. El resto de puntos, seguirán en el mismo orden, pero moviéndose todos una posición hacia la izquierda.

Las hs_1, hs_7, hs_8 de antes de atravesar la recta, tendría los siguientes valores:

1	2	-4	-6	-3	-8	-7	-5

7	1	8	3	4	6	-5	-2

8	1	-7	3	4	-5	6	-2

realizando una transformación para quedar :

1	2	-4	-6	-3	-7	-8	-5

7	1	3	4	6	-5	-2	-8

8	1	3	4	-5	6	-2	7

2.5. Caracterización de las paredes de las celdas

En este apartado explicaremos, como establecer si una pared de una celda tiene posibilidades de ser obstáculo o no.

Los obstáculos poligonales están compuestos por aristas que definen rectas que generan la partición del plano. Cada recta queda subdividida en dos semirectas y un segmento. En este caso, el segmento no podrá ser atravesado.

Pero las rectas que particionan el plano, pueden proceder también de puntos que no corresponden a ninguna una arista de un obstáculo. En este caso, el segmento que una el par de puntos, si puede atravesarse.

En la Figura Figura 2.8 se verán 3 posiciones diferentes para un robot, dentro del mismo entorno.

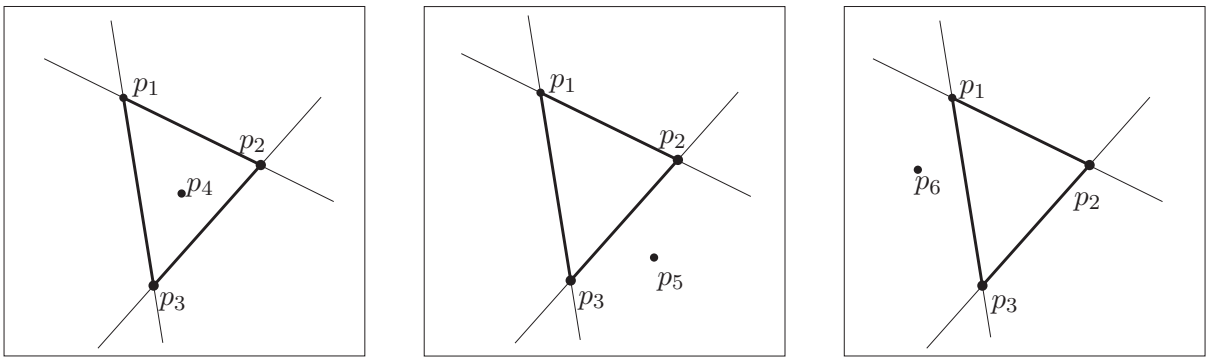

Figura 2.8: Situación de p_4 p_5 y p_6 en el mismo entorno creado por 3 puntos p_1, p_2, p_3.

Para cada entorno calcularemos su hyperline sequences:

1	2	-3	-4
2	1	4	3
3	1	-2	-4
4	1	-2	3

Cuadro 2.2: Hyperline sequences correspondientes a la situación del punto p_4.

1	2	-3	-5
2	1	3	5
3	1	-5	-2
5	1	3	-2

Cuadro 2.3: Hyperline sequences correspondientes a la situación del punto p_5.

1	2	-6	-3
2	1	6	3
3	1	6	-2
6	1	-3	-2

Cuadro 2.4: Hyperline sequences correspondientes a la situación del punto p_6.

Gráficamente, se ve perfectamente que pared puede ser obstáculo y que pared no, pero el objetivo es conocer esa información utilizando la información de las hyperline sequences. Para ello, sólo nos hace falta la hyperline sequence correspondiente a los puntos que representan al robot.

4	1	-2	3

5	1	3	-2

6	1	-3	-2

Cuadro 2.5: Hyperline sequences de los puntos p_4, p_5, p_6

Hay que tratar con especial cuidado toda la información que trate con el punto p_1, por lo que se tratarán dos casos distintos: si la pared que se quiere atravesar contiene al punto p_1 o si no contiene al punto p_1.

Caso 1: Si la pared que quieres atravesar contiene al punto p_1.

En este caso trataremos al punto que representa al robot como p_k, y al punto que contiene la recta le llamaremos p_i.

Para comprobar si una pared $w_{1,i}$ tiene posibilidad de ser obstáculo para el punto p_k, comprobaremos la posición y el signo del punto p_i, dentro de la hyperline sequence hs_k.

Para que la pared pueda ser obstáculo, el punto p_i debe cumplir alguna de estas dos condiciones:

- estar en la tercera posición de hs_k (la primera y la segunda posición siempre estarán ocupadas por el punto que represente al robot y por p_1) y ser negativo
- estar en la última posición de la hs_k y ser positivo.

En cualquier otro caso, la pared nunca sería obstáculo.

En las tres situaciones de la Figura 2.8, la pared $w_{1,2}$, sería obstáculo para el punto p_4, y no lo sería ni para p_5 ni p_6. Comprobaremos como se cumplen las condiciones anteriores en las hyperline sequences de cada punto que se muestran en el Cuadro 2.5.

En el caso de la pared $w_{1,3}$, sería obstáculo para p_4 y p_5, y no sería obstáculo para p_6.

Caso 2: Si la pared que queremos atravesar no contiene al punto p_1.

Si se está tratando la pared $w_{i,j}$, comprobaremos el signo de p_i, p_j dentro de la ks_k. Si ambos han sido almacenado con la misma orientación, no es obstáculo, en cambio si tienen diferente orientación, si será obstáculo.

En los ejemplo de la Figura 2.8, vemos que la pared $w_{2,3}$, podría sería obstáculo para el punto p_4 y p_6.

Para el punto p_5, la pared $w_{2,3}$ nunca sería obstáculo.

2.6. Algoritmo

El algoritmo A* es el algoritmo usado en el problema. Este es un algoritmo de búsqueda en grafos que encuentra siempre el camino de menor coste entre un nodo origen y un nodo objetivo, si cumple unas determinadas condiciones. El mayor problema de este algoritmo, es el espacio requerido, dado que tiene que almacenar todos los posibles siguientes nodos de cada estado. La cantidad de memoria requerida será exponencial con respecto al tamaño del problema.

El algoritmo nos garantiza que si existe un camino posible entre la configuración de origen y la de destino, lo va a encontrar, si existen varios caminos, nos va a devolver el óptimo, y devolverá fallo si no existe

El algoritmo se basa en construir un árbol de búsqueda, en el que se van explorando los sucesores de cada nodo, eligiendo los sucesores que vamos a explorar, en función del conocimiento que tenemos del problema. El objetivo final es construir una lista de nodos que recorriéndola nos lleve desde el nodo inicial hasta el nodo final con el menor coste (camino óptimo). Por lo tanto, debemos definir una función que aplicada a cada nodo nos diga la posibilidad de que dicho nodo forme parte del camino óptimo. Esta función es la suma de dos factores:

$f = g + h$.

2.6.1. Función h

La función h se llama función heurística, y nos proporcionará el conocimiento del problema. Cabe señalar que cuando calculamos el valor de h, ignoramos cualquier obstáculo que intervenga. Es una estimación de la distancia que queda, es por eso que se llama heurística.

Esta función debe ser admisible, es decir, debe cumplir la siguiente condición:

$$Distancia\ real(Nodo\ actual, Nodo\ destino) \geq h(Nodo\ actual, Nodo\ destino)$$

Además, la función heurística debe ser consistente, por lo que debe cumplir que:

Siendo x el nodo actual, y siendo z un nodo sucesor de x, se cumple que:

$$h^*(x) - h^*(z) \leq d(x, z),$$

siendo d(x,z) el valor de la función g, de ir del nodo x al nodo z.

En nuestro caso, es consistente, ya que el valor de g entre nodos adyacentes, es siempre 1, al igual que el valor de la función h, cada vez que se avanza a una celda más cercana al destino, se disminuye en 1.

El valor de h, lo calcularemos comparando las hyperline sequences de la posición actual del robot, con la hyperline sequences de la posición final del robot. Se calcularán el número de posiciones que se ha movido el punto que representa al robot, dentro de las $N - 1$ primeras hyperline. Dependiendo de si los signos del punto que representa al robot, en ambas hyperline sequence coinciden, se calculará de una manera o de otra.

En el ejemplo de la Figura 2.9 se muestra el valor de la función h en distintas celdas del espacio de búsqueda.

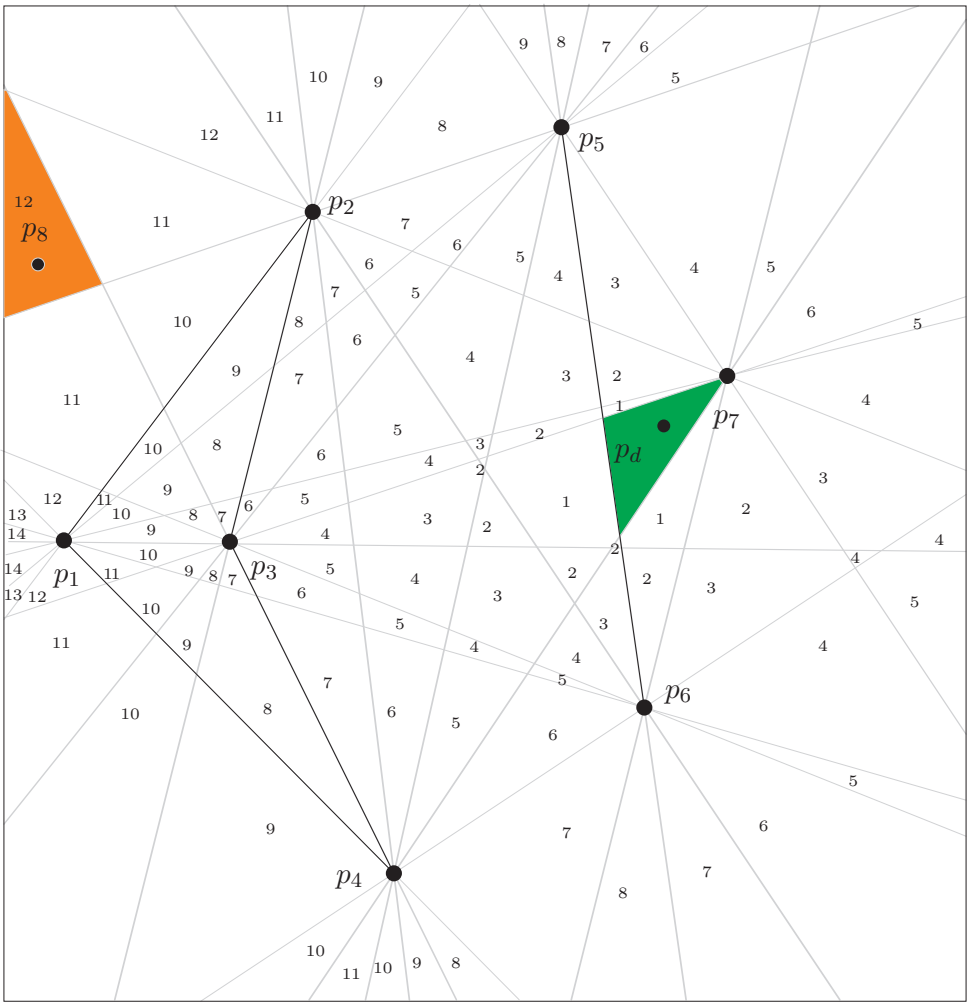

Figura 2.9: Valores de la función h en distintas celdas del espacio de búsqueda.

