UNIVERSIDAD DE SEVILLA

Facultad de Física

Doble Grado en Física e Ingeniería de Materiales

Departamento de Ingeniería de Sistemas y Automática

Departamento de Electrónica

Planificación de Trayectorias para Robots Autónomos en Entornos Industriales

Trabajo Fin de Grado

presentado por

Rafael Rey Arcenegui

Dirigido por:

Fernando Caballero Benítez,

Gloria Huertas Sánchez.

Sevilla, Junio de 2019

Índice general

Capítulo 1

Introducción

Actualmente en el campo de la robótica encontramos dos ramas bien diferenciadas en lo que respecta a escala de uso, costes y usuarios finales. Por un lado existe la robótica industrial, un conjunto de tecnologías presentes en el mercado desde hace décadas que ha permitido la automatización de innumerables procesos de fabricación industriales con alto nivel de especialización. Un sector que se ha transformado profundamente gracias a esta robótica industrial es la industria automovilística, en la cual se usan líneas de montaje con brazos robóticos en casi todo el proceso de fabricación de los vehículos.

Por otro lado, tenemos el sector de la robótica de servicios. Aún no existe total consenso en cuanto a la definición de robot de servicios, pero según la IFR[1] se trata de un robot que opera de forma semi o totalmente autónoma para realizar servicios útiles para el bienestar de los seres humanos y equipos, excluyendo las operaciones de fabricación [1]. Dentro de los robots de servicios, existen los de uso profesional y los de uso personal según la aplicación final a la que se destinen.

Un ejemplo de robot de servicio comercial de uso profesional sería por ejemplo el prototipo de robot repartidor de Amazon (Fig. 1.1a), mientras que un robot de servicios de uso personal sería la aspiradora automática Roomba (Fig. 1.1b)

Dentro de esto, la planificación de caminos o trayectorias es un parte fundamental del problema del movimiento autónomo del robot. Para esta tarea existen numerosos

[1]IFR: International Federation of Robotics

(a) Prototipo de robot repartidos de la compañia Amazon

(b) Robot aspiradora automático Roomba 980

Figura 1.1: Ejemplos de robots de servicios

enfoques, nosotros concretamente usaremos un algoritmo heurístico, pero también existen otras aproximaciones, como los algoritmos probabilísticos por ejemplo. En el Capítulo 2 se describirá el estado del arte de las diferentes metodologías existentes para abordar la resolución de tal problemática.

1.1. El problema de la planificación

El problema de la planificación de forma general estudia como un sistema puede pasar de estado inicial a un estado final o *goal* dada una serie de posibles acciones simples en un espacio dado. Así pues, la resolución de un cubo de Rubbik es un ejemplo de problema de planificación en el cual el estado inicial puede ser cualquier configuración aleatoria del cubo, el estado final el cubo resuelto y las acciones, los giros de las caras. La solución a este problema consistiría en una secuencia ordenada de giros que partiesen del estado aleatorio y terminasen en el estado ordenado o resuelto.

De forma más concreta, en el campo de la robótica, la planificación se refiere a la automatización de sistemas mecánicos con sensores, actuadores y capacidad de computación [2]. Esto crea la necesidad de encontrar algoritmos eficientes que transformen órdenes de alto nivel en comandos de movimiento de bajo nivel. La necesidad de eficiencia suele venir impuesta por las limitaciones computacionales del sistema.

1.2. Planificación de caminos

La planificación de caminos es una de las partes esenciales involucradas en el funcionamiento de un robot autónomo. Sin embargo, esto es una aplicación concreta de la planificación de caminos, ya que el objetivo general de ésta es encontrar el camino más corto entre dos puntos o en general el camino óptimo. De esta forma, dependiendo de como se defina la métrica, podremos aplicarlo a diferentes campos. Es por esto que este tipo de algoritmos también son importantes en *network routing*, videojuegos o en la comprensión de como se doblan las proteínas [3].

En el campo de la robótica, la planificación de caminos o trayectorias responde a la necesidad de ejecutar una serie de acciones para ir de un estado inicial hasta un estado final de forma realizable y segura. Entiéndase aquí por camino o trayectoria realizable aquella que es capaz de ejecutar la plataforma concreta, dadas sus limitaciones mecánicas.

En el Capítulo 2 describiremos las metodologías existentes para planificar tanto caminos como trayectorias así como sus virtudes y defectos, ya que no existe una solución universal al problema, si no que debe elegirse la que mas convenga según los requerimientos de la aplicación considerada.

La diferencia entre planificación de caminos y trayectorias radica en que las trayectorias son una sucesión de estados temporales, mientras que los caminos son una sucesión de puntos espaciales separados una distancia que puede ser arbitrariamente pequeña (en la práctica la separación máxima útil es la resolución del mapa usado).

La planificación de trayectorias se suele referir al problema de como moverse a lo largo de un camino dado (sucesión de puntos en el espacio sin obstáculos entre ellos) respetando las limitaciones mecánicas del sistema robótico (restricciones de velocidad y aceleración por ejemplo).

1.3. Marco del trabajo

El contenido de este trabajo se enmarca dentro de las tareas del proyecto ARCO[2].

[2]Autonmous Robot Co-Worker

Se trata de un proyecto europeo financiado por el proyecto HORSE (número de subvención 680734) que a su vez se enmarca en el programa Horizon2020. El proyecto tiene como objetivo el diseño de robot autónomo destinado a ayudar en el transporte de materiales dentro de pequeños y/o medianos almacenes o líneas de producción con vistas a la Industria 4.0 y el uso de IoT[3]. Este sistema permitiría combinar la extrema flexibilidad y adaptación de los humanos con la seguridad y control de los robots terrestres.

Esta plataforma robótica permitiría reducir la carga de trabajo físico que actualmente soportan los operarios que trasladan por el almacén las numerosas mercancías. Además de esta reducción en el trabajo lesivo, esta plataforma robótica permitiría la paralelización de tareas de forma que mientras el robot desplaza materiales, los operarios del almacén podrían desempeñar otras tareas de forma simultánea, incrementando la productividad. Este sistema es particularmente interesante para almacenes y fabricas pequeñas y medianas donde el coste de la robotización total no es asumible.

En concreto, el proyecto ARCO se validará en la sede en Lisboa de la empresa portuguesa Tintas Robbialac S.A., un fabricante de pinturas perteneciente al grupo Cromology, uno de los líderes mundiales en el área de pinturas. En el Capítulo 4 se mostrarán varios resultados de los experimentos realizados en dicha planta.

El contenido de este trabajo se centra en el apartado de la planificación de caminos, uno de los elementos centrales en la navegación autónoma. Esta navegación autónoma consta de dos partes bien diferenciadas, la planificación del camino y la ejecución de este. Esto último sería tarea de un módulo de path-tracking. Una descripción completa del sistema puede verse en [4].

Existen ciertos requisitos de funcionamiento relativos a la seguridad que condicionan el enfoque que se la ha dado en este trabajo a la planificación de trayectorias. Para asegurar que el sistema responde rápidamente a la aparición de obstáculos inesperados, se adoptará una estrategia de replanificación local continua. Esta replanificación será solo de caminos, y luego el seguidor de caminos se encargará de enviar comandos de velocidad siguiendo un algoritmo simple, por lo que las trayectorias serán creadas en tiempo real a partir del camino.

[3]Internet of Things

Capítulo 2

Metodologías para la Planificación de Caminos

Como se ha dicho anteriormente, existen diferentes métodos que se diferencian por el enfoque del que parten. A continuación introduciremos tres grandes grupos, a saber: las metodologías probabilísticas (RRT por ejemplo), los métodos de campos potenciales y los métodos heurísticos. Como ya hemos dicho, el algoritmo usado en este trabajo, Lazy Theta* pertenece a éste último grupo.

Por otro lado, en este trabajo hablaremos mas propiamente de planificación de caminos que de trayectorias, entendiéndose por camino el conjunto de traslaciones y rotaciones necesarias para mover un objeto (el robot en nuestro caso) entre dos puntos. La planificación de trayectorias partiría del camino resultante anterior y resolvería el problema de mover el robot a lo largo de estos puntos respetando las limitaciones mecánicas del robot, de esto último se encargará el módulo de *seguimiento de caminos* descrito en la sección 3.4.

En general, cualquier enfoque, necesita una información mínima para calcular un camino o trayectoria entre un estado A y un estado B. Por un lado se necesita un mapa del entorno, que puede ser estático o dinámico. Por otro lado necesitamos saber las coordenadas del objetivo en ese mapa y finalmente pero no menos importante la propia posición y orientación del robot. Para tener una estimación fiable de esta última se utilizan diferentes sensores (odometría, láseres, cámaras, sistemas RF, IMU)

además de algoritmos que combinen estas informaciones (AMCL[1], filtros de Kalman).

Según sean los requerimientos de la aplicación concreta y el escenario donde navegará el robot, se elige una metodología u otra ya que cada una conlleva una serie de ventajas y desventajas que trataremos en las próximas secciones. Los criterios para elegir una metodología u otra dependen se basan en aspectos como los costes computacionales, la libertad de elección de caminos, longitud de estos comparada con la longitud mínima posible y aspecto realista de los caminos resultantes entre otros.

2.1. Algoritmos basados en el muestro (Sampling-Based motion planning)

El funcionamiento de esta clase de algoritmos consiste en realizar una búsqueda en el espacio de la configuración sin construir explícitamente este espacio, en el que se representan los posibles estados del robot en cada instante de tiempo.

Una de las razones que impulsa a evitar esta construcción explícita del espacio de la configuración es que para construirlo, se debe tener en cuenta el espacio de la configuración de obstáculos. Este espacio es aquel que incluye todas las configuraciones susceptibles de provocar colisiones con objetos externos o con partes del propio robot (caso de brazos articulados, por ejemplo) [2]. Por otro lado, esta construcción implica una mayor complejidad en el algoritmo y su implementación, lo que repercutiría negativamente en el tiempo necesario para la ejecución.

De esta forma, este tipo de algoritmos sondea el espacio de la configuración con un esquema de muestreo. Para evitar configuraciones del espacio de obstáculos se emplea un módulo de detección de colisiones que a los ojos de este tipo de algoritmos es una caja negra (Fig. 2.1). Esto permite al algoritmo ser independiente del modelo geométrico particular, lo que constituye una clara ventaja.

A continuación se describen dos algoritmos de esta clase utilizados en planificación. Como se ha dicho anteriormente, la diferencia fundamental entre cada uno es la forma de muestreo del espacio de la configuración.

[1]AMCL: Adaptative Monte Carlo Localization

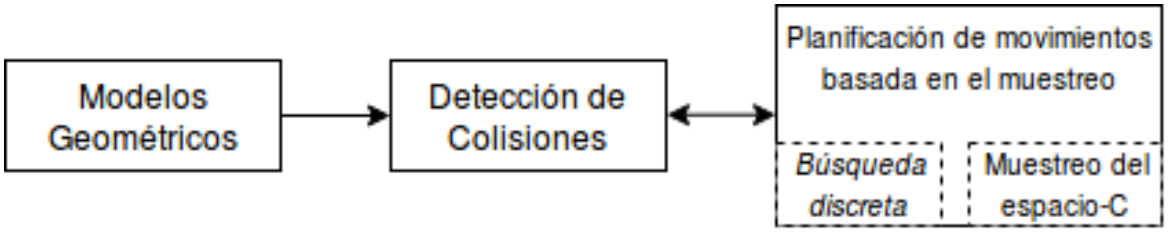


Figura 2.1: Modelo general de los métodos basados en el muestreo [2]

2.1.1. Probabilistic Roadmaps (PRM)

Este método consiste en una primera fase de aprendizaje y una segunda fase de consulta o *query*. Fue propuesto por Lydia E. Kavraki et al. [5] en 1996.

