UNIVERSIDAD REY JUAN CARLOS

Ingeniería Técnica en Informática de Sistemas

Escuela Superior de Ciencias Experimentales y Tecnología

Curso académico 2004-2005

Proyecto Fin de Carrera

Navegación global utilizando el método del grafo de visibilidad.

Tutores: José M. Cañas Plaza, Francisco Martín Rico.

Autor: Alejandro López Romero.

Febrero 2.005

Índice general

Capítulo 1

Introducción

La robótica, considerada como rama de la tecnología, surgió como consecuencia de las necesidades del ser humano con su entorno. Estas necesidades, junto con los conocimientos teóricos desarrollados y la tecnología disponible, se fueron resolviendo aplicando en muchos casos conocimientos de diferentes disciplinas como la matemática, mecánica, electrónica, automática,... etc. La historia de la robótica ha estado unida a la construcción de artefactos, muchas veces por obra de genios autodidactas, que trataban de materializar el deseo humano de crear seres que nos descargasen del trabajo.

Los cambios en el campo de la robótica se han sucedido tan deprisa en los últimos años, que puede hablarse de la existencia de varias generaciones desde sus orígenes hasta la actualidad. La primera generación surgió a principios de los años 60, y la constituyeron principalmente los robots manipuladores. Estos sólo podían realizar movimientos repetitivos, asistidos por sensores internos que les permitían ejecutar ciertos movimientos con precisión. La segunda generación de robots entra en escena a finales de los años 70, e incorpora nuevos tipos de sensores (de proximidad y visión por lo general) que proporcionan al robot información del mundo exterior. Esta clase de robots son capaces de tomar decisiones simples y reaccionar ante el entorno de trabajo. La tercera generación, surgida en los últimos años, emplea la inteligencia artificial y potentes ordenadores para resolver problemas complejos e interpretar información procedente de sensores avanzados. Todos estos progresos han permitido ampliar los campos de aplicación de la robótica respecto a los de su interés inicial, centrado principalmente en aplicaciones industriales.

Los robots son usados hoy en día para llevar a cabo tareas que son demasiado sucias, peligrosas, difíciles o repetitivas para los humanos. Esto usualmente toma la forma de un robot industrial usado en las líneas de producción. La industria de automoción ha sido impulsada por esta nueva tecnología donde los robots se programan para reemplazar el trabajo de los humanos en muchas tareas repetitivas. Otras aplicaciones incluyen la limpieza de residuos tóxicos, exploración espacial, minería y localización de minas terrestres. La manufactura continúa siendo el principal mercado donde los robots son utilizados. En particular, robots articulados (similares en capacidad de movimiento a

un brazo humano) son los más usados comunmente. Las aplicaciones incluyen pintado, carga de maquinaria y soldado, como el robot de la figura 1.1(a).

Respecto a la exploración espacial hemos sido testigos de la llegada de robots a Marte como la Rover Sojourner (figura 1.1(b)) en 1997 a bordo de la nave Mars Pathfinder. Este robot realizó diversos análisis geológicos sobre las rocas y envió un total de 550 fotografías durante sus 83 días de operación en la superficie.[1]

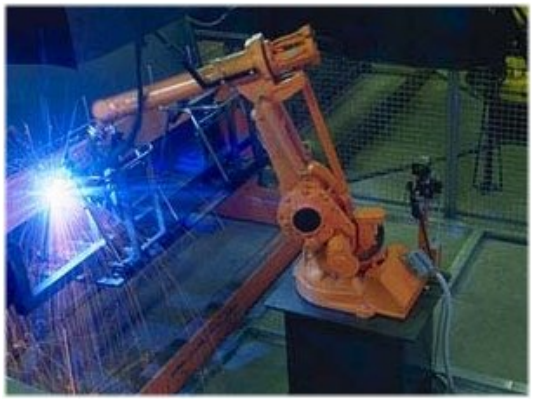