2.6.2. Función g

El valor de la función g será el número de transiciones de paredes que lleva el robot. Independientemente de la recta que se atraviese, todos los movimientos tendrán el mismo valor. Esta función, nos permitirá saber cuántas transiciones ha tenido que realizar el robot para llegar al destino.

2.6.3. Complejidad del algoritmo A*

La complejidad computacional

La complejidad computacional del algoritmo está relacionada con la calidad de la heurística que se utilice en el problema.

En el peor caso, el número de nodos expandidos es exponencial a la longitud de la solución, pero es polinomial cuando utilizamos una función heurísitca óptima, como es nuestro caso, ya que calcula la distancia exacta de llegar al nodo destino, sin tener en cuenta los obstáculos.

Complejidad en memoria

El espacio requerido por A* para ser ejecutado es el gran problema del Algoritmo A*, ya que tiene que almacenar todos los posibles siguientes nodos de cada estado. La cantidad de memoria que requerirá será exponencial con respecto al tamaño del problema.

Capítulo 3

Descripción informática

3.1. Análisis

Tenemos varios ficheros de entrada, todos ellos contienen una serie de números en la misma línea, y cada uno de ellos seguido de un espacio en blanco. Cada par de números crea un punto con la coordenada X y la coordenada Y. La información que contiene cada fichero es la siguiente:

- Inicio: Contendrá las coordenadas de todos los puntos de los robots móviles.
- Final: Contendrá las coordenadas de las posiciones finales donde deberá llegar los robots móviles.
- Fijos: Contendrá las coordenadas de todos los landmarks y los vértices de los obstáculos.

Otro fichero de entrada será el fichero Obstáculos, el cuál contendrá una cantidad par de números enteros, los cuáles serán los extremos de las aristas que forman algún obstáculo. Ningún número de este fichero deberá ser mayor a la cantidad de puntos que contenga nuestra configuración de puntos.

Salida: Por salida estándar mostrará la secuencia de aristas que ha atravesado cada robot, para conseguir el camino mínimo. También obtendremos el número de movimientos que ha necesitado para conseguirlo y el tiempo que necesita cada robot, para encontrar el camino óptimo, o para devolver fallo en caso de que no haya ningún camino posible.

La misma salida que vemos en la salida estándar, también quedará guardada en un fichero de texto llamado “solucion.txt”.

El lenguaje de programación escogido para la implementación del algoritmo es C, con el entorno de desarrollo que nos proporciona Eclipse.

Se han creado 4 varios archivos con extensión “.c”para facilitar la organización de las tareas a realizar. Todos ellos tendrán su correspondiente archivo de cabecera con extensión “.h”, en los que definimos los parámetros y funciones que utilizaremos.

- TipoPunto.c : Se ha creado para obtener información de los puntos, y poder realizar cualquier tipo de información con ellos.
- TipoLista.c : Se ha creado con la finalidad de poder calcular la hyperline sequences de una configuración de puntos. Para ello, haremos uso de TipoPunto.
- mutaciones.c : Se ha creado con la finalidad de poder obtener el conjunto de mutaciones, dad una hyperline sequences. Además de para caracterizar las paredes de las celdas.
- datos.c : Ha sido creado para poder llevar a cabo el algoritmo A*. Este hará uso de los tres anteriores.

En nuestro programa se puede planificar el camino de n robots, pero el algoritmo solo calcula el camino de uno en uno, sin tener en cuenta al resto de robots. Una vez se encuentre el camino para un robot, se liberara la memoria reservada para dicha planificación.

Para la representación de puntos, nos hemos ayudado de la aplicación Cinderella.

3.2. Matroide orientada e hyperline sequences

3.2.1. Análisis

En este apartado explicaremos como calculamos la hyperline sequences en nuestro programa. Primero saber que llamaremos punto *inicial* al punto del que queremos calcular la hyperline sequence, y llamaremos punto $final$ al punto con el que unimos la recta, que giramos para obtener la hyperline sequence.

Vamos a calcular la hyperline sequence de cada punto por separado, cambiando simplemente cual será el punto *inicial* y el punto $final$. Para todos los puntos, el punto inicial, será el propio punto del que se quiere obtener la hyperline sequence , mientras el punto final, en el caso del punto p_1 es p_2, en cualquier otro caso siempre es p_1.

```
PARA cada punto i DESDE 1 HASTA N HACER
    hallar_hyperline()
FIN_PARA
```

Una vez tengamos el punto *inicial* y el punto $final$, realizaremos un cambio de base, es decir, se creará un nuevo sistema de coordenadas, en el que el centro de dicho sistema será el punto para el que se quiere calcular la hyperline sequence, y el eje de abscisas será la línea que una dicho punto, con el punto final.

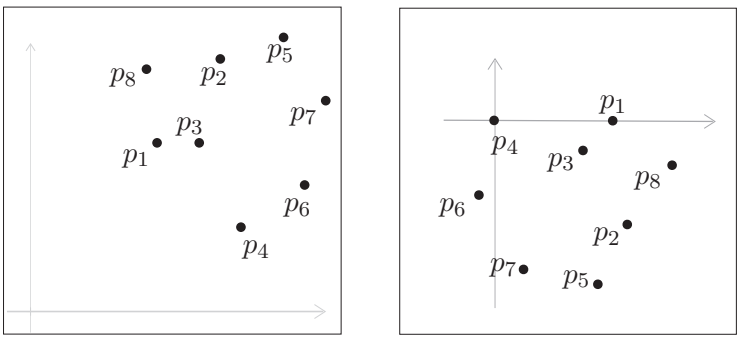

Figura 3.1: Ejes de coordenadas antes y después del cambio del centro de coordenadas, cuando se quiere calcular la hyperline sequence del punto p_4.

Una vez tengamos el nuevo sistema de coordenadas, habrá que incluir al resto de puntos en el nuevo sistema, sumándole a todos los puntos las coordenadas del punto *inicial*. Cuando estén todos los puntos en su posición dentro del nuevo sistema de coordenadas, calcularemos el ángulo que forman con respecto al eje de abscisas y el centro del sistema de coordenadas.

Todos los puntos que estén situados en el tercer o cuarto cuadrante, los pasaremos al primer o segundo cuadrante, sumándoles 180 grados. A todos estos, se les cambiará el nombre, y se los pondrá en negativo. Una vez realizado esto, ya solo hay que ordenarlos de menor a mayor por el nuevo ángulo.

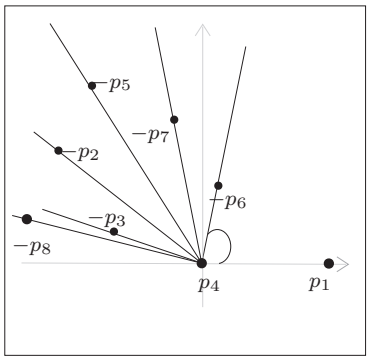

Figura 3.2: Posición de los puntos y sus ángulos, en el nuevo sistema de coordenadas con centro de ordenadas en p_4.

La hyperline sequence correspondiente al punto p_4 será la siguiente:

4	1	-6	-7	-5	-2	-3	**-8**

Una vez hayamos calculado la hyperline sequence de todos los puntos, la hyperlines sequences será el conjunto de hyperline sequence de todos los puntos.

3.2.2. Pseudocódigo

En nuestro programa, primero hemos calculado la hyperline sequence del punto p_1, y luego el resto, ya que tendrán un punto $final$ distinto

```
void HS(lista, sol[MFIL], nfil){
    aux=copia(lista);
    inicio->(aux,1);
    final->(aux,2);
    Cambio_base(aux,1);
    Modifica_ángulos(aux,1);
    cambio_cuadrantes(aux);
    Ordenar_lista_por_ángulos(aux1,1,sol);

    FOR i=2 hasta N HACER
        aux=copia(lista);
        inicio->(aux,i);
        final->(aux,1);
        Cambio_base(aux,i);
        Modifica_ángulos(aux,i);
        cambio_cuadrantes(aux);
        Ordenar_lista_por_ángulos(aux,1,sol);
    }
}
```

3.3. Lista de mutaciones

Para hallar las paredes de la celda donde está situado el punto que representa al robot, utilizaremos el siguiente criterio.

Dada una hyperline sequences, si tenemos tres índices i, j, k, podremos decir que forman una mutación si se cumplen dos de las siguientes condiciones:

1) si en la hyperline i, los índices j y k son adyacentes.

2) si en la hyperline j, los índices i y k son adyacentes.

3) si en la hyperline k, los índices j y i son adyacentes.

Este algoritmo saca todas las posibles mutaciones de las hyperline sequences, para todos los puntos. Una vez obtenido toda la lista, descartaremos todas aquellas en las que no aparezca el punto que representa al robot, en este caso p_8.

Hyperline sequences							
1	2	8	-4	-6	-3	-7	-5
2	1	3	4	6	7	-8	5
3	1	-7	-5	-2	4	-8	6
4	1	-6	-7	-5	-2	-3	-8
5	1	3	4	6	7	-8	-2
6	1	4	-7	-5	-2	-8	-3
7	1	3	4	6	-5	-2	-8
8	1	4	3	6	7	2	5

Cuadro 3.1: Hyperline sequences correspondiente a la Fig. 3.3, del cuál sacaremos todas las posibles mutaciones.