En la fase de aprendizaje se construye un mapa probabilístico (PRM) partiendo de puntos aleatorios del espacio de la configuración y se almacena como un grafo cuyos nodos corresponden a estos estados del espacio de la configuración libre de colisiones y cuyos bordes o uniones corresponden a posibles caminos entre los estados. Esta fase de construcción se extiende hasta que se alcanza una densidad de puntos adecuada que permita una transición suave entre estados.

Finalmente, en la fase de consulta o *query* se unen los nodos entre los estados inicial y final usando otro algoritmo de búsqueda del camino más corto como puede ser el Dijsktra que se describirá en la sección 2.3.1.

Como ventajas podemos decir que es muy eficiente, sencillo de implementar y aplicable a gran variedad de problemas de planificación, sin embargo este enfoque deja varios aspectos a elección libre como el método de muestrear el espacio o el planificador para la fase de *query* elegido [6].

Las fases que más influencian el tiempo necesario para la ejecución del algoritmo y la estructura del Road Map resultante son la elección de configuraciones útiles (linea 3 en 2.3) y la elección de pares de nodos útiles (linea 5 en 2.3).

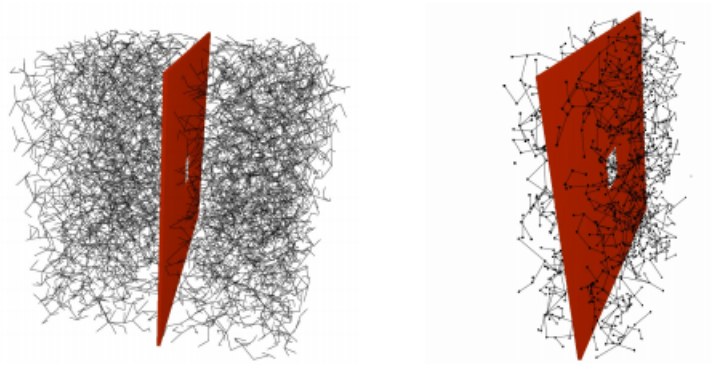


Figura 2.2: Ejemplo de grafo PRM para en un escenario complicado. A la izquierda el resultado de usar muestreo halton y a la derecha muestreo gaussiano [6].

```
1  Sea V ← ∅; E ← ∅ ;
2  Loop()
3  |  c ← a Configuración útil en C_free ;
4  |  V ← V ∪ c ;
5  |  N_c ← Conjunto de nódos útiles tomados de V ;
6  |  foreach c' ∈ N_c en orden creciente de distancia desde c do
7  |  |  if c' y c no están conectados en G then
8  |  |  |  if Planificador local encuentra camino entre c y c' then
9  |  |  |  |  Añadir borde entre c' y c a E
10 |  |  |  end
11 |  |  end
12 |  end
13 end
```

Figura 2.3: Pseudocódigo de construcción de un RoadMap

2.1.2. Rapidly-exploring Random Trees (RRT)

Propuesto en 1996 por Steven M. LaValle [7] se ha convertido, junto con sus decenas de variantes en uno de los algoritmos más usados en planificación. Como su nombre indica, emplea árboles como tipos de datos, que imitan la estructura jerárquica de estos (Fig. 2.4).

Este algoritmo esta diseñado específicamente para tratar problemas con restricciones

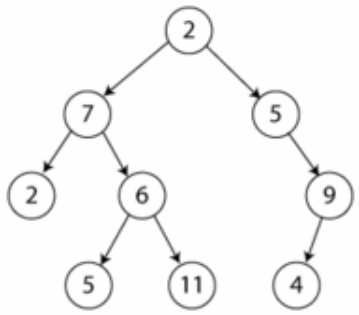


Figura 2.4: Representación de una estructura de datos tipo árbol. Cada nodo puede tener varios hijos pero cada uno solo puede tener un padre.

```
1 G.init(q_init);
2 for k = 1 to K do
3 |   q_rand ← RANDOM_STATE();
4 |   q_near ← NEAREST_NEIGHBOR(q_rand, G));
5 |   q_new ← NEW_CONF(q_near, q_rand, Δq);
6 |   G.add_vertex(q_new);
7 |   G.add_edge(q_near, q_new);
8 return G;
```

Figura 2.5: q_i: Configuración inicial, K: Número de vértices, Δq: Incremento de distancia entre vértices.

no-holónomas[2](incluyendo dinámicas y kinodinámicas) y sistemas con muchos grados de libertad, como brazos robóticos multiarticulados.

El árbol en el espacio de la configuración se crea a partir de unos puntos aleatorios, y crece de forma dirigida hacia las áreas mas inexploradas del espacio. El pseudocódigo puede verse en la figura 2.5.

Entre las ventajas de los RRTs podemos señalar su simplicidad y el hecho de que todos los vértices se mantienen conectados y que son probabilísticamente completos bajo condiciones muy generales [7], mientras que una desventaja importante consiste en el hecho de que no son deterministas, pudiendo dar diferentes resultados en cada ejecución. Las variantes existentes de este algoritmo se diferencian sobre todo en como muestrean el espacio y como lidian con problemas como el no-determinismo para obtener resultados controlables.

[2]Aquellas que no pueden expresarse mediante una función de la forma $f(\vec{r}_1, \vec{r}_2, ..., t)) = 0$

Figura 2.6: Ejemplo de exploración por el algoritmo RRT. A la izquierda, después de 45 iteraciones. A la derecha, 2345 iteraciones. Puede verse como el RRT alcanza rápidamente las zonas inexploradas [2]

2.2. Métodos de campos potenciales

Este acercamiento al problema de encontrar el camino óptimo consiste en crear un campo potencial artificial en el espacio donde se mueve el robot. El caso más simple emula el potencial eléctrico. El punto objetivo tendría un potencial atractivo mientras que los obstáculos uno repulsivo de forma que cuando el robot se acerca a un obstáculo, aparece una fuerza repulsiva. Las ecuaciones 2.1 constituyen la base de éste método.

$$\begin{aligned} U(q) &= U_{att}(q) + U_{rep}(q) \\ \vec{F}(q) &= -\vec{\nabla} U(q) \end{aligned} \tag{2.1}$$

De esta forma, uno de de las formas de encontrar el camino objetivo sería considerar aquel que va en la dirección negativa del gradiente en cada punto, y el camino termina cuando se llega a un punto donde se anula el gradiente, es decir, un punto crítico. Este enfoque se conoce como *Gradient Descent* [8]. El pseudo-código de este algoritmo puede verse en la figura 2.7

De aquí surge uno de los principales problemas de este tipo de métodos, los mínimos locales de potencial. Dado que el robot tendería a moverse a las zonas de mínimo potencial, si entre su posición y el punto objetivo existe un mínimo local, cabe la

Input : Un medio de calcular el gradiente $\nabla U(q)$ en un punto q
Output: Una secuencia de puntos $\{q(0), q(1), ..., q(i)$

```
1 q(0) = q_start;
2 i = 0 ;
3 while ∇U(q) ≠ 0 do
4 |   q(i + 1) = q(i) + α(i)∇U(q(i));
5 |   i = i + 1;
```

Figura 2.7: Pseudo-código del algoritmo Gradient Descent [8]

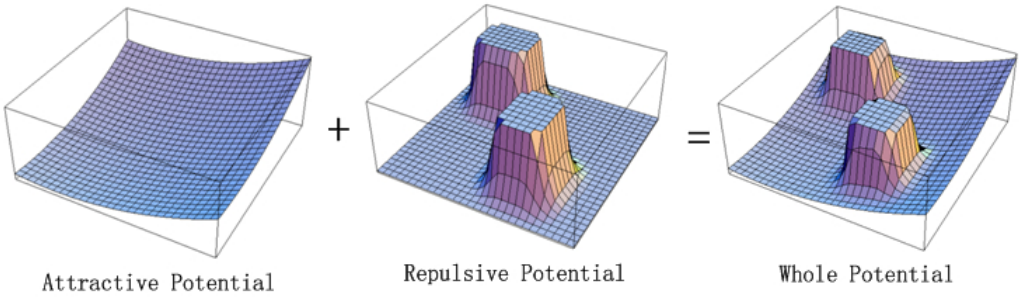


Figura 2.8: Ejemplo de potenciales atractivos y repulsivos y potencial resultante. Fuente: `https://taylorwang.files.wordpress.com/2012/04/potential-field1_robot.jpg`

posibilidad de que entre en él y no pueda salir. No obstante existen varios enfoques que tratan de solucionar esta problemática como el propuesto por H. Noborio et .al [9], así como la introducción de campos rotacionales alrededor de los obstáculos o los campos aleatorios.

2.3. Algoritmos heurísticos

En general una técnica heurística es un procedimiento práctico o informal de resolver problemas, es decir, busca soluciones a problemas concretos que pueden ser muy difíciles de abordar de forma rigurosa o analítica o poco práctico.

Los métodos o algoritmos heurísticos surgen de la necesidad de encontrar una solución a problemas que por su complejidad, resulta poco práctico buscar soluciones analíticas o exactas. Esto puede darse bien porque no existen o bien por que es innecesario un nivel de detalle tan profundo. Estos son un conjunto de métodos puramente

prácticos.

En este tipo de métodos el problema se reduce a buscar un camino dentro de un grafo que representa el espacio donde se mueve el robot. Una forma común de representar el espacio para este tipo de algoritmos es discretizándolo en forma de una malla de vértices. Uno de los problemas que suelen surgir en este tipo de algoritmos es que el camino resultante requiere un post-procesado para suavizar los caminos obtenidos lo que repercute en el tiempo de computación total necesario.

Existen multitud de algoritmos dentro de esta categoría y cada uno con multitud de variantes, algunos de ellos son Dijkstra, A*, Theta*, Lazy Theta*, D*, ARA*. En esta sección explicaremos concretamente los algoritmos Dijkstra, A*, Theta* y finalmente Lazy Theta*, el usado en la implementación de los planificadores descritos en el Capítulo 3.

2.3.1. Dijkstra

Fue publicado en 1959 por E.W. Dijkstra [10]. Se trata de un algoritmo de búsqueda con coste uniforme, o dicho de otra forma, emplea una heurística de valor 0. En lo que sigue nos referiremos al valor de la heurística como la estimación de la distancia al punto objetivo.

Esta heurística cero hace que explore todos los nodos contiguos al nodo inicial y así sucesivamente de forma uniforme hasta que llega al nodo final. A continuación se describen los pasos en el algoritmo. Sea $A = S$ con S el nodo inicial y A el nodo actual:

1. Se marcan todos los nodos como *no visitados* y se crea el conjunto correspondiente

2. Se inicializan todas las distancias relativas como infinito ya que son desconocidas salvo la distancia del nodo inicial a si mismo que es 0.

3. Para el nodo A, se calculan todas las distancias de sus vecinos no visitados al nodo inicial. Se comparan las distancias resultantes con las ya calculadas anteriormente. Si la calculada en esta iteración para un nodo dado es menor que la distancia que constaba anteriormente, se sustituye ésta por la nueva.