Lista de Mutaciones		
1	2	5
1	3	6
1	3	7
1	4	6
1	4	8
2	3	4
2	5	8
2	7	8
3	4	8
3	6	8
4	6	7
5	6	7

Lista de Mutaciones		
1	4	8
2	5	8
2	7	8
3	4	8
3	6	8

Cuadro 3.2: A la izquierda, lista de todas las posibles mutaciones que obtenemos de la hyperline sequences del cuadro Fig. 3.1. A la derecha, las mutaciones en las que participa el robot representado por p_8.

3.4. Algoritmo A*

Para realizar la planificación del camino, utilizaremos el algoritmo A*.

Durante el algoritmo hemos usado nodos, que estarán compuestos por la hyperline sequences, el historial de rectas atravesadas hasta llegar a la posición actual, el valor de las funciones f, g y h, y por el nombre del punto que le representa.

Este algoritmo, tiene dos estructuras auxiliares: lista cerrada y lista abierta.

La `lista abierta` contendrá todos los posibles nodos que se pueden expandir. La principal característica de esta estructura, es que todos los nodos que contenga, estarán ordenados de menor a mayor valor de la función f. Esto hará, que siempre se expanda el mejor nodo de todos los posibles nodos que pueden llegar a ser solución.

Cuando se vaya a sacar un elemento de la lista abierta, tendrá prioridad el que tenga menor valor de la función f, pero en caso de que haya varios nodos con el mismo valor, se tendrá en cuenta el que tenga menor valor de la función g. En caso de que siga habiendo varios nodos con ambos valores igual, el nodo a sacar será el que haya sido introducido antes en la lista abierta.

La `lista cerrada` contendrá a todos los nodos que ya han sido visitados. Estos nodos, ya no podrán volver a ser tratados.

En nuestro programa, una vez hayamos leído los ficheros de entrada, tendremos dos listas de nodos: la lista de nodos de los robots, y la lista de nodos de los destinos. Una vez se llame al algoritmo A*, se realizará la planificación de un sólo robot, llegándole el nodo perteneciente a la posición inicial del robot y el nodo correspondiente a la posición final del robot.

El algoritmo es una combinación entre búsquedas en anchura con profundidad: mientras que h'(n) tiende primero en profundidad, g(n) tiende primero en anchura. De este modo, se cambia de camino de búsqueda cada vez que existen nodos más prometedores. Al final de la búsqueda de cada robot, se liberará la memoria reservada para dicha búsqueda, vaciando tanto la lista abierta, como la lista cerrada, en caso de tener algún nodo.

3.4.1. Cálculo del camino

Para realizar la búsqueda, se deberán seguir estos pasos, con su correspondiente orden: Considerar que el nodo inicial, tendrá la hyperline sequences de la celda inicial, así como los

valores de las funciones f, g y h.

Sumario del método A*

1) Añade el nodo inicial a la `lista abierta`.

2) Repite lo siguiente:

   a) Busca el nodo con el coste f más bajo en la `lista abierta`. Nos referimos a este como el nodo actual.

   b) Cámbialo a la `lista cerrada`.

   c) Para cada uno de los nodos adyacentes al nodo actual:

      - Si no es posible acceder porque les separa un obstáculo o si está en la `lista cerrada`, ignóralo. En cualquier otro caso haz lo siguiente.
      - Si no está en la `lista abierta`, añádelo a la `lista abierta`. Haz que el historial de esta sea el historial del nodo actual, más la arista que acaba de atravesar. Almacena los nuevos valores de las funciones f, g y h del nodo.
      - Si ya está en la `lista abierta`, comprueba si el camino para ese es mejor usando el coste g como baremo. Un coste g menor significa que este es un mejor camino. Si es así, cambia el historial de las aristas atravesadas por las de éste nuevo nodo y recalcula g y f del nodo. Se volverá a ordenar la lista de menor a mayor f.

   d) Para cuando:

      - añadas el nodo destino a la lista abierta, el camino ha sido encontrado.
      - la lista abierta se quede vacía. En este caso no hay camino.

3) Guarda el camino. El camino será la secuencia de aristas atravesadas que guarda en el historial el nodo que ha conseguido llegar al destino.

En el punto 2.c, todos los nodos adyacentes, tendrán un orden de prioridad para ser expandidos. Esta prioridad lo marca la recta que debe atravesar cada nodo, teniendo prioridad en ser tratado el nodo que atraviese la recta que tenga una etiqueta menor.

3.4.2. Pseudocódigo

```
void  ALGORITMO ( inicio:Nodo, final: Nodo ) {
    insertamos (inicio,listaAbierta);
    while (listaAbiertos!=NULL) {
        nodoactual=sacar(listaAbierta);
        insertamos(nodoactual,listaCerrados)
        if esSolucion(origen) {
            imprime historialDeTodoLosNodos;
        else {
            listaMutaciones(listamutaciones,nodo);
            while (listamutaciones != vacia) {
                sacarMutacion(mutacion,listamutaciones);
                if NoesObstaculo (mutacion){
                    funcion g ++;
                    cambiarMatroide(nodoActual->matroide);
                    CalcularNuevoH(Nodoactual);
                    actualizarF(Nodoactual);
                    if NoEsta(listaCerrada,Nodoactual){
                        if NoEsta(listaAbierta,Nodoactual){
                            insertamos (Nodoactual,listaAbierta);
                        }
                        else {
                            if (NuevoValorDeF<antiguoValorF)
                                Actualizar(listaAbierta,Nodoactual);
                        }/*else*/
                    }/*if*/
                }/*if*/
            } /*while*/
        }/*else*/
    }/*while*/
}
```

3.4.3. Desarrollo

En este apartado, vamos a ver como se desarrollaría la búsqueda en el entorno que se muestra en la Figura 3.3, en el que el robot es representado por el punto p_8 que está situado en la celda de color verde, y debe llegar a su destino representado por el punto pg_1, situado en la celda de color rojo.

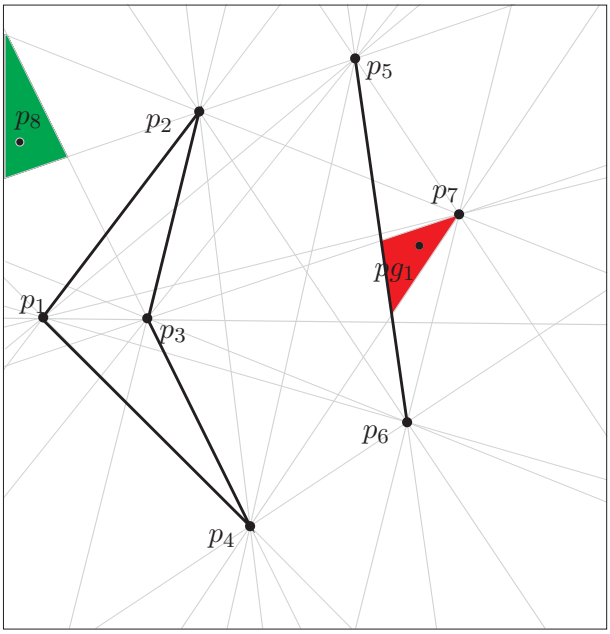

Figura 3.3: Situación inicial representada por una celda de color verde, y situación final del robot representada por una celda de color roja en el espacio de búsqueda.

El nodo inicial, estará formado por las hyperline sequences correspondiente a la configuración actual de todos los puntos. El historial de rectas atravesadas estará vacía. Los valores de la función h, se calcula comparándola con la hyperline sequences del destino. El valor de la función g, empezaría desde 0.

El nodo destino estará formado por las hyperlines sequences correspondientes a la configuración de puntos, en la que está contenido pg_1. El resto de valores del nodo, estarán inicializados a 0, ya que no tendrán ninguna importancia.

1.- Empieza con el nodo inicial y se añade a una `lista abierta` de nodos a tener en cuenta. Ahora mismo solo contiene un nodo en la lista, pero más tarde contendrá más. La `lista abierta` contiene los nodos que podrían formar parte del camino que se busca, pero que quizás no lo hagan. Esta es una lista de las nodos que necesitan ser comprobados para llegar a un camino óptimo.

2.- Se saca el nodo inicial de la `lista abierta` y se añade a una `lista cerrada` de nodos que no necesitan ser mirados de nuevo. A partir de ahora se le llamará nodo actual.

3.- Se determina el conjunto de nodos adyacentes al nodo actual, ignorando los nodos que estén separados por un obstáculo. En los nodos adyacentes, se actualiza las hyperline sequences, el valor de las funciones f, g y h. Guarda la arista que acaba de ser atravesada en su propio historial. Se añaden a la `lista abierta` cada uno de esos nodos.

El orden de prioridad para tratar a los nodos adyacentes del nodo actual, dependerá de la arista que tienen que atravesar. Tendrá prioridad el que atraviese la arista de etiquetado menor. En este ejemplo primero se tratará el nodo que atraviesa la recta $l_{1,4}$, luego el nodo que atraviesa la recta $l_{2,5}$, luego $l_{2,7}$ y luego $l_{3,4}$.

Esto es importante, ya que un nodo en caso de tener el mismo valor de las funciones f y g, tendrá preferencia si ha sido ingresado antes en la lista abierta.

El resultado de los tres primeros pasos en nuestra búsqueda puede verse en la Figura 3.4, en que los nodos que contienen a las celdas de color amarillo, han sido añadidas a la `lista abierta`, y el nodo que contiene a la celda de color rojo, es un nodo que ya ha sido tratado, por lo que estará en `lista cerrada`.

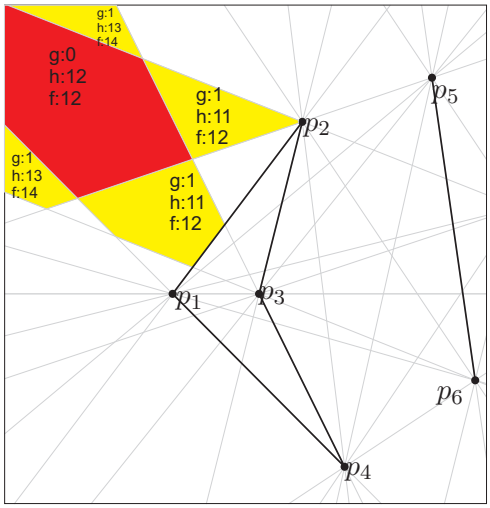

Figura 3.4: Conjunto de nodos de lista abierta coloreados de amarillo, y conjunto de nodos de la lista cerrada coloreados de rojo.

Para continuar la búsqueda, simplemente elegimos el nodo con el valor de f más bajo, de todos los nodos que estén en la `lista abierta`. En caso de que haya varios nodos con el mismo valor de h, elegiremos al nodo que tenga menor valor de g. Si aún así, hay diferentes nodos que tienen el mismo valor de ambas funciones, elegiremos el nodo que haya sido ingresado antes en la `lista abierta`. Después hacemos lo siguiente con el nodo seleccionado:

4) Se saca el nodo de la lista abierta y se añade a la lista cerrada.

5) Se comprueba todos los nodos adyacentes, ignorando aquellos que estén en la `lista cerrada` o aquellos que sean inalcanzables debido a que les separa un obstáculo. Si no están en la lista abierta, se añaden una vez hayamos actualizado el valor de las funciones f,g y h, y se haya guardado en el historial de dicho nodo la arista atravesada.

6) Si el nodo adyacente ya está en la `lista abierta`, comprueba que el valor de la función g de ese nodo sea más bajo que la del que estamos usando para ir allí. Si no es así, no hagas nada. Por otro lado, si el coste g del nuevo camino es más bajo, cambia el historial de las aristas atravesada al nodo seleccionado. Finalmente, recalcula el valor de f y de g de esa celda, y se vuelve a ordenar la `lista abierta` de menor a mayor valor de f.

7) Los pasos 4,5 y 6 se repetirán hasta que:

   - se encuentre un nodo, que tenga valor de la función h igual a 0. En este caso el conjunto de rectas atravesadas estará guardado en el historial del nodo
   - la `lista abierta` se haya quedado vacía, lo cual indica, que no se ha podido encontrar una solución.

Una vez que se encuentre la solución, se tendrá que liberar el espacio en memoria reservado para la búsqueda. Se eliminarán todos los nodos almacenados en la lista abierta y en la lista cerrada. En este caso, no hay más robots que vayan a realizar la planificación en este entorno, pero si lo hubiese, ya se encontraría ambas listas, vacías y preparadas para ser usadas.

Capítulo 4

Resultados

El algoritmo descrito en capítulos anteriores, se ha probado sobre distintos problemas de planificación.

4.1. Planificación del camino de un robot en un entorno poligonal con 7 vértices

En este primer ejemplo, trataremos el problema representado en la Figura 4.1 en el que el robot, desde su posición inicial ps_1 debe llegar a la posición final representada por el punto pg_1, evitando los obstáculos.

El entorno estará formado por un polígono no convexo de cuatro vértices p_1,p_2,p_3,p_4, un segmento con vértices p_5,p_6 y un landmark representado por p_7. La partición del entorno, quedaría como se muestra en la Figura 4.2

punto origen			punto destino		
Nombre_Origen	x	y	Nombre_Destino	x	y
ps_1	550	1150	pg_1	1300	925

Cuadro 4.1: Coordenadas del punto origen ps_1 y del punto final pg_1 en el entorno de la Figura 4.1.

Coordenadas		
Vértice	x	y
p_1	600	800
p_2	900	1200
p_3	800	800
p_4	1000	400
p_5	1200	1300
p_6	1300	600
p_7	1400	1000

Cuadro 4.2: Coordenadas x-y de los vértices de los obstáculos y de los landmarks del problema descrito en la Sección 4.1.

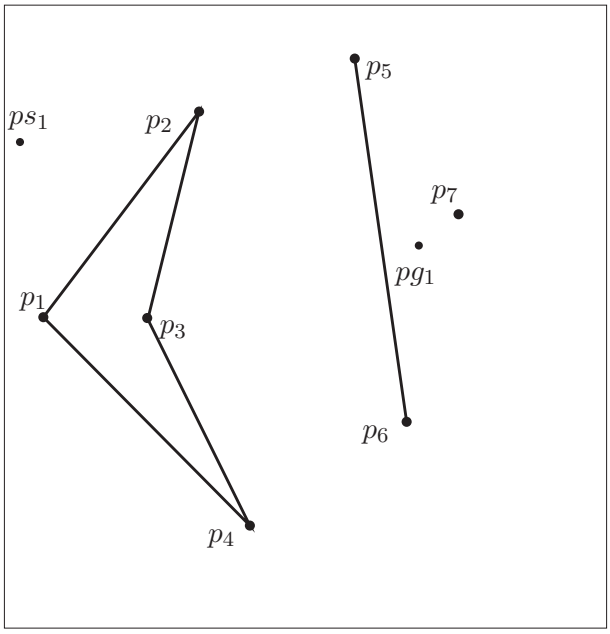

Figura 4.1: Planificación del camino de un robot cuya posición inicial es representada por el punto ps_1 en un entorno poligonal con 7 vértices descrito la Sección 4.1.

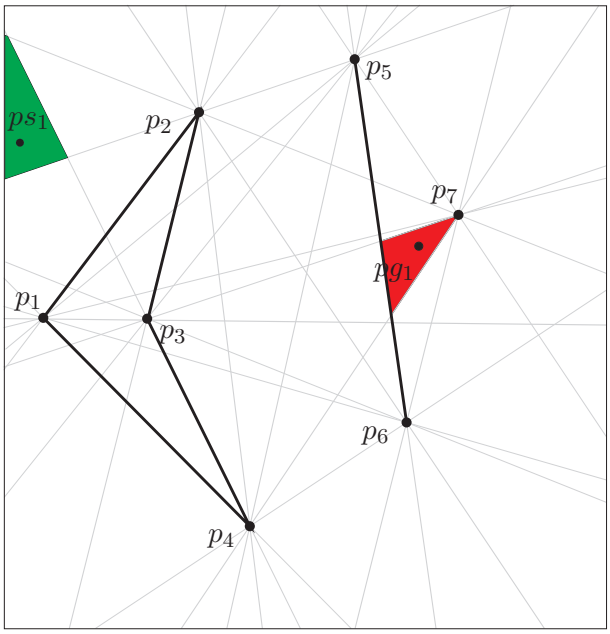

Figura 4.2: Partición del entorno en celdas en el problema descrito en la Sección 4.1.

En este ejemplo, el robot ha encontrado un camino mínimo formado por 17 celdas (incluidas la inicial y la final) y en el que hay que atravesar un total de 16 rectas que se muestran en el cuadro 4.3. El camino más corto, se muestra en la Figura 4.3 con una línea de color amarillo, que pasa por un punto aleatorio de todas las celdas que forman el camino mínimo. Estas celdas están de color gris claro y oscuro, diferenciándolas para saber exactamente de qué celda a que celda avanza.

Para encontrar el camino mínimo, el robot ha tenido que explorar un total de 68 celdas.

El tiempo de ejecución de está búsqueda es de 0.046 segundos, ejecutándolo en un portátil ASUS con procesador Intel Core i.5, M430 a 2.27 GHz, con 4GB de RAM.

Secuencia de las rectas atravesadas para conseguir el camino mínimo																
nº transición	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
recta	$l_{3,4}$	$l_{2,7}$	$l_{2,6}$	$l_{2,4}$	$l_{2,3}$	$l_{1,2}$	$l_{5,7}$	$l_{5,6}$	$l_{4,5}$	$l_{3,5}$	$l_{1,5}$	$l_{2,5}$	$l_{5,7}$	$l_{2,7}$	$l_{1,7}$	$l_{3,7}$

Cuadro 4.3: Secuencia de rectas atravesadas por el robot representado por el punto ps_1, para encontrar el destino pg_1 en el problema descrito en la Sección 4.1.

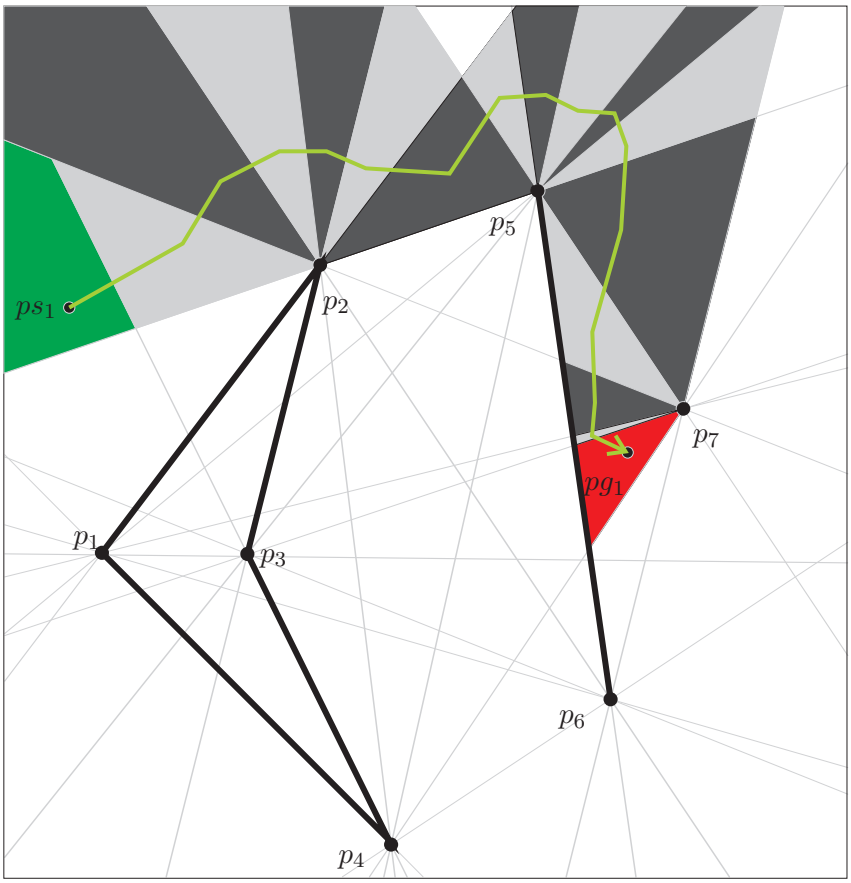

Figura 4.3: Camino mínimo para un robot situado en una posición inicial ps_1, y cuyo destino es representado por pg_1, en el ejemplo de la Sección 4.1.

4.2. Planificación del camino para 3 robots en un entorno poligonal formado por 2 obstáculos convexos.

El entorno de este segundo ejemplo estará formado por dos obstáculos poligonales convexos, uno con vértices p_1, p_2, p_3 y un segundo obstáculo con vértices p_4, p_5, p_6, p_7. Hay tres robots que desde las 3 posiciones iniciales representados por los puntos ps_1, ps_2, ps_3, deben llegar a sus destinos correspondientes representados por los puntos pg_1, pg_2, pg_3.

Las coordenadas de los vértices de los obstáculos se muestran en el Cuadro 4.4, y las coordenadas de la posición inicial y final del robot, se muestran en el Cuadro 4.5.