4. Una vez recorridos todos los vecinos sin visitar de A, se marca el nodo actual como visitado y se elimina del conjunto de nodos no visitados. Un nodo visitado no se volverá a analizar nunca.

5. Si el nodo objetivo se ha marcado como visitado o si la menor distancia entre nodos del conjunto de nodos sin visitar es infinita, el algoritmo ha terminado. En el caso contrario toma como próximo nodo A aquel con la menor distancia y se regresa al paso 3 mientras existan nodos no visitados.

Como puede verse se trata de un algoritmo simple, fácil de implementar.

2.3.2. A*

Este algoritmo es una extensión del Dijkstra, fue publicado en 1968 por Peter Hart, Nils Nilsson y Bertram Raphael [11]. Usando una heurística no trivial (diferente de cero) obtiene un mejor rendimiento que su predecesor, pues realiza una exploración dirigida al nodo objetivo. Se trata de uno de los algoritmos más extendidos.

La diferencia fundamental respecto al Dijkstra, es que en cada iteración A* toma como nodo siguiente aquel con el menor valor de una función $f(n)$ definida como sigue:

$$f(n) = g(n) + h(n) \tag{2.2}$$

Donde:

- $g(n)$ es la distancia del nodo n al nodo inicial siguiendo el camino generado hasta el momento para llegar a n

- $h(n)$ es la estimación de la distancia desde el n hasta el nodo objetivo o *goal*. Esta es la función que se conoce comúnmente como *heurística*

Como vemos, el algoritmo Dijkstra es un caso concreto de A* con $h(n) = 0$ para todo n. La dificultad en este tipo de algoritmos reside en seleccionar o diseñar la heurística adecuada. En la figura 2.9 podemos ver el pseudo-código, que nos será útil para explicar las diferencias de A* con Theta* y Lazy Theta*.

Como vemos en el pseudo-código, el algoritmo parte de una lista "abierta" y otra "cerrada". El nodo de partida se introduce en la lista abierta y mientras esta lista no esté vacía se toma el nodo s de menor f en ella. Este nodo s se saca de la lista abierta y se generan sus ocho vecinos, asignándoles como padre s.

A continuación para cada vecino s', se comprueba si es el nodo objetivo, en este caso se ha terminado la búsqueda. En caso contrario, si el nodo no está en la lista cerrada se comprueba si tampoco está en la lista abierta. En este caso se le asigna $g = \infty$ y se pone a nulo su padre. A continuación se actualiza el valor de g del nodo comprobando si la distancia del nodo inicial al nodo s más la distancia entre s y s' es menor que la distancia del nodo inicial a s'. En caso afirmativo se actualiza el valor $g(s')$ y se le asigna a s' el nodo s como padre.

2.3.2.1. Ejemplos de heurísticas

Según la libertad de movimiento en una malla de puntos o *grid* bidimensional, podemos distinguir tres casos generales para lo que existe consenso en cuanto a la heurística óptima que elegir. A continuación se describen a modo de ejemplo.

- Movimiento en cuatro direcciones (horizontal y vertical): Distancia Manhattan

$$h(n) = |n_x - goal_x| + |n_y - goal_y| \tag{2.3}$$

- Movimiento en ocho direcciones (horizontal, vertical y diagonal): Distancia diagonal, el máximo de los valores absolutos de las diferencias de las coordenadas x e y del nodo actual y el objetivo

$$h(n) = max(|n_x - goal_x|, |n_y - goal_y|) \tag{2.4}$$

- Movimiento en cualquier dirección: Distancia euclídea

$$h(n) = \sqrt{(n_x - goal_x)^2 + (n_y - goal_y)^2} \tag{2.5}$$

Estas heurísticas tienen la propiedad de ser consistentes. Es decir la estimación que hacen de la distancia a un nodo N es siempre igual o menor a la estimación de

```
1  Main()
2      open := closed := ∅;
3      g(s_start) := 0;
4      parent(s_start) := s_start;
5      open.Insert(s_start, g(s_start) + h(s_start));
6      while open ≠ ∅ do
7          s := open.Pop();
8          if s = s_goal then
9              return "path found";
10         closed := closed ∪ {s};
11         foreach s' ∈ nghbr_vis(s) do
12             if s' ∉ closed then
13                 if s' ∉ open then
14                     g(s') := ∞;
15                     parent(s') := NULL;
16                 UpdateVertex(s, s');
17     return "no path found";
18 end
19 UpdateVertex(s, s')
20     g_old := g(s');
21     ComputeCost(s, s');
22     if g(s') < g_old then
23         if s' ∈ open then
24             open.Remove(s');
25         open.Insert(s', g(s') + h(s'));
26 end
27 ComputeCost(s, s')
28     /* Path 1 */
29     if g(s) + c(s, s') < g(s') then
30         parent(s') := s;
31         g(s') := g(s) + c(s, s');
32 end
```

Figura 2.9: Pseudo-código del algoritmo A*

distancia desde cualquier vecino directo de N al objetivo más la distancia de alcanzar este vecino.

$$h(N) \leq c(N, P) + h(P) \tag{2.6}$$

Esto no es más que la desigualdad triangular. Por otro lado, otro concepto relevante

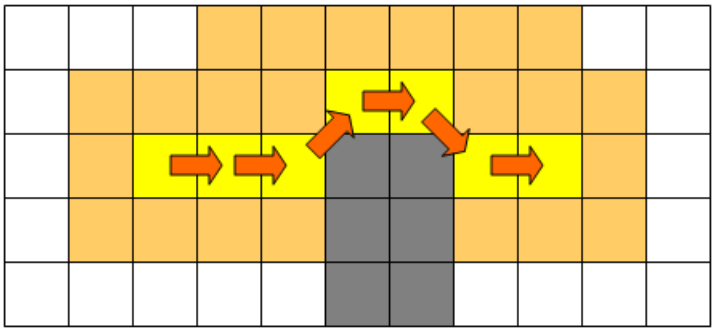


Figura 2.10: Ejemplo de exploración realizada por A* en un malla con un obstáculo y posibilidad de movimiento diagonal

aquí es el de heurística admisible, esto es, una heurística que nunca sobrestima el coste o distancia al nodo objetivo. Todas las heurísticas admisibles son consistentes, pero lo contrario no es cierto [12].

2.3.3. Theta*

Se trata de una variación de A*, propuesto en 2010 por K.Daniel et al. [13]. Esta variación surge del hecho de que el algoritmo A* siempre encuentra el camino más corto en el grafo, sin embargo este camino más corto en el grafo no tiene por qué coincidir con el camino más corto en el entorno continuo. Esto se debe a que el algoritmo A* restringe el camino a los bordes del grafo.

El algoritmo Theta* parte de la base de A* pero elimina esta restricción, lo que convierte a Theta* en un algoritmo de *Any-angle* según los autores. En la figura 2.11 se ilustra la diferencia entre los caminos encontrados por Theta* y A*.

La diferencia entre ambos algoritmos radica en que Theta* permite que el padre de un vértice sea cualquier vértice visible, al contrario que A*, donde el padre tiene que ser obligatoriamente un vecino desocupado. La diferencia en el pseudo-código de ambos es mínima, afectando solo a la función de cálculo de costes, la figura 2.12 muestra la función de cálculo de costes usada por Theta*.

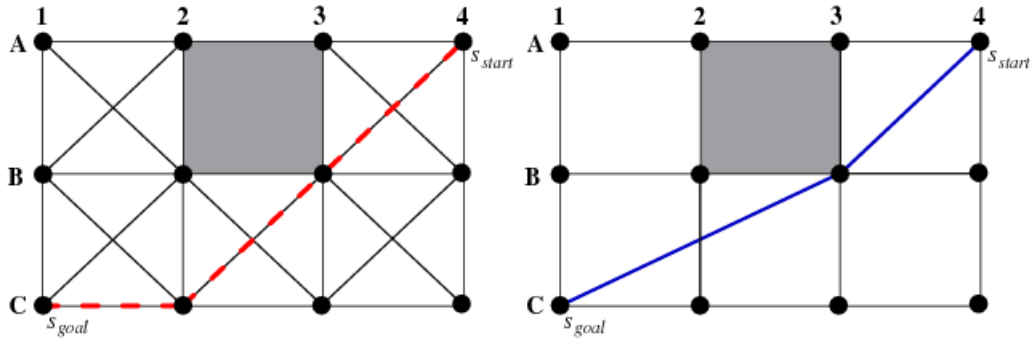


Figura 2.11: A la izquierda, el camino más corto con la restricción de pasar por los bordes del grafo, resultado del algoritmo A*. A la derecha, camino calculado por Theta* sin la restricción mencionada.

```
1  ComputeCost(s, s')
2      if LineofSight(parent(s), s') then
3          /* Path 2 */
4          if g(parent(s)) + c(parent(s), s') < g(s') then
5              parent(s') := parent(s);
6              g(s') := g(parent(s)) + c(parent(s), s');
7      else
8          /* Path 1 */
9          if g(s) + c(s, s') < g(s') then
10             parent(s') := s;
11             g(s') := g(s) + c(s, s');
12 end
```

Figura 2.12: Función ComputeCosts del algoritmo Theta*

2.3.4. Lazy Theta*

Este es el algoritmo usado en la implementación del sistema planificación que se ha realizado. Se trata de una variación más reciente de Theta*. Al igual que Theta* es un algoritmo de *any-angle*.

Esta variante del algoritmo Theta* surge del hecho de que este último realiza muchos más análisis de visibilidad o *line of sight checks* de los necesarios como puede verse en la figura 2.13.

Lo que sucede es que si Theta* comprueba la visibilidad entre un vértice y su padre, y este vértice nunca llega a expandirse (es decir, no se llegan a analizar sus vecinos), la

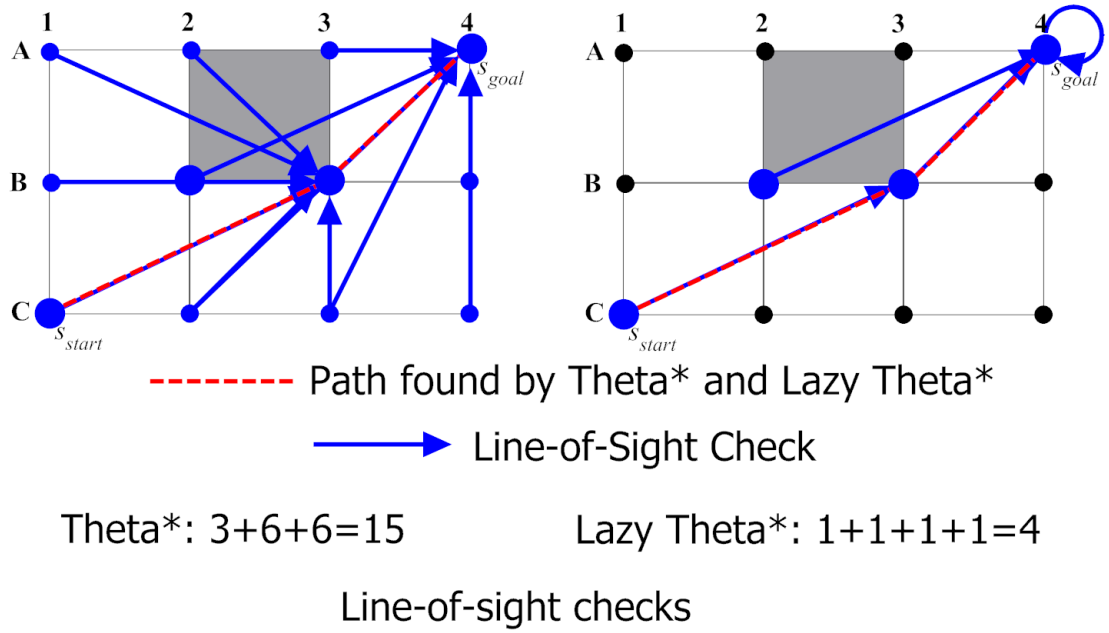


Figura 2.13: Diferencia entre los análisis de linea de visibilidad realizados por Theta* y Lazy Theta*. Fuente: AiGameDev.Com

comprobación ha malgastado tiempo de cálculo. De ahí la denominación de la variante *lazy*, ya que realiza menos cálculos pero devuelve prácticamente los mismos caminos que Theta* (Fig. 2.13).