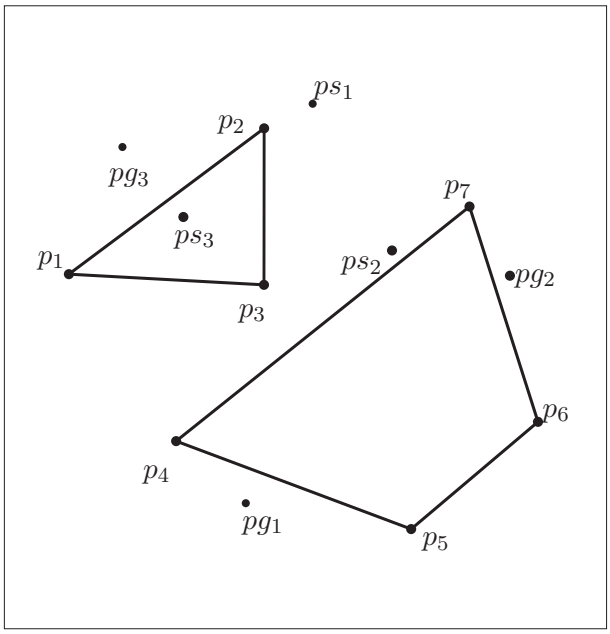

Figura 4.4: Planificación del camino para 3 robots en un entorno poligonal formado por 2 obstáculos convexos del ejemplo de la Sección 4.2.

Coordenadas		
Vértice	x	y
p_1	250	1000
p_2	450	1150
p_3	450	990
p_4	360	830
p_5	600	740
p_6	730	850
p_7	660	1070

Cuadro 4.4: Coordenadas de los vértices de los obstáculos del ejemplo descrito en la Sección 4.2

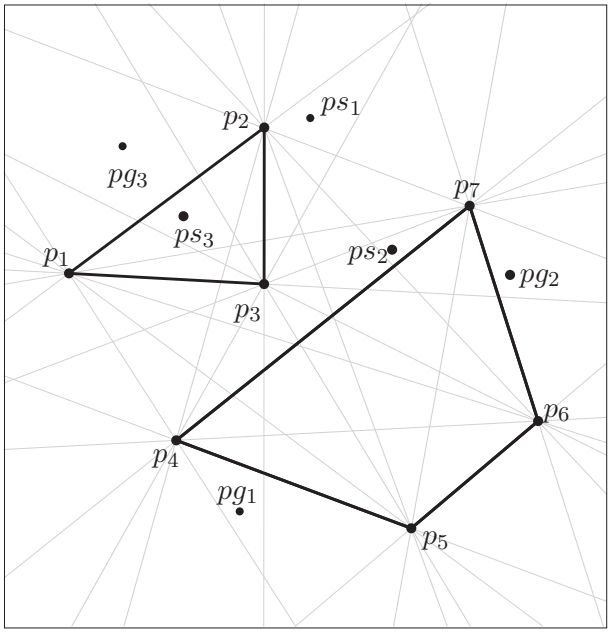

Figura 4.5: Partición del entorno en celdas en el problema descrito en la Sección 4.2.

puntos origen			puntos destino		
Nombre_Origen	x	y	Nombre_Destino	x	y
ps_1	490	1160	pg_1	430	750
ps_2	580	1025	pg_2	700	1000
ps_3	370	1050	pg_3	300	1100

Cuadro 4.5: Coordenadas de los puntos origen y final en el entorno de la Figura 4.4

Los tres robots, iniciarán la búsqueda por orden, sin tener en cuenta la posición del resto de robots.

El primer robot en empezar la búsqueda será ps_1, situado en una celda de color verde, el cual debe llegar a la celda de color rojo, donde está situado el punto pg_1.

El camino mínimo para ps_1 estará formado por un total de 22 celdas (incluidas la celda inicial y final). Para realizarlo hay que atravesar 21 rectas, mostradas en la tabla 4.6.

El camino está marcado con una línea de color negro, que pasa por todas las celdas que forman el camino mínimo. Estas celdas serán de color azul claro y azul oscuro, diferenciandolas para saber exactamente de qué celda a que celda avanza el robot. Dentro de cada celda que forma parte del camino mínimo, se ha representado un punto arbitrario que será el punto por el que pasará la recta que marca el camino.

Una vez el robot ps_1 haya encontrado su camino, el robot ps_2 iniciará su búsqueda. Esté estará situado en una celda de color verde, y deberá llegar a su destino pg_2 situado en una celda de color rojo. El camino encontrado estará formado por 10 celdas (incluidas la inicial y final), de color rosa en diferentes tonalidades (diferenciandolas para saber exactamente de que celda a que celda avanza el robot), y marcado con una línea de color negro. La secuencia de 9 rectas atravesadas por el robot, se muestra en la tabla 4.7.

El robot ps_3 iniciará la búsqueda hacia su destino pg_3. El robot está situado situadas dentro del obstáculo formado por los vértices (p_1, p_2, p_3) ,mientras que el punto destino está situado fuera del obstáculo, por lo que el robot nunca podrá encontrar un camino hasta él.

El robot, una vez haya visitado el total de 12 celdas que hay dentro del obstáculo (celdas de color amarillo en la Figura 4.6), la lista abierta de nuestro algoritmo quedará vacía, por lo que

el algoritmo devolverá una alerta diciendo que no es posible encontrar un camino para el robot ps_3

El tiempo de búsqueda para cada robot, y el número de celdas exploradas es el siguiente:

-El robot 1 tarda 0.109000 segundos y ha necesitado explorar 152 celdas para encontrar el camino mínimo.

-El robot 2 tarda 0.015000 segundos y ha necesitado explorar 22 celdas para encontrar el camino mínimo.

-El robot 3 tarda 0.014000 segundos en explorar las 12 celdas, hasta ver que no encuentra ningún camino.

Estos tiempos de búsqueda, han sido obtenidos al ejecutarlo en un ordenador portátil ASUS con procesador Intel Core i.5, M430 a 2.27 GHz, con 4GB de RAM.

Conjunto de rectas atravesadas por ps_1											
nº transición	1	2	3	4	5	6	7	8	9	10	11
recta	$l_{2,7}$	$l_{1,7}$	$l_{2,6}$	$l_{2,5}$	$l_{3,4}$	$l_{3,7}$	$l_{1,3}$	$l_{3,6}$	$l_{1,6}$	$l_{3,5}$	$l_{1,5}$
nº transición	12	13	14	15	16	17	18	19	20	21	
recta	$l_{2,3}$	$l_{3,4}$	$l_{2,4}$	$l_{1,4}$	$l_{4,5}$	$l_{4,6}$	$l_{4,7}$	$l_{3,4}$	$l_{2,4}$	$l_{1,4}$	

Cuadro 4.6: Secuencia de rectas atravesadas por el robot representado por el punto ps_1, en el problema descrito en la Sección 4.2.

Conjunto de rectas atravesadas por ps_2									
nºtransición	1	2	3	4	5	6	7	8	9
recta	$l_{3,7}$	$l_{1,7}$	$l_{2,7}$	$l_{6,7}$	$l_{5,7}$	$l_{4,7}$	$l_{3,7}$	$l_{1,7}$	$l_{2,7}$

Cuadro 4.7: Secuencia de rectas atravesadas por el robot representado por el punto ps_2, en el problema descrito en la Sección 4.1.

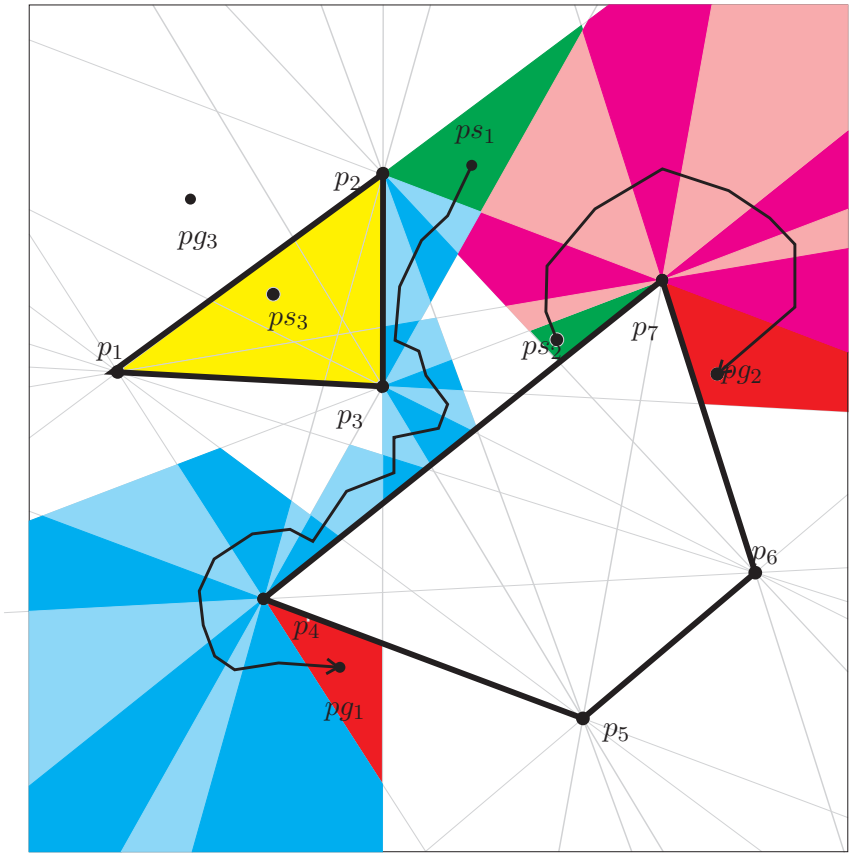

Figura 4.6: Camino mínimo para los 3 robots en el entorno de la Figura 4.4.

4.3. Planificación del camino para un robot en un entorno poligonal formado por 11 vértices.

En este caso, el entorno está formado por 3 curvas poligonales simples, por lo que al no haber ningún obstáculo cerrado, el robot siempre encontrará una solución. La dificultad de este problema, es que estará formado por 11 vértices, una cantidad elevada, que hace que la partición del plano quede con una gran cantidad de celdas.