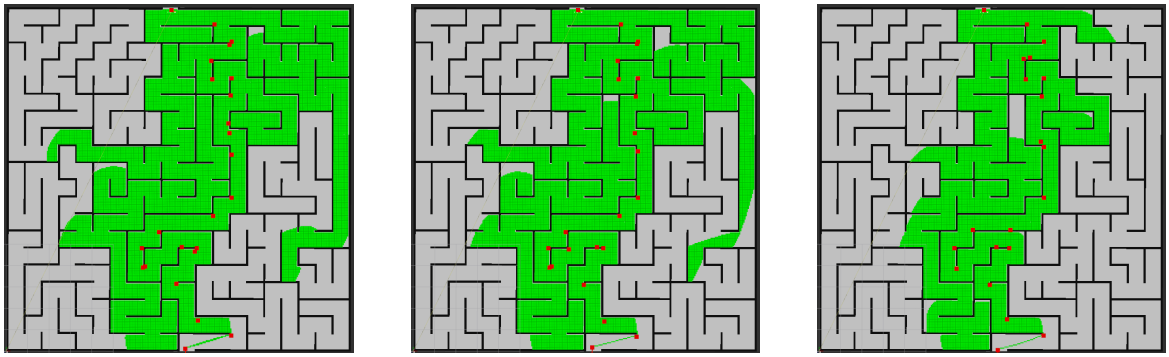
Dados tres vértices, s, s' y el padre de s, Lazy Theta* asume que hay linea de visibilidad ente s' y el padre de s, pudiendo omitir en el camino al vértice s. Esta suposición puede ser por supuesto incorrecta, de ahí que Lazy Theta* lo compruebe en el procedimiento $SetVertex$ (algoritmo 2,14), justo después de expandir el vértice s'. Si resulta que efectivamente hay línea de visión entre s' y el padre de s, no cambia nada. En el caso contrario recalcula el valor g y el padre de s'. El ahorro en número de cálculos proviene de retrasar la comprobación de la línea de vista.

2.3.4.1. Lazy Theta* con optimizaciones

Se denomina así a una modificación adicional de este algoritmo que mejora notablemente la eficiencia. En este caso, la optimización se traduce en una heurística modificada.

```
1  Main()
2      open := closed := ∅;
3      g(s_start) := 0;
4      parent(s_start):=s_start;
5      open.Insert(s_start,g(s_start) + h(s_start));
6      while open ≠ ∅ do
7          s := open.Pop();
8          SetVertex(s);
9          If s = s_goal then
10             return "path found";
11         closed := closed ∪s;
12         foreach s' ∈ nghbr_vis(s) do
13             If s' ∉ closed then
14                 If s' ∉ open then
15                     g(s') := ∞;
16                     parent(s') := NULL;
17                 UpdateVertex(s,s');
18         return "no path found";
19 UpdateVertex(s,s')
20     g_old := g(s');
21     ComputeCost(s,s');
22     If g(s') < g_old then
23         If s' ∈ open then
24             open.Remove(s');
25         open.Insert(s',g(s') + h(s'));
26 ComputeCost(s,s')
27     /* Path 2 */
28     If g(parent(s)) + c(parent(s), s') < g(s') then
29         parent(s') := parent(s);
30         g(s') := g(parent(s)) + c(parent(s), s');
31 SetVertex(s)
32     If NOT lineofsight(parent(s),s) then
33         /* Path 1 */
34         parent(s) := argmin (s' ∈ nghbr_vis ∩ closed(g(s') + c(s, s')));
35         g(s) := min (s' ∈ nghbr_vis ∩ closed(g(s') + c(s, s')));
```

Figura 2.14: Algoritmo Lazy Theta*



(a) $w = 1$, $t = 525ms$, $l = 28{,}2m$ **(b)** $w = 1{,}1$, $t = 484ms$, $l = 28{,}2m$ **(c)** $w = 1{,}5$, $t = 368ms$, $l = 28{,}5m$

Figura 2.15: Comparativa entre tres valores diferentes de *goal weight* con sus correspondientes resultados de tiempos y distancias. En verde los nodos explorados y en rojo los puntos del camino resultante. La resolución del mapa es de 0.05m/pixel.

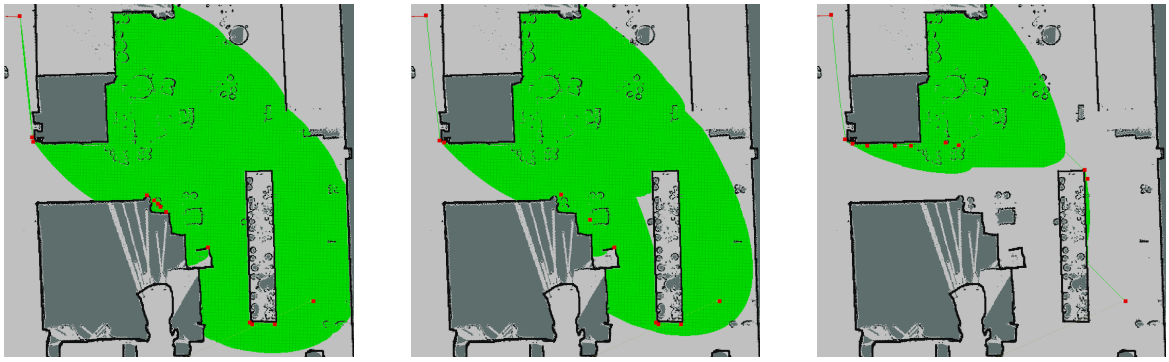
Como ya se describió en la sección 2.3, esta clase de algoritmos siempre encuentran una solución si existe ya que usan heurísticas consistentes (esto es que obedecen la desigualdad de los triángulos, la distancia euclídea por ejemplo). Sin embargo, existe la opción de usar heurísticas inconsistentes ponderando los valores h calculados por el algoritmo.

De esta forma, Lazy Theta* con optimizaciones obtiene caminos ligeramente más largos pero reduce notablemente el tiempo de cálculo. En la ecuación 2.7 se muestra la heurística, donde $w > 1$ es el peso.

$$h(s) = w \cdot c(s, s_{goal}) \tag{2.7}$$

Como vemos en la figura 2.15 los tiempos se reducen un 8 % y un 30 % para pesos de 1.1 y 1.5 respectivamente lo que constituye una mejora notable. Visualmente también se aprecia como de izquierda a derecha disminuye el número de nodos explorados, que es proporcional al tiempo. El número de nodos explorados es una indicación más relevante que el tiempo, puesto que el tiempo depende de la implementación realizada así como de la velocidad del procesador sobre el que se ha ejecutado el algoritmo.

En cuanto a la longitud del camino ésta no se ve prácticamente afectada, lo que es beneficioso, sin embargo en el laberinto usado como mapa en ésta figura los caminos



(a) $w = 1$, $t = 6362ms$, $l = 38{,}10m$ **(b)** $w = 1{,}1$, $t = 4950ms$, $l = 38{,}25m$ **(c)** $w = 1{,}5$, $t = 2624ms$, $l = 39{,}50m$

Figura 2.16: Comparativa entre tres valores diferentes de *goal weight* con sus correspondientes resultados de tiempos y distancias en un mapa real. En verde los nodos explorados y en rojo los puntos del camino resultante. La resolución del mapa es de 0.05m/pixel.

están muy restringidos por lo que era de esperar que las longitudes de los caminos fuesen similares.

En la figura 2.16 podemos ver más claramente la diferencia entre los resultados con diferentes ponderaciones. En este caso se trata de un mapa de un escenario real, donde los caminos no están tan restringidos como en el laberinto anterior. Vemos que para los pesos de 1.0 y 1.1 el algoritmo produce un camino similar con un número de nodos explorados similar. Sin embargo para un peso de 1.5 pasa por una zona diferente, lo que conlleva un aumento de la distancia del camino en mas de $1m$ (entorno a un 2 % superior) con una reducción de tiempo de cálculo de casi un 60 %.

Ambas series de figuras han sido obtenidas a partir de la implementación propia del planificador global que se detalla en la sección 3.2.

Capítulo 3

Implementación y uso del algoritmo Lazy Theta* en un planificador global y local

En este capítulo se detallarán las estructuras de los sistemas implementados así como la metodología seguida y las herramientas usadas para ello.

La implementación de los planificadores local y global así como del seguidor de caminos se ha llevado a cabo en C++ utilizando el *framework* ROS (Robot Operating System) en su versión Kinetic ejecutándose en Ubuntu 16.04. Se ha elegido este marco de implementación por su gran acogida por parte de la comunidad de desarrolladores de software y laboratorios de investigación de robótica. Además existen numerosos robots comerciales que funcionan con ROS, como por ejemplo el Tiago, de la emrpesa española Pal Robotics.

ROS es un conjunto de librerías, paquetes y herramientas flexibles para escribir software para robótica. Cada aplicación en ROS es un nodo que pertenece a un paquete. Los nodos se comunican entre ellos publicando información en *topics* y suscribiéndose a estos *topics*. Así por ejemplo, un nodo que actúe como un *driver* de un sensor publicará en un *topic* la información de este sensor, y el resto de nodos del sistema que necesite esta información se suscribirá al correspondiente *topic* (Fig. 3.1).

Además de *topics* existen también otras utilidades como los servicios que utilizan un

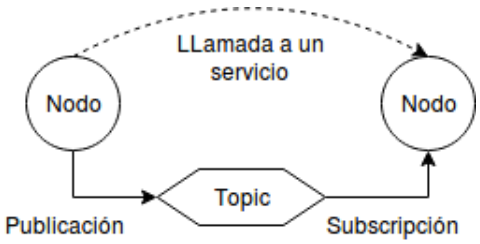


Figura 3.1: Esquema de comunicación de dos nodos a través de un topic y un servicio. Fuente: wiki.ros.org

modelo de comunicación de tipo solicitud-respuesta (*request/reply*) en vez el sistema de de publicación-subscripción. Estos servicios se definen a partir de un par de mensajes, uno de solicitud y otro de respuesta. Un ejemplo de servicio sería aquel que al recibir una solicitud activa o desactiva un sensor según el estado previo de éste, y responde adecuadamente si ha tenido éxito o no.

Todo estos conceptos adicionales pertenecen al nivel de grafo de computación de ROS, que es la red de procesos peer-to-peer que procesa todo el conjunto de datos.

3.1. Arquitectura del sistema

Los sistemas descritos en las siguientes secciones (planificador global, local y seguidor de caminos) pertenecen al subsistema de navegación del robot (Fig. 3.2).

Estos sistemas reciben información de la localización del robot, a partir de láseres y el sistema UWB (Ultra-WideBand System) que cotejan información con el mapa de costes o costmap que se describirá a continuación. Un nivel por encima se encuentra el MPMS, otro subsistema que es el encargado de gestionar el flujo del proceso de navegación para recolectar los diferentes materiales en el almacén para completar un pedido.

Finalmente en la arquitectura vemos otro subsistema que es el encargado de la detección y seguimiento de las personas en el entorno para tratarlas de forma distinta a los obstáculos inertes, este sistema queda fuera del alcance de este trabajo.

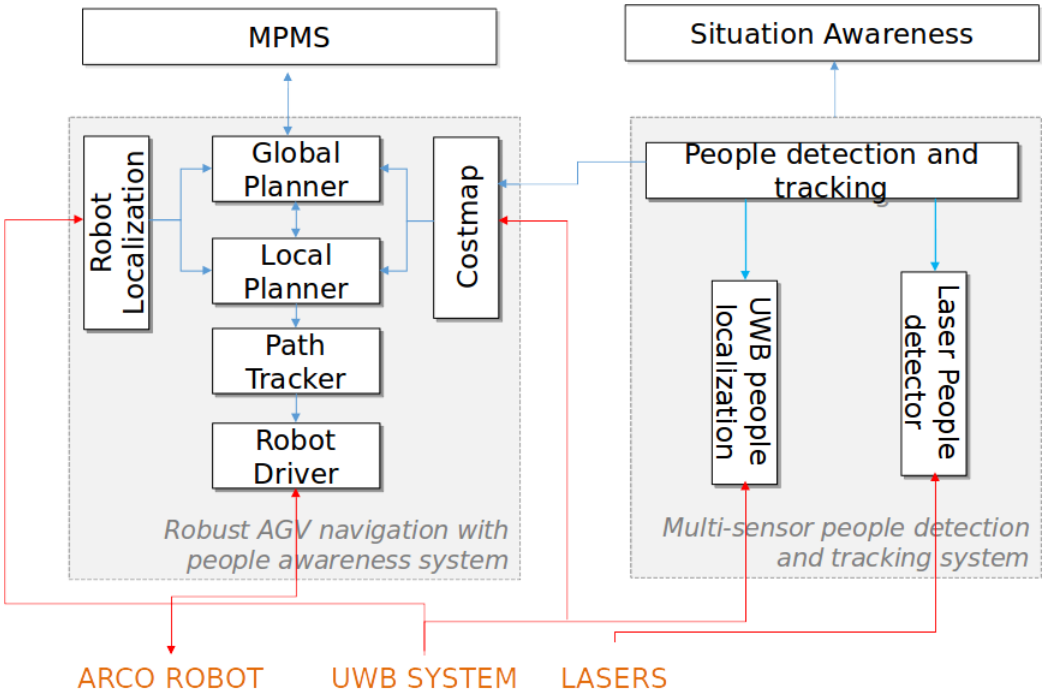


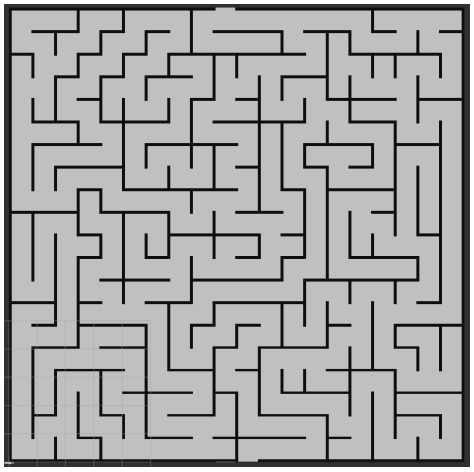
Figura 3.2: Arquitectura del sistema completo. Los planificadores y el seguidor de caminos o path-tracker pertenecen al subsistema de navegación del AGV

3.2. Planificador Global

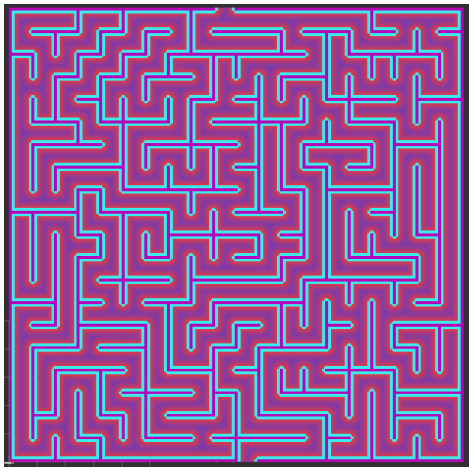
El planificador global consiste en un nodo de ROS que recibe un costmap global del módulo *Costmap 2d*[1] construido a partir del mapa publicado por el módulo *Map server*[2]. En la figura 3.3 puede verse un ejemplo visualizado de la información publicada por estos módulos.

El módulo *map server* publica un mensaje del tipo *occupancy grid* que no es más que una matriz donde 0 representa desocupado, -1 desconocido y 100 ocupado. El módulo costmap infla los alrededores de los obstáculos (aquellas casillas con valor 100) con valores intermedios entre 0 y 100 que decaen de forma exponencial de acuerdo a una serie de parámetros como son el radio de inflado y el factor de escala de costes.

[1] http://wiki.ros.org/costmap_2d
[2] wiki.ros.org/map_server



(a) Ejemplo de mapa publicado por el módulo map server



(b) Visualización de mapa de costes creado a por el costmap 2d

Figura 3.3: A la izquierda el negro representa 100 (ocupado) y gris 0 (libre). A la derecha el gradiente de colores de rojos a azules representa los valores intermedios entre 100 y 0. El color cyan corresponde a la distancia letal a los obstáculos,

El problema que surge aquí consiste en que el algoritmo detallado en el capitulo 2.3.4 no está diseñado para buscar en un costmap, ya que trabaja con información de nodos ocupados y desocupados. Es por ello que en la sección 4.2 se detallará una modificación realizada al algoritmo usado para que tenga en cuenta el costmap así como análisis de su rendimiento y eficacia.

El planificador global parte del costmap estático global y una serie de obstáculos cercanos detectados por láser. De esta forma, cuando se lanza un punto objetivo, el camino global devuelvo por el planificador tiene en cuenta el mapa y el entorno de ese instante. En la Sección 4 de experimentación se muestran algunos de los mapas estáticos usados para verificar el correcto funcionamiento del planificador sin introducir obstáculos a partir de sensores. Este mapa se actualiza a baja frecuencia, del orden de 1 Hz.

En la figura 3.4 puede verse un diagrama de flujo representativo del bucle de ejecución de este planificador. La salida de este módulo, el camino global, es una de las entradas del módulo de planificación local.

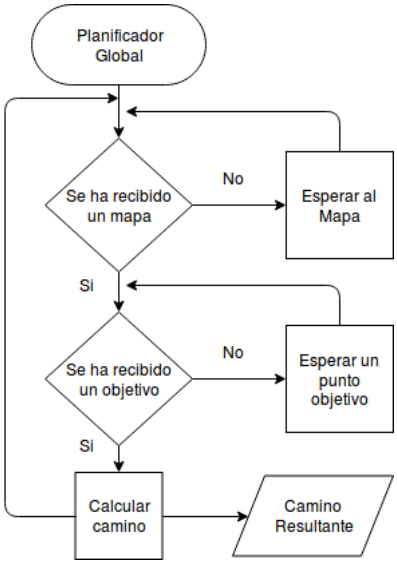


Figura 3.4: Diagrama de flujo del bucle de ejecución del planificador global

3.3. Planificador Local

Como se ha dicho anteriormente, este módulo recibe como entradas el camino global calculado por el planificador global, y el mapa local del entorno, un mapa dinámico centrado en el robot que se actualiza constantemente con la información de los láseres. Este planificador funciona a una frecuencia de entre 20 y 25 Hz, estando limitado por la frecuencia a la que se actualiza el mapa local, de forma que cada vez que recibe un nuevo mapa local, recalcula el camino local.

Para este planificador, el punto objetivo o *local goal* es el primer punto del camino global fuera de los límites del mapa local. El hecho de que se tome el primer punto fuera de los límites y no un punto interior al mapa hace que este punto objetivo siempre este desocupado ya que esta fuera del mapa local, de forma que si justo en el borde tenemos un obstáculo, el camino producido tenderá a rodearlo conforme el robot se acerque a este y todo el obstáculo entre en el mapa local, evitando problemas de puntos ocupados.

A medida que el robot avance, el planificador local irá recorriendo los puntos de la trayectoria global que caigan inmediatamente fuera del mapa local, hasta que el objetivo local coincida con el último punto de la trayectoria global, el objetivo global.

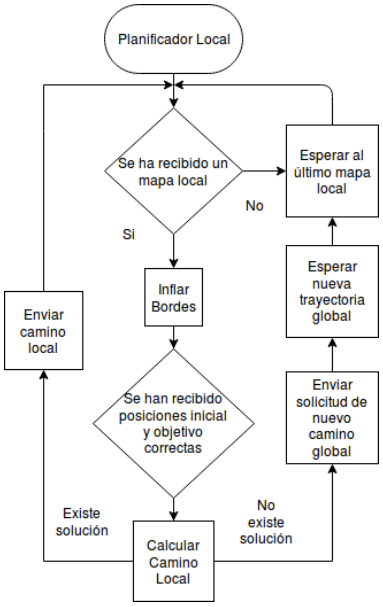


Figura 3.5: Diagrama de flujo del bucle de ejecución del planificador local

En la figura 3.5 se muestra el diagrama general de flujo del bucle principal del planificador local. Podemos ver, que para cada mapa recibido, el planificador local calcula y devuelve un camino. En ausencia de obstáculos, este devolverá continuamente el mismo camino, que coincidirá con una fracción del camino global devuelto por el planificador global.

El proceso de inflar bordes al que nos referimos aquí es independiente del inflado que realiza el módulo del costmap. Consiste en tomar el costmap local y añadir un borde de coste máximo excepto en un entorno del objetivo local, que tendrá coste cero. Esto permite que el algoritmo pueda tomar el primer punto fuera del costmap local "real" mencionado anteriormente como punto objetivo, y asegurarnos de que siempre este libre.

Si el espacio libre dejado alrededor de este objetivo local es insuficiente y hay por ejemplo un obstáculo grande delante que no permite al algoritmo llegar desde la posición del robot hasta el objetivo local, el planificador local mandará una solicitud al planificador global pidiendo una nueva trayectoria global que recorrer.

Esto asegura que durante la navegación el robot no se quede encerrado en un pasillo en el que se ha colocado un obstáculo dinámico impidiéndole cruzarlo.

3.4. Seguidor de Caminos

Este módulo recibe los puntos del camino local devuelto por el planificador local y en función de una serie de parámetros y restricciones cinemáticas del robot, envía comandos de velocidad para seguir los puntos, obteniéndose de esta forma una trayectoria en el dominio espacio temporal.

La particularidad de este sistema es que la trayectoria se obtiene en línea, es decir va cambiando según cambia el escenario local. Esto se realiza en condiciones de seguridad dada la alta frecuencia de re-planificación local (entre 20-23 Hz) de forma que el sistema reacciona rápidamente si aparece un obstáculo inesperado.