El entorno es representado en la Figura 4.7, y las coordenadas de los vértices se muestran en el Cuadro 4.8

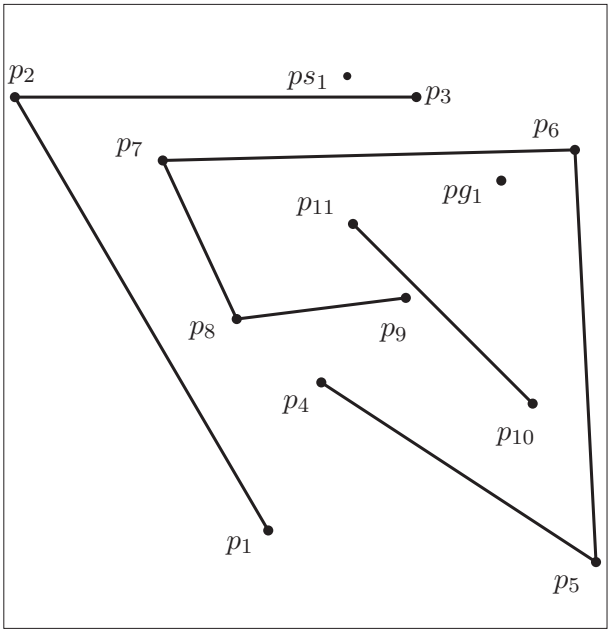

Figura 4.7: Planificación del camino para un robot en un entorno poligonal formado por 11 vértices.

En este ejemplo, solo habrá un robot representado por el punto ps_1. El destino al que debe encontrar, está representado por el punto pg_1. Sus coordenadas se muestran en el Cuadro 4.9

Coordenadas											
vértice	p_1	p_2	p_3	p_4	p_5	p_6	p_7	p_8	p_9	p_{10}	p_{11}
x	125	101	139	130	156	154	115	122	142	150	133
y	17	58	58	31	14	53	52	37	39	29	46

Cuadro 4.8: Coordenadas de los vértices de los obstáculos del ejemplo descrito en la Sección 4.2

punto origen			punto destino		
Nombre_Origen	x	y	Nombre_Destino	x	y
ps_1	137	59	pg_1	147	50

Cuadro 4.9: Coordenadas de los puntos origen y final en el ejemplo de la Figura 4.7

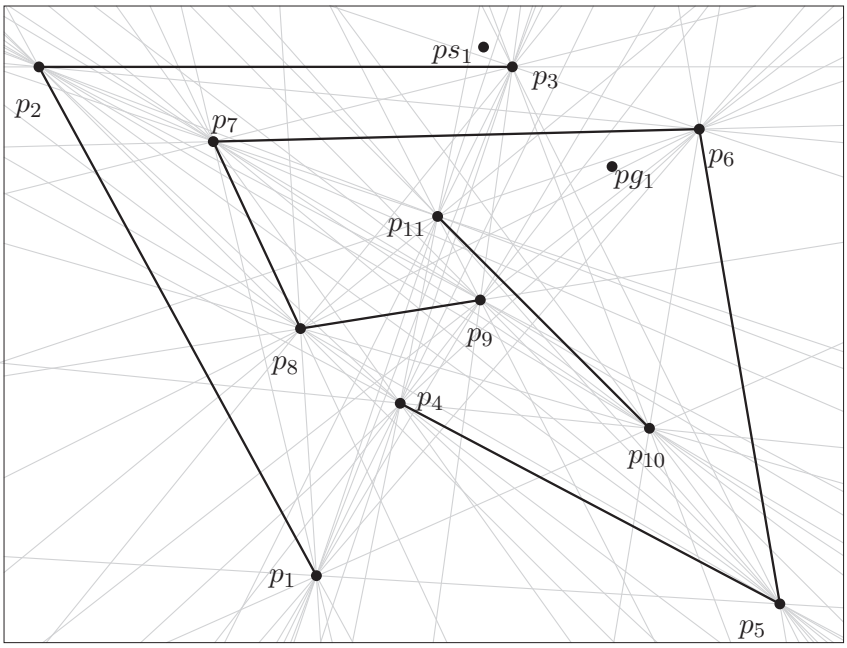

Figura 4.8: Partición del entorno en celdas en el problema descrito en la Sección 4.3.

El camino mínimo lo forman un total de 105 celdas, incluidas la inicial y final.

Para encontrarlo, el robot a tenido que atravesar 104 rectas, cuya secuencia de todas ellas se muestra en el Cuadro 4.10. En la Figura 4.9 se muestra el camino mínimo, marcado con una recta de color rojo.

El robot ha explorado el estado de 4841 celdas, para encontrar la solución óptima. Esta cantidad de celdas, son los nodos que han quedado en la `lista cerrada` en nuestro Algoritmo.

El tiempo de búsqueda del robot ha sido de 85.956 segundos.

Este tiempo de búsqueda, ha sido obtenido al ejecutar el algoritmo en un ordenador portátil ASUS con procesador Intel Core i.5, M430 a 2.27 GHz, con 4GB de RAM.

transición	recta
1	$l_{5,10}$
2	$l_{3,5}$
3	$l_{3,10}$
4	$l_{3,9}$
5	$l_{3,4}$
6	$l_{1,3}$
7	$l_{3,11}$
8	$l_{3,8}$
9	$l_{3,7}$
10	$l_{2,3}$
11	$l_{3,6}$
12	$l_{2,6}$
13	$l_{3,5}$
14	$l_{3,10}$
15	$l_{3,9}$
16	$l_{3,4}$
17	$l_{1,3}$
18	$l_{3,11}$
19	$l_{1,11}$
20	$l_{3,8}$
21	$l_{1,8}$
22	$l_{3,7}$
23	$l_{1,7}$
24	$l_{4,11}$
25	$l_{5,10}$
26	$l_{5,9}$

transición	recta
27	$l_{5,11}$
28	$l_{7,8}$
29	$l_{4,7}$
30	$l_{5,7}$
31	$l_{9,10}$
32	$l_{10,11}$
33	$l_{7,10}$
34	$l_{9,11}$
35	$l_{2,11}$
36	$l_{7,9}$
37	$l_{2,7}$
38	$l_{7,11}$
39	$l_{6,7}$
40	$l_{3,7}$
41	$l_{1,7}$
42	$l_{6,11}$
43	$l_{2,9}$
44	$l_{2,10}$
45	$l_{2,5}$
46	$l_{2,4}$
47	$l_{2,8}$
48	$l_{4,8}$
49	$l_{5,8}$
50	$l_{8,10}$
51	$l_{8,9}$
52	$l_{6,8}$

transición	recta
53	$l_{8,11}$
54	$l_{3,8}$
55	$l_{1,8}$
56	$l_{3,11}$
57	$l_{1,11}$
58	$l_{1,3}$
59	$l_{7,8}$
60	$l_{2,8}$
61	$l_{4,8}$
62	$l_{2,4}$
63	$l_{4,7}$
64	$l_{4,11}$
65	$l_{3,4}$
66	$l_{1,4}$
67	$l_{5,8}$
68	$l_{2,5}$
69	$l_{5,7}$
70	$l_{5,11}$
71	$l_{4,6}$
72	$l_{4,9}$
73	$l_{6,9}$
74	$l_{1,9}$
75	$l_{3,9}$
76	$l_{5,9}$
77	$l_{1,6}$
78	$l_{4,10}$

transición	recta
79	$l_{1,10}$
80	$l_{6,10}$
81	$l_{3,10}$
82	$l_{5,10}$
83	$l_{3,5}$
84	$l_{9,10}$
85	$l_{10,11}$
86	$l_{7,10}$
87	$l_{2,10}$
88	$l_{8,10}$
89	$l_{4,10}$
90	$l_{1,10}$
91	$l_{6,10}$
92	$l_{1,6}$
93	$l_{9,11}$
94	$l_{7,9}$
95	$l_{2,9}$
96	$l_{2,7}$
97	$l_{2,11}$
98	$l_{7,11}$
99	$l_{8,9}$
100	$l_{4,9}$
101	$l_{6,9}$
102	$l_{1,9}$
103	$l_{4,6}$
104	$l_{6,8}$

Cuadro 4.10: Secuencia de las 104 rectas atravesadas por el robot ps_1, en el problema descrito en la Sección 4.3.

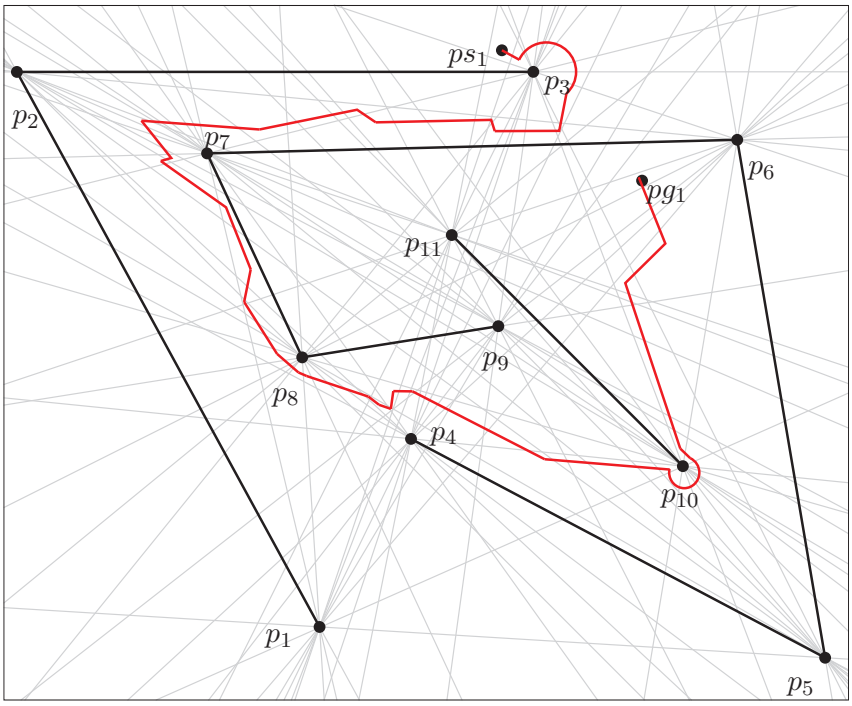

Figura 4.9: Camino mínimo para un robot situado en una posición inicial ps_1, y cuyo destino es representado por pg_1, en el entorno de la Figura 4.7.

Capítulo 5

Conclusiones y posibles trabajos futuros

En este proyecto de fin de carrera se presenta un método para planificar el camino mínimo que debe realizar un robot en un entorno conocido creado por obstáculos poligonales para llegar a su destino, basado en la teoría de matroides orientadas.

Se ha dejado demostrado, que con información cualitativa es posible solucionar el problema, sin ser necesario usar información métrica como las coordenadas.

Un Futuro proyecto podría ser, resolver la planificación cualitativa de caminos para un robot móviles, pero en un espacio en tres dimensiones. El espacio de búsqueda estaría formado por poliedros irregulares.

Otro posible proyecto, sería resolver el mismo problema, pero en vez de robots móviles que se representan con puntos, representarlos con segmentos, o polígonos.

Bibliografía

[1] Almodovar Vialás, N. *Isomorfismo de matroides orientadas aplicado al reconocimiento de objetos a partir de una única vista.* Universidad Rey Juan Carlos, 2008.

[2] La Valle, S. M. *Planning Algorithms.* Cambridge University Press, 2006.

[3] Latombe, J. C. *Robot Motion Planning.* Kluwer Academic Publishers, 1991.

[4] Rosco Carpizo, J. *Isomorfismo de matroides orientadas aplicado al reconocimiento de objetos a partir de una única vista de una escena con múltiples objetos.* Universidad Rey Juan Carlos, 2008.

[5] Tovar, B., Freda, L., and La Valle, S. M. Mapping and navigation from permutations of landmarks. Tech. rep., Department of Computer Science, University of Illinois, 2006.

Apéndice A

Implementación

A.1. TipoPunto

En la estructura tPunto, trataremos la información referida a los puntos. Un punto estará formado por el nombre de dicho punto, el cual será un número, las coordenadas x e y, que serán coordenadas enteras, y un ángulo, que lo utilizaremos para calcular las hyperline sequences. Aquí mostramos las funciones principales del tipo punto(TPunto).

tPunto;
int x;
int y;
int nombre;
double angulo;

Cuadro A.1: Estructura del tipo TPunto.

```
void Actualizar(tPunto *punt, int a, int b, int n);
void DameX(tPunto punt, int *x);
void DameY(tPunto punt, int *y);
void DameNombre(tPunto punt, int *n);
void Dame_angulo(tPunto punt, double *ang);
void actualiza_Angulo(tPunto *punt, double ang);
void Negar_Nombre(tPunto *punt);
double CalcularAngulo(tPunto punt);
void Modifica_X(tPunto *punt, int n);
void Modifica_Y(tPunto *punt, int n);
void modifica_Angulo(tPunto *punt, double ang);
```

Cuadro A.2: Funciones del tipo TPunto.

A.2. TipoLista

En la estructura TLista, trataremos las funciones que realizan alguna operación sobre una lista. Los nodos de la lista, simplemente estarán compuestos por un punto y un puntero que apunta al siguiente nodo de la lista. A continuación vemos todas las funciones y la implementación de las de mayor importancia dentro del problema. Con el TipoLista y TipoPunto, conseguiremos obtener las hyperline sequences.

tLista;
tPunto punto; tLista *sig;

Cuadro A.3: Estructura del tipo TLista.

```
void crearListaVacia(tLista *lista);
void anxLista(tLista *lst, tPunto elem);
void elimLista(tLista *lst, int pos);
tPunto infoLista(tLista lst, int pos);
int longLista(tLista lst);
void modifica_X_lista(tLista *lista, int n);
void modifica_Y_lista(tLista *lista, int n);
void modifica_angulos(tLista *lista, double n, int nombre);
int dame_X(tLista lista, int nombre);
int dame_Y(tLista lista, int nombre);
void copia(tLista list, tLista *copia);
void buscarPunto(tLista lista, int nombre, tPunto *punto);
void cambio_base(tLista *lista, int inicio);
void elimLista_nombre(tLista *lista, int nombre);
void Ordenar_lista_por_angulos(tLista *lista, int n, int *sol[MFIL]);
void cambio_cuadrantes( tLista *lista);
void HS(tLista lista, int *sol[MFIL], int nfil);
void ultimoLista(tLista lista, tPunto *punto);
void leerFijos(tLista *lista, int *longitud);
void leerRobots(tLista *lista, int *longitud ,int fijos);
void leerDestinos(tLista *lista, int *longitud ,int fijos);
void imprime_solucion(int *sol[MFIL], int nfil);
```

Cuadro A.4: Funciones del tipo TLista.

Método que modifica todos los ángulos de la lista de puntos excepto el del primer punto

```
void modifica_angulos(tLista *lista, double n, int nombre){
    int i,longitud,aux;
    tLista temp;
    double ang;
    temp=*lista;
    longitud=longLista(temp);
    for(i=1; i<=longitud; i++){
        DameNombre(temp->punto, &aux);
        if (nombre!=aux){
            modifica_Angulo(&temp->punto, n);
        }
        Dame_angulo(temp->punto,&ang);
        temp=temp->sig;
    }
}
```

Método que cambia a toda la lista los valores de x, y restándoles el valor de dichas coordenadas del punto inicio. Luego recalculamos el ángulo, con los nuevos valores

```
void cambio_base(tLista *lista, int inicio){
    double ang;
    int i,longitud,x1,y1;
    tPunto punto;
    tLista temp,aux;
    aux=*lista;
    x1=dame_X(aux,inicio);
    y1=dame_Y(aux,inicio);
    modifica_X_lista(&aux, x1);
    modifica_Y_lista(&aux, y1);
    longitud=longLista(aux);
    temp=*lista;
```

```
    for(i=1; i<=longitud; i++){
        punto=infoLista(aux, i);
        ang=CalcularAngulo(punto);/*se realiza el cambio de cuadrante*/
        actualiza_Angulo( &temp->punto, ang);
        temp=temp->sig;
    }
}
```

Método que ordena la lista por ángulos de menor a mayor

```
void Ordenar_lista_por_angulos(tLista *lista, int n, int *sol[MFIL]){
    int nombre,i,j,longitud,longitud_modificada;
    tLista aux;
    tPunto punto,minimo;
    double ang_min,ang;
    int var;
    aux=*lista;
    longitud=longLista(aux);
    elimLista_nombre(&aux, n);
    if (n>longitud){
        var=n-longitud;
        *(sol[n-1-var]+0)=n;
    }
    else {
        *(sol[n-1]+0)=n;
    }
        for(j=1; j<longitud; j++){
            longitud_modificada=longLista(aux);
            ang_min=180.0;
            for (i=1; i<=longitud_modificada; i++){
                punto=infoLista(aux, i);
                Dame_angulo(punto, &ang);
                if (ang<ang_min){
                    minimo=punto;
                    ang_min=ang;
```

```
                }
            }
            DameNombre(minimo, &nombre);/*aquí vemos que numero es el punto*/
            elimLista_nombre(&aux, nombre);
            if (n>longitud){
                var=n-longitud;
                *(sol[n-1-var]+j)=nombre;
            }
            else
                *(sol[n-1]+j)=nombre;
        }
}
```

Método que cambia los puntos de cuadrante, si es que están en el tercer o cuarto, pasándolos al primero o segundo, sumándole 180 grados

```
void cambio_cuadrantes(tLista *lista){
    int i,longitud;
    tLista temp;
    double ang;
    tPunto punto;
    temp=*lista;
    longitud=longLista(temp);
    for(i=1; i<=longitud; i++){
        punto=infoLista(*lista, i);
        Dame_angulo(punto, &ang);
        if (ang<0.0) {
            ang=ang+180.0;
            Negar_Nombre(&temp->punto);
            actualiza_Angulo( &temp->punto, ang);
        }
        temp=temp->sig;
    }
}
```

Procedimiento que calcula la matroide orientada de una lista de puntos, almacenándola

```
en la variable sol

void HS(tLista lista, int *sol[MFIL], int nfil){
    double ang;
    int z,longitud;
    tPunto final, inicio;
    tLista aux1;
    copia(lista,&aux1);
    longitud=longLista(lista);

    buscarPunto(aux1,1,&inicio);/*busca punto que tenga nombre 1*/
    buscarPunto(aux1,2,&final);/*el punto 2 será el final para la primera línea */
    cambio_base(&aux1,1);

    buscarPunto(aux1,1,&inicio);
    buscarPunto(aux1,2,&final);
    Dame_angulo(final, &ang);
    modifica_angulos(&aux1, ang, 1);
    cambio_cuadrantes(&aux1);
    Ordenar_lista_por_angulos(&aux1,1,sol);
    longitud=longLista(lista);
    for(z=2; z<=longitud-1; z++){
        copia(lista,&aux1);
        buscarPunto(aux1,z,&inicio);
        buscarPunto(aux1,1,&final);
        cambio_base(&aux1,z);
        buscarPunto(aux1,z,&inicio);
        buscarPunto(aux1,1,&final);
        Dame_angulo(final, &ang);
        modifica_angulos(&aux1, ang, z);
        cambio_cuadrantes(&aux1);
        Ordenar_lista_por_angulos(&aux1,z,sol);
    }

    copia(lista,&aux1);
    ultimoLista(lista,&inicio);
```

```
    z=inicio.nombre;
    buscarPunto(aux1,z,&inicio);
    buscarPunto(aux1,1,&final);
    cambio_base(&aux1,z);
    buscarPunto(aux1,z,&inicio);
    buscarPunto(aux1,1,&final);
    Dame_angulo(final, &ang);
    modifica_angulos(&aux1, ang, z);
    cambio_cuadrantes(&aux1);
    Ordenar_lista_por_angulos(&aux1,z,sol);
}
```

A.3. TipoArrays

La estructura Tipo Arrays está creada con el propósito de obtener la lista de mutaciones. El dato TArray será un array de 3 enteros. Y el dato tArrays será una lista de TArray. A continuación mostramos la implementación de las funciones principales.

tArray;
int x[3];

Cuadro A.5: Estructura del tipo TArray.

tArrays;
tArray array;
tArrays *sig;

Cuadro A.6: Estructura del tipo TArrays.

```
int buscarPosElemento(int j, int array[],int logitud);
int comparacion(int i, int j, int k, int *matriz[MFIL], int longitud, int numero);
int comprobarAdyacencia(int i, int j, int k, int *matriz[MFIL], int longitud,int numero);
tArray crearArray (int i, int j, int k);
void crearListaArraysVacia(tArrays *lista);
int comparaArray(tArray dato, tArray array);
int noEsta( tArray array, tArrays lista);
void insListaArrays(tArrays *lst, tArray elem);
void ListaMutaciones(tArrays *lista,int *matriz[MFIL],int longitud,int numero );
void damePrimero(tArrays lista,tArray *array);
void eliminaPrimero(tArrays *lista);
void hacerCambios(int *hyperline[MFIL],tArray array, int longitud);
int pertenece(int longitud,tArray array);
int calcularCambios( int *hyperline[MFIL],int *final[MFIL], int longitud, int num);
int comparaMatroide( int *hyperline[MFIL],int *final[MFIL], int longitud);
void imprime_historial(tArrays historial);
void copiaListaArrays(tArrays list, tArrays *copia);
int esObstaculo(tArray array, int longitud, int *hyperline[MFIL]);
int buscarposicion(int x, int longitud,int *hyperline[MFIL]);
tArrays ObtenerObstaculos(int fijos);
void actualizarObstaculos(int numero,tArrays *historial);
void leerObstaculos(tArrays *lista);
```

Cuadro A.7: Funciones del tipo TArrays.