Aquí se presentará el planteamiento propuesto para el caso de un robot holónomo, es decir omnidireccional. Esto es posible gracias a las ruedas tipo Mecanum Wheels (Fig. 3.6). En la figura 3.7 se muestra como se obtiene el movimiento en x, y y de rotación.

El módulo de seguimiento de caminos o *path-tracker* envía comandos de velocidad,

Figura 3.6: Rueda tipo Mecanum Wheel. Este tipo de rueda permite un movimiento omnidireccional

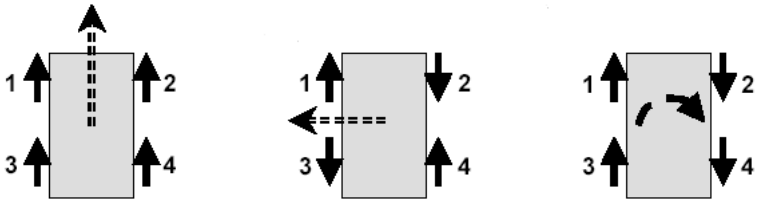


Figura 3.7: Principio de funcionamiento de una plataforma omnidireccional con cuatro ruedas Mecanum Wheel [14].

esto es, valores de V_x, V_y y W_z al driver, que de acuerdo con las ecuaciones de la cinemática de este sistema (En [15] podemos ver un desarrollo formal de esta cinemática) envía comandos de velocidad de rotación individual a cada rueda.

El sistema de seguimiento de puntos consiste en tomar los puntos devueltos por el planificador local y enviar comandos de velocidad en V_x y V_y dada una velocidad máxima V_{max} de la siguiente forma:

$$V_x = V_{max} \cos(\theta) \tag{3.1}$$

$$V_y = V_{max} \sin(\theta) \tag{3.2}$$

Donde el ángulo θ es aquel formado entre la dirección frontal del robot (en el sistema de referencia base-link, el eje x en sentido positivo) y la dirección hacia el punto considerado. A primera vista, esto conduciría a movimientos cualesquiera ya que el robot es libre de moverse en cualquier dirección por ser omnidireccional.

Sin embargo esto no siempre es beneficioso por diversas razones. Por un lado, por el principio de funcionamiento de las ruedas, que producen más deslizamiento al mover el sistema lateralmente (en dirección y) o en las rotaciones, lo que empeora la estimación de localización por odometría. Por otro lado, un robot que puede moverse de forma repentina en cualquier dirección resulta poco predecible y esto puede no ser lo más adecuado en un entorno dinámico con personas.

Es por ello que además de los comandos de velocidad en x e y, se mandan también comandos de rotación W_z, para de forma simultánea al movimiento de traslación omnidireccional, orientar el robot hacia el punto siguiente, de forma que el robot vaya "encarando" los puntos que debe seguir.

Finalmente, este módulo también se comunica con el planificador local para indicarle cuando se ha llegado al punto objetivo para que el planificador local pare de enviar caminos, y vuelva a esperar un camino global que seguir por parte del planificador global.

Como medida adicional de seguridad, se ha implementado un sistema de seguridad que para automáticamente el envío de comandos de velocidad si una medida del láser es menor o igual a una cierta distancia de seguridad, configurable. Esto permite evitar

colisiones en caso de que la replanificación no fuese lo suficientemente rápida. Este sistema funciona a la misma frecuencia que los láseres (40 Hz).

Únicamente cuando el obstáculo se ha alejado una cierta distancia (mayor que la primera, también configurable) el sistema continúa siguiendo el camino devuelto por el planificador local.

En el Capítulo 4 se mostrarán los resultados conjuntos al usar este sistema de seguimiento de caminos con los planificadores global y local descritos anteriormente.

Capítulo 4

Resultados y Discusión

Para validar el funcionamiento de los planificadores se han realizado tanto experimentos en simulación (para el caso del planificador global) como con una plataforma real omnidireccional para el caso de los dos planificadores funcionando simultáneamente con el seguidor de caminos.

En primer lugar se mostrarán los caminos devueltos por el planificador global en simulación para diferentes escenarios de pruebas y un breve análisis de su funcionamiento. En segundo lugar se mostrarán los resultados de modificar el algoritmo Lazy Theta* con optimizaciones para hacerlo sensible a un costmap como el descrito en la sección 3.2.

Por último se mostrarán los resultados del sistema completo en funcionamiento, planificadores global y local con algoritmo modificado y seguidor de caminos en un entorno industrial real.

4.1. Planificador global en simulación

Dado que este módulo de planificación solo requiere un mapa estático y las posiciones inicial y final del camino, puede evaluarse su funcionamiento simplemente por simulación.

Para verificar su correcto funcionamiento usaremos mapas de laberintos rectangula-

res obtenidos de MazeGenerator[1]

Este generador nos permite obtener laberintos con una única solución y variar la complejidad del laberinto mediante los siguientes dos parámetros:

- Factor R: Controla la tendencia elitista del algoritmo generador de laberintos. Un laberinto elitista es aquel con una solución corta respecto a las dimensiones de este mientras que uno no elitista tiene una solución que atraviesa una mayor parte del laberinto
- Factor E: Controla la tendencia a la ramificación. Un alto valor de este factor tiene pocas calles sin salida pero largas, mientras que con un valor bajo sucede lo contrario.

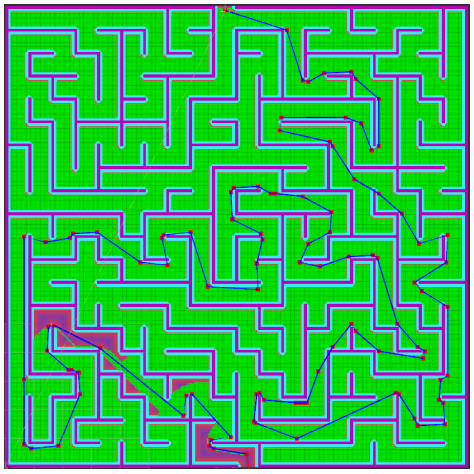
En las Figuras 4.1a y 4.1b se ilustra el efecto de usar valores extremos de estos parámetros y como afectan a la cantidad de nodos explorados y por tanto al tiempo de cálculo del camino.

En las Figuras 4.2a, 4.2b y 4.2c se ilustran casos de combinaciones de valores intermedios de estos parámetros. Visualmente puede apreciarse la diferencia entre los nodos explorados, aprenciándose casos en los para hallar la solución el algoritmo solo explora una mínima parte del mapa, como en la figura 4.2c mientras que en otros casos explora casi la totalidad de este (Fig. 4.1a.

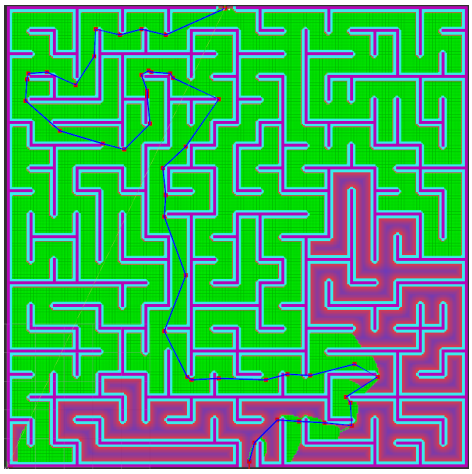
El objetivo aquí es poner a prueba el planificador global en laberintos con diferentes factores R y E para verificar el correcto funcionamiento de este. Además los tiempos y el número de nodos explorados visualmente deberán estar correlaciones con los valores de R y E en cierta medida. Además los laberintos se han explorado dando como punto inicial la salida superior y como punto final la inferior. Esto es importante para comparar resultados ya que estos pueden variar según de que punto se parta debido a la ausencia de simetría.

Estos mapas tipo laberinto resultan útiles para hacer análisis iniciales del comportamiento del algoritmo. Sin embargo en la Sección 4.2 se usará un mapa real para poner a prueba el algoritmo modificado con el objetivo de que los resultados ilustren mejor el comportamiento en un entorno realista.

[1] `http://www.mazegenerator.net/` : Generador de laberintos online

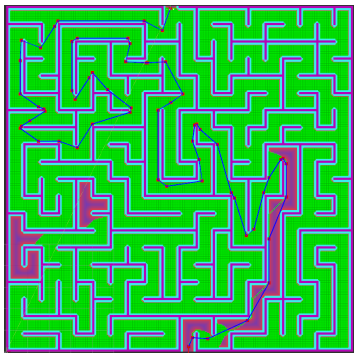


(a) $E = 0$, $R = 0$, $L = 98{,}4m$, $t = 710ms$

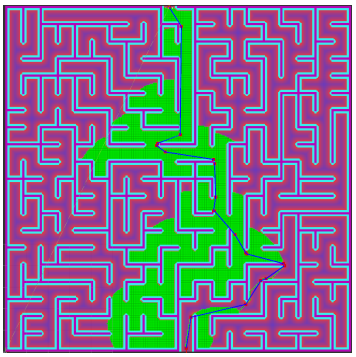


(b) $E = 100$, $R = 100$, $L = 44{,}1m$, $t = 595ms$

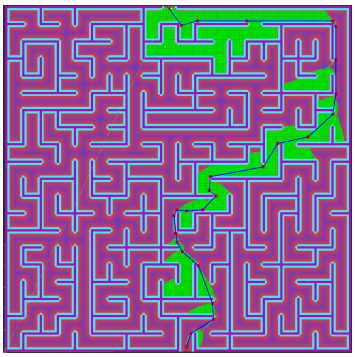
Figura 4.1: A la izquierda un laberinto con muchas calles sin salida cortas y una solución muy larga respecto a sus dimensiones. Puede verse como el algoritmo necesita explorar casi la totalidad del laberinto. A la derecha, el caso opuesto, al haber menos calles sin salida y ser la solución más corta respecto a las dimensiones del laberinto, el algoritmo necesita explorar menos nodos.



(a) $E = 0$, $R = 50$, $L = 64{,}7m$, $t = 714ms$



(b) $E = 50$, $R = 0$, $L = 23{,}7m$, $t = 195ms$



(c) $E = 50$, $R = 50$, $L = 29.m$, $t = 100ms$

Figura 4.2: A la izquierda laberinto poco ramificado pero con solución muy larga, en el centro, lo contrario y a la derecha un caso intermedio. Puede verse como en estos dos últimos casos el algoritmo solo necesita explorar una pequeña fracción del laberinto para encontrar la solución.

4.2. Lazy Theta* sensible a los costes

Como ya se explicó anteriormente, esta modificación esta motivada por el uso de un costmap, ya que por defecto el algoritmo solo distingue entre nodos ocupados y no ocupados. La motivación del uso del costmap es la de obtener caminos más seguros, entendiéndose como caminos seguros aquellos que que pasan por los puntos más alejados de los obstáculos en cada momento, por ejemplo por el centro de un pasillo.

Además de obtener caminos más seguros, un camino que discurra por las zonas de mínimo coste posible aumenta también la suavidad de este. Como hemos visto en los ejemplos anteriores, los caminos devueltos por el algoritmo sin tener en cuenta costes son bruscos en cuanto al ángulo que forman los puntos entre ellos. Esta suavidad resulta muy necesaria en aplicaciones como la navegación autónoma.

La solución propuesta consiste en modificar la función $g(s)$, la cual originalmente es la distancia del nodo s al nodo inicial por el camino calculado hasta el momento al nodo s.