Función que comprueba si los 3 números cumplen adyacencia dentro del array

```
int comprobarAdyacencia(int i, int j, int k, int *matriz[MFIL], int longitud,
                        int numero){
    int respuesta;
    respuesta=0;
    if (((comparacion(i-1,j,k,matriz,longitud,numero)==1) &
        (comparacion(j-1,i,k,matriz,longitud,numero)==1)) |
        ((comparacion(i-1,j,k,matriz,longitud,numero)==1) &
        (comparacion(k-1,i,j,matriz,longitud,numero)==1)) |
        ((comparacion(k-1,i,j,matriz,longitud,numero)==1) &
        (comparacion(j-1,i,k,matriz,longitud,numero)==1)))
    {
    respuesta=1;
    }
return respuesta;
}
```

Función que devuelve la lista de mutaciones que hay en una matroide orientada

```
void ListaMutaciones(tArrays *lista,int *matriz[MFIL], int longitud ,int numero){
    int i,j,k;
    tArray array;
    tArrays tmp;
    *lista=NULL;
    tmp=*lista;
    for(i=1;i<=longitud;i++){
        for(j=1;j<=longitud;j++){
            for(k=1;k<=longitud;k++){
                if((i!=k)&(i!=j)&(j!=k)) {
                    if (comprobarAdyacencia(i,j,k,matriz,longitud,numero)==1){
                        if (k==longitud){
                            array=crearArray (i,j,numero);
```

```
                        }
                        else if (i==longitud){
                            array=crearArray (numero,j,k);
                        }
                        else if (j==longitud){
                            array=crearArray (i,numero,k);
                        }
                        else array=crearArray (i,j,k);
                        if (noEsta(array,tmp)==1){
                            insListaArrays(&tmp,array);
                        }
                    }
                }
            }
        }
    }
*lista=tmp;
}
```

Función que calcula las diferencias que hay entre una matroide y otra.
Será el valor de la función h

```
int calcularCambios( int *hyperline[MFIL],int *final[MFIL],int longitud, int num){
    int i,pos1,pos2,total,parcial;
    total=0;
    parcial=0;
    for (i=0; i<longitud-1;i++){
        pos1=0;
        pos2=0;
        pos1=buscarPosElemento(num,hyperline[i],longitud);
        pos2=buscarPosElemento(num,final[i],longitud);
            if (*(hyperline[i]+pos1)==*(final[i]+pos2)){
                parcial=abs(pos1-pos2);
            }
```

```
            else{
                parcial=abs(pos1-pos2);
                parcial=longitud-2-parcial;
            }
            total=total+parcial;
    }
return total;
}
```

Función que dice si la recta de los dos primeros números del array es parte de un segmento de la recta soporte

```
int esObstaculo(tArray array, int longitud, int *hyperline[MFIL]){
  int resultado,x,y,z,pos1,pos2;
  resultado=1;
  x=array.x[0];
  y=array.x[1];
  z=array.x[2];
  if (x==1){
     if((*(hyperline[longitud-1]+longitud-1)==-y)|(*(hyperline[longitud-1]+2)==y)){
       resultado=0;
     }
  }
  else {
     pos1=buscarposicion(x,longitud,hyperline);
     pos2=buscarposicion(y,longitud,hyperline);
     if (((*(hyperline[longitud-1]+pos1)>0)&(*(hyperline[longitud-1]+pos2)>0)) |
        ((*(hyperline[longitud-1]+pos1)<0)&(*(hyperline[longitud-1]+pos2)<0))){
              resultado=0;
     }
  }
return resultado;
}
```

A.4. TipoDatos

El tipo TDatos es una lista de nodos. Estos nodos tendrán toda la información necesaria para poder llevar a cabo el algoritmo. Esta información será el valor de las funciones f, g y h. El historial de los saltos realizados y la matroide orientada que tiene el nodo actual.

int numero;
int g;
int h;
int f;
tArrays histo;
int *hyper[MFIL];

Cuadro A.8: Elementos por los que está formado la estructura TInfo.

```
int longituddLista(tDatos lst);
void crearListaTotalVacia(tDatos *lista);
tDatos CrearListaDestino(tLista listaMovible, tLista listafija, int fijos);
void anxListaTotal(tDatos *todo, tinfo elemento);
tDatos CrearListaOrigen(tDatos destino, tLista listaMovible, tLista listafija, int fijos);
int AlgoritmoFinal (tinfo inicio, tinfo final, int longitud, tArrays *historial, tArrays obstaculos);
void copiaNodo(tinfo nodoactual, int longitud, tinfo *copianodo);
void insOrdenado(tinfo elem,tDatos *origen);
void sacarNodo(tDatos listaAbierta, tinfo *elem);
void borraNodo(tDatos *listaAbierta);
void imprimoTodo(tDatos lista,int longitud);
int cumplecondicion(tinfo copianodo,tDatos listaAbierta);
void ActualizarF(tinfo *info);
int esta(tinfo copianodo, tDatos nodosVisitados, int longitud);
void ActualizarNodo(tinfo *copianodo,tArray array,tinfo final,int longitud);
int cumpleCondicion(tinfo copianodo,tDatos listaAbierta,int longitud);
void ActualizalistaAbierta(tinfo copianodo,tDatos *listaAbierta,int longitud);
void ordenar(tDatos *listaAbierta);
```

Cuadro A.9: Funciones del tipo TDatos.

Función que crea una lista con todos los nodos origen, con sus respectivas matroides orientadas, y con el historial y numero de saltos inicializados a 0

```
tDatos CrearListaOrigen(tDatos destino, tLista listaMovible, tLista listafija,
                        int fijos){
    int m,n,longitud,longitud2,fila,col;
    tDatos todo;
    tinfo elemento;
    tPunto p1;
    tArrays historial;
    crearListaTotalVacia(&todo);
    crearListaArraysVacia(&historial);
    longitud=longLista(listaMovible);
    longitud2=fijos+1;
    n=fijos+1;
    for(m=1; m<=longitud; m++){
        elemento.numero=n;
        elemento.g=0;
        elemento.histo=historial;
        p1=infoLista(listaMovible,m);
        anxLista(&listafija,p1);
        for(fila=0;fila<fijos+1;fila++)
            elemento.hyper[fila]=(int *)malloc(longitud2*sizeof(int));
            for (fila=0; fila<longitud; ++fila)
                for(col=0; col<fijos+1;++col)
                    *(elemento.hyper[fila]+col)=34;
        HS(listafija,elemento.hyper,fijos+1);
        elemento.h=calcularCambios(elemento.hyper,destino->info.hyper,longitud2,
                                   elemento.numero);
        destino=destino->sig;
        elemento.f=elemento.g+elemento.h;
        elimLista(&listafija, fijos+1);
        anxListaTotal(&todo,elemento);
        n++;
    }
```

```
return todo;
}
```

Función que inserta un nodo de menor a mayor valor de la función f

```
void insOrdenado(tinfo elem,tDatos *origen){
    tDatos aux,aux2;
    aux2=*origen;
    struct informacion *nuevo = malloc(sizeof(struct informacion));
    nuevo->info=elem;
    if (*origen==NULL) {
        nuevo->sig=NULL;
        *origen=nuevo;
    }
    else if (nuevo->info.f < aux2->info.f){
        nuevo->sig=*origen;
        *origen=nuevo;
    }
    else {
        aux=*origen;
        while ( (aux->sig!=NULL)  && (nuevo->info.f >= aux->sig->info.f) ){
            aux=aux->sig;
        }
        nuevo->sig=aux->sig;
        aux->sig=nuevo;
    }
}
```

Algoritmo que calcula el camino más corto entre un nodo inicial y su destino

```
int AlgoritmoFinal (tinfo inicio, tinfo final, int longitud, tArrays *historial,
tArrays obstaculos){
  tDatos listaAbierta,listaCerrados;
  tArray array;tArrays mutaciones;int cant,resultado,cantidad;
  tinfo nodoactual,copianodo;
```

```
listaCerrados=NULL;
listaAbierta=NULL;
nodosVisitados=NULL;
int solucion=0;
insOrdenado(inicio,&listaAbierta);
while ((listaAbierta!=NULL)&(solucion==0)){
  sacarNodo(listaAbierta,&nodoactual);
  borraNodo(&listaAbierta);
  insOrdenado(nodoactual,&listaCerrados);
  if(nodoactual.h==0){
     printf("HE ENCONTRADO SOLUCION\n");
     solucion=1;
  }
  else {
     crearListaArraysVacia(&mutaciones);
     ListaMutaciones(&mutaciones,nodoactual.hyper,longitud ,nodoactual.numero);
     while (mutaciones!=NULL){
        damePrimero(mutaciones,&array);
        eliminaPrimero(&mutaciones);
        if (pertenece(nodoactual.numero,array)==1){
           if ((esObstaculo(array,longitud, nodoactual.hyper)==0) |
              (noEsta(array,obstaculos)==1)) {
                  copiaNodo(nodoactual,longitud,&copianodo);
                  hacerCambios(copianodo.hyper,array,longitud);
                  ActualizarNodo(&copianodo,array,final,longitud);
                  if (esta(copianodo,listaCerrados,longitud)==0){
                      if (esta(copianodo,listaAbierta,longitud)==0){
                         insOrdenado(copianodo,&listaAbierta);
                      }
                      else {
                        if (cumpleCondicion(copianodo,listaAbierta,longitud)==1){
                          ActualizalistaAbierta(copianodo,&listaAbierta,longitud)
                        }
                      }
                  }
           }
```

```
          }
        }/*while*/
      }
    }/*while lista abierta*/
    while(listaCerrados!=NULL){
      borraNodo(&listaCerrados);
    }
    if (listaAbierta==NULL){
      printf("No ha encontrado solucion");
    }
    else {
      while(listaAbierta!=NULL){
          borraNodo(&listaAbierta);
      }
    }
    if (nodoactual.h==0){
      *historial=nodoactual.histo;
      resultado=nodoactual.g;
    }
    else {
      *historial=NULL;
      resultado=0;
    }
  return resultado;
```