Esta modificación afecta a la forma de calcular distancias entre nodos, añadiendo un termino aditivo proporcional al coste del nodo siguiente. Es decir, cuando el algoritmo este situado en un nodo y calcule la distancia a los vecinos más próximos, esta se verá distorsionada, favoreciendo a los vecinos de coste menor frente a los de coste mayor.

Dados 3 nodos, s, s' y $p(s)$ el padre de s:

$$D(s, s') = C_w \cdot c(s') + \sqrt{(s_x - s'_x)^2 + (s_y - s'_y)^2} \tag{4.1}$$

$$D(start, s) = D(p(s), start) + D(p(s), s) \tag{4.2}$$

Con *start* el nodo de partida y $D(p(s), start)$ calculada de forma acumulativa a lo largo del camino desde *start* a $p(s)$ mediante la ec. 4.1. Esto hace que de forma acumulada la nueva distancia al punto inicial se vea afectada favoreciendo caminos por zonas de menor coste.

Por otro lado, para que ésta modificación surta efecto, debe modificarse también la distancia de linea de vista. Esto se debe a que el propio algoritmo tiende a unir los

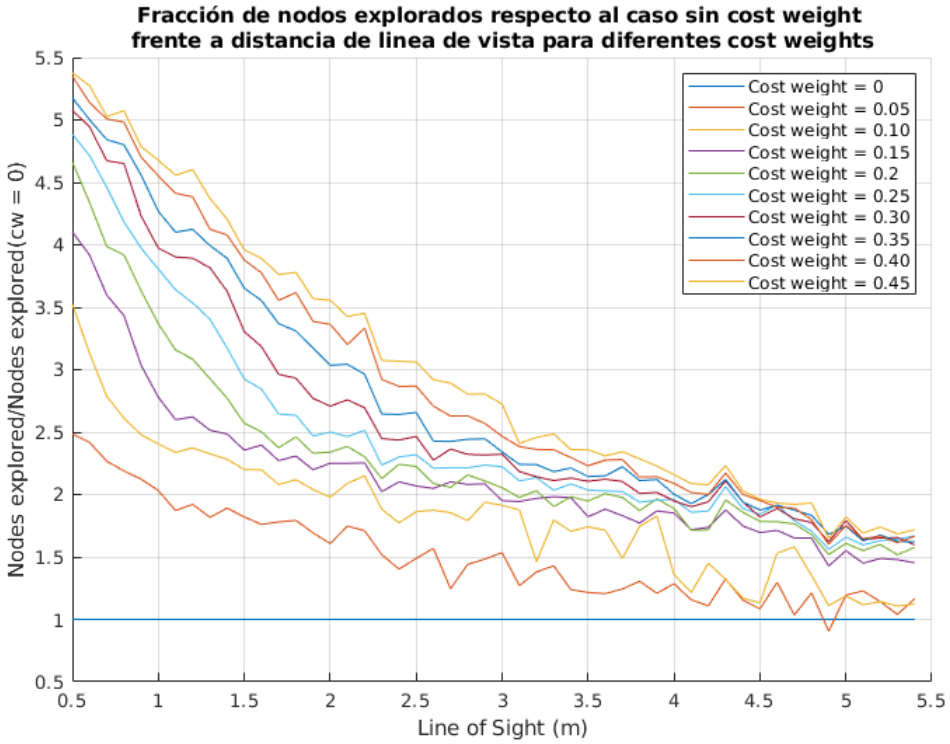


Figura 4.3: Comparativa entre número de nodos explorados respecto al caso sin cost weight(algoritmo sin modificar) frente a distancia de línea de vista para diferentes valores de cost weights.

nodos que tienen linea de vista con los padres de otros nodos, de forma que aunque se fuerce al algoritmo a buscar caminos por las zonas de menor coste alterando la distancia euclídea entre ellos, si hay linea de vista entre un nodo y el padre de otro, los unirá y el hecho de haber tenido en cuenta los costes no habrá servido de nada.

A continuación se muestra un breve análisis del resultado de estas modificaciones. Los resultados presentados a continuación se han realizado bajo el mismo sistema con el planificador global en simulación, con un mapa de la planta de Tintas Robbialac S.A. con sede en Lisboa. En todos los ensayos se han usado los mismos puntos como inicial y final. En las figuras 4.5 y 4.6 pueden visualizarse algunos caminos obtenidos con parámetros representativos. En todos ellos el punto inicial es aquel de la esquina inferior derecha y el punto final el más cercano a la esquina superior derecha del mapa.

En las Figura 4.3 se muestra la proporción entre el número de nodos explorados con

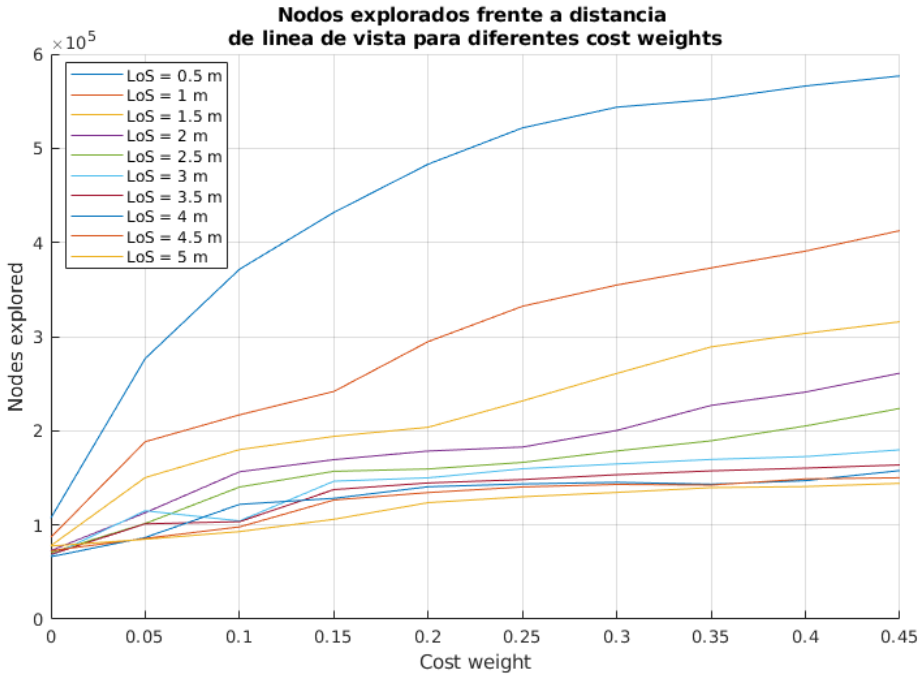
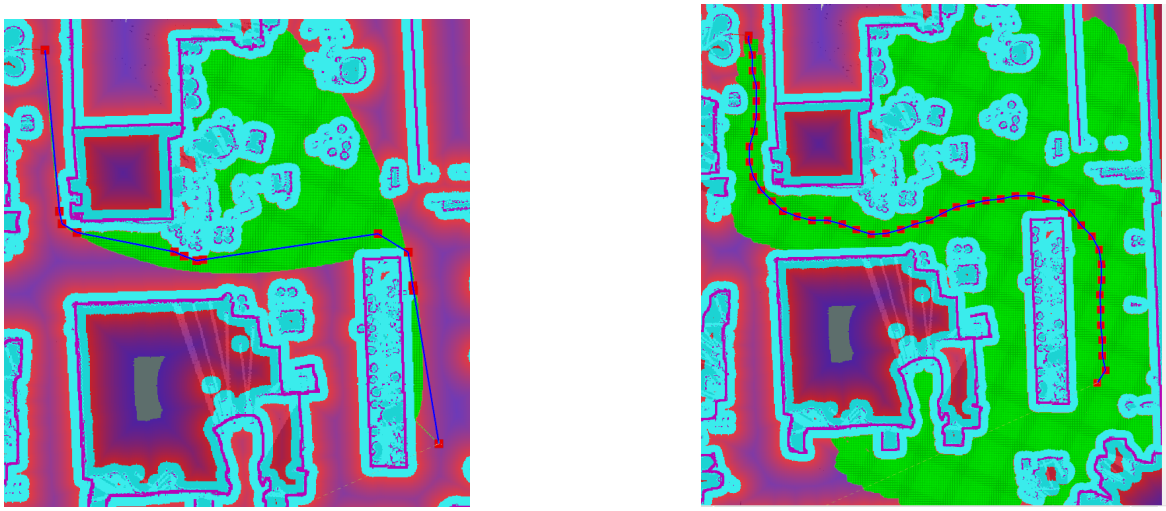


Figura 4.4: Comparativa de número de nodos explorados frente a cost weight para diferentes distancias de línea de vista.

el algoritmo modificado y sin modificar, para diferentes valores del parámetro cost weight en función de la restricción de línea de vista. Como era de esperar, al aumentar la línea de vista, el cost weight deja de tener efecto por lo que se explicó anteriormente. Además se observa como al aumentar el cost weight para una misma línea de vista, aumentan los nodos explorados, esto puede deberse a que cuanto más grande es éste, mas se distorsiona el mapa que ve el algoritmo. En la Figura 4.4 se muestran datos análogos, pero usando como variable independiente el cost weight. Puede observarse como los nodos explorados dependen no linealmente de la restricción de línea de vista.

Por un lado, tal y como era de esperar, la modificación realizada reduce la eficiencia del algoritmo, aumentando la cantidad de nodos explorados para el mismo caso. Como se ve en la figura 4.5a este efecto se reduce al aumentar la línea de vista. Sin embargo este aumento de la línea de vista, como ya se ha dicho, reduce la efectividad de tener en cuenta los costes. En la figura 4.5b puede verse como efectivamente esta modificación conduce a caminos más suaves ,aunque ligeramente más largos, que respecto al caso sin modificar (Fig. 4.5a). Puede observarse también que prácticamente



(a) $L = 41m$, 62000 nodos explorados

(b) $L = 43{,}3m$, 320000 nodos explorados

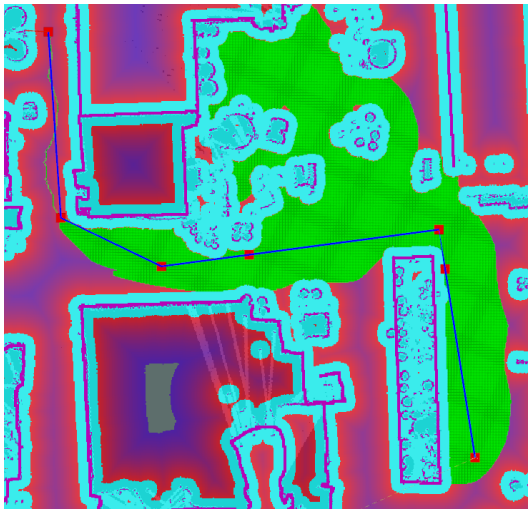
Figura 4.5: A la izquierda, camino resultante sin modificar el algoritmo, a la derecha, algoritmo con parámetros cost weight = 0.25 y linea de vista = 1 m.

independientemente del valor del cost weight, al aumentar la línea de vista, el número de nodos explorados converge al caso con cost weight cero. Esto refleja la necesidad de restringir la línea de vista.

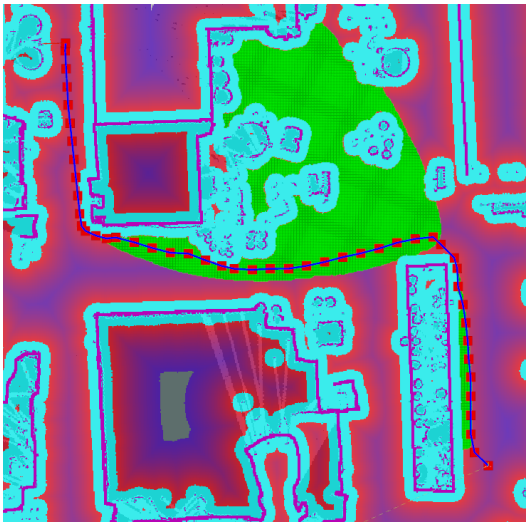
Por otro lado vemos como el número de nodos explorados es creciente con el cost weight y decreciente con la distancia de línea de vista (Fig. 4.5b) como era de esperar. Estos análisis permiten acotar las posibilidades en la elección del valor de estos parámetros según la geometría del escenario y otros requerimientos como el tiempo de ejecución, que esta relacionado directamente con el número de nodos explorados.

En las figuras 4.5 y 4.6 podemos ver una serie de caminos ilustrativos de los efectos de los parámetros estudiados anteriormente. Los puntos verdes corresponden a los nodos explorados, los puntos rojos a los nodos que pertenecen al camino resultante. Las zonas de color cyan representan puntos ocupados para el algoritmo, las zonas más cercanas a estas(rojas) son las de mayor coste.

Puede verse como claramente el camino más suave (Fig. 4.5b) es aquel que explora más nodos. También en la figura 4.6a puede verse como el parámetro cost weight resulta inútil sin la restricción de línea de vista y viceversa en la figura 4.6b.



(a) $L = 42{,}3m$, 82000 nodos explorados



(b) $L = 41{,}5$, 84000 nodos explorados

Figura 4.6: Comparativa entre efectos separados de los factores cost weight y linea de vista. A la izquierda, sin restricción de línea de vista pero con cost weight = 0.25, a la derecha, linea de vista = 1 m con cost weight = 0.

4.3. Planificadores y Lazy Theta* sensible a los costes sobre una plataforma robótica en un escenario industrial real

Para ilustrar los resultados globales del funcionamiento del sistema se incluyen dos vídeos como contenido audiovisual extra. Estos también pueden encontrarse online en los enlaces vídeo 1 [2] y vídeo 2 [3]. En el primer vídeo puede verse la plataforma de pruebas utilizadas en un escenario simulado, mientras que en el segundo se muestra la plataforma final en el escenario real.

En ambos vídeos se muestra una grabación del robot final (Fig. 4.8) en operación y se superpone una visualización de RViz, el visualizador de datos de ROS. En la Figura 4.7 puede verse un ejemplo de esto. Se observa el costmap local rodeando y siguiendo al robot y moviéndose con el, superpuesto al costmap global creado a partir del mapa

[2] `https://youtu.be/aTMpI3Jwg5M`
[3] `https://youtu.be/JZP_gAS-B6o`

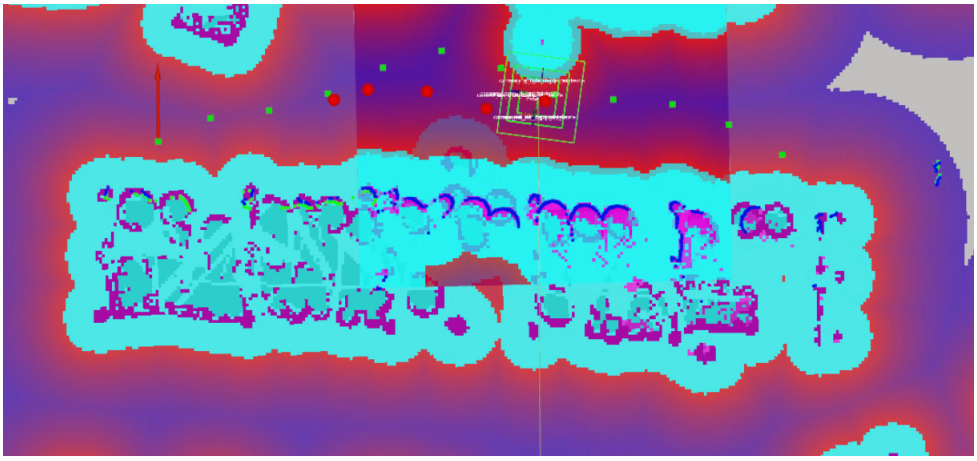


Figura 4.7: Ejemplo de visualización del estado del robot en tiempo real mediante RViz. El cuadrado verde representa la proyección de la superficie del robot sobre el suelo.

estático. En verde se muestra el camino global y el rojo el camino local que se recalcula continuamente como puede verse en los vídeos mencionados anteriormente. Las zonas de color cyan son aquellas definidas como letales en el costmap, tal que si el robot entrase en ellas, correría riesgo de colisión, por lo que el planificador ve esas zonas como zonas ocupadas y nunca planificará en ellas.

En el primer vídeo se muestra la plataforma usada para las pruebas en un escenario simulado. Aquí la plataforma se mueve en el modo más simple de navegación, el modo no omnidireccional. Puede verse como realiza una misión simulada, acudiendo a puntos de recogida de materiales y transportándolos secuencialmente a un tanque de mezcla simulado, esquivando a lo largo del camino obstáculos y personas.

En el segundo vídeo se muestra el sistema en un estado más avanzado, puede apreciarse como en este punto, el sistema se mueve de forma omnidireccional, intentando en la medida de lo posible ir de cara hacia los puntos objetivos, por motivos de seguridad y por el hecho de resultar más predecible.

Puede verse al robot final con la geometría final diseñada para transportar entre otros materiales, barriles de productos químicos. De nuevo aquí se mandan una serie de puntos objetivos que en RViz aparecen como flechas rojas y el robot se desplaza a lo largo de ellos, esquivando los obstáculos estáticos y dinámicos que aparecen en su camino.

Figura 4.8: Plataforma robótica final utilizada para el transporte de materiales durante el proceso de producción de pinturas en Tintas Robbialac S.A. Debajo de la superficie metálica plana tiene una báscula que permite controlar la carga que lleva y adaptar el movimiento a esta.

En ambos casos se consiguió el objetivo deseado, planificar y ejecutar un camino libre de obstáculos entre la posición del robot y el punto objetivo por lo que el resultado se considera satisfactorio.

Capítulo 5

Conclusiones

En primer lugar, dados los resultados mostrados en la Capítulo 4, podemos decir que el desarrollo y la implementación del sistema ha sido exitosa. Si bien los métodos heurísticos usados pueden no haber sido los más óptimos debido a la propia naturaleza de los algoritmos heurísticos, estos han permitido cumplir los objetivos propuestos, es decir, un sistema autónomo que planifique a alta frecuencia y desarrolle trayectorias evitando exitosamente colisiones en entornos industriales dinámicos.

En segundo lugar y más concretamente, se ha verificado el correcto funcionamiento y la mejora cualitativa obtenida modificando el algoritmo Lazy Theta* con optimizaciones, para hacerlo sensible a un mapa de costes, favoreciendo la planificación y por tanto las trayectorias más alejadas de los obstáculos pero sin restringir el movimiento cerca de estos cuando no hay otra opción.

Sin embargo, habría que realizar un análisis más exhaustivo de esta modificación del algoritmo, para saber en que grado es sub-óptimo para así desarrollar futuras estrategias que permitan mejorarlo. Otro paso que podría resultar natural, sería extender este algoritmo a la planificación de trayectorias, no solo a caminos.

Esto último no se ha intentado para esta aplicación dado el aumento esperable en los tiempos de planificación, con la consecuente reducción de la frecuencia de re-planificación local, lo que repercutiría negativamente en la seguridad durante la navegación.

Finalmente, también es importante hablar del papel de estos sistemas autónomos en

el futuro de la industria y sociedad. Existen posturas tanto a favor como en contra de la tendencia a la automatización que existe actualmente. Por un lado, los detractores argumentan que favorece la destrucción de empleos ya que la automatización permite realizar las tareas menos cualificadas y ahorrar en mano de obra.

Por otro lado, otros autores opinan que la automatización confiere una ventaja estratégica a las empresas, permitiéndoles ser más eficientes y aumentar la productividad, lo que las vuelve más competitivas y favorece su crecimiento, acabando con un balance de generación de empleo positivo.

Esto hace que actualmente este tipo de tecnologías no sean bien recibidas en determinados sectores de la sociedad. La solución a esto es complicada e implica a múltiples actores: industria, estado y laboratorios de investigación entre otros. Un primer paso podría ser desarrollar sistemas de co-working como el aquí presentado, que mejoren las condiciones de trabajo de los operarios, sin reemplazarlos totalmente y permitir así una transición suave hacia una automatización final más completa.

Bibliografía

[1] IFR. Service robots - definition and classification. https://www.ifr.org/service-robots/, 2016.

[2] Steven M. Lavalle. *Planning Algorithms*. Cambridge University Press, 2006.

[3] Nikolas Correl. *Introduction to Autonomous Robots, v1.7*. Magellan Scientific, 2016.

[4] J.A. Cobano F. Caballero L. Merino R. Rey, M. Conflan. Human-robot co-working system for industry 4.0 warehouse automation. In *Proceedings of the 24th IEEE Conference on Emerging Technologies and Factory Automation*, 2019.

[5] L. E. Kavraki, P. Svestka, J. . Latombe, and M. H. Overmars. Probabilistic roadmaps for path planning in high-dimensional configuration spaces. *IEEE Transactions on Robotics and Automation*, 12(4):566–580, Aug 1996.

[6] Roland Geraerts and Mark H. Overmars. *A Comparative Study of Probabilistic Roadmap Planners*, pages 43–57. Springer Berlin Heidelberg, Berlin, Heidelberg, 2004.

[7] Steven M. Lavalle. Rapidly-exploring random trees: A new tool for path planning. Technical report, Iowa State University, 1998.

[8] Seth Hutchinson George Kantor Wolfram Burgard Lydia Kavraki Sebastian Thrun Howie Choset, Kevin Lynch. *Principles of Robot Motion: Theory, Algorithms, and Implementation*. The MIT Press, 2005.

[9] S. Arimoto H. Noborio, S. Wazumi. An implicit approach for a mobile robot running on a force field without generation of local minima. *IFAC Proceedings Volumes*, 23:87–92, Aug 1990.

[10] E.W. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1:269–271, 1959.

[11] Nilsson N. J. Raphael B. Hart, P. E. A formal basis for the heuristic determination of minimum cost paths. *IEEE Transactions on Systems Science and Cybernetics*, SSC-4(2), Jul 1968.

[12] Richard E. Korf. Recent progress in the design and analysis of admissible heuristic functions. In Berthe Y. Choueiry and Toby Walsh, editors, *Abstraction, Reformulation, and Approximation*, pages 45–55, Berlin, Heidelberg, 2000. Springer Berlin Heidelberg.

[13] S. Koenig K. Daniel, A. Nash and A. Felner. Theta*: Any-angle path planning on grids. *Journal of Artificial Intelligence Research*, 39:533–579, 2010.

[14] Thomas Bräunl. *Embedded Robotics*. Springer, 2006.

[15] Hamid Taheri, Bing Qiao, and Nurallah Ghaeminezhad. Kinematic model of a four mecanum wheeled mobile robot. *International Journal of Computer Applications*, 113(3):6–9, March 2015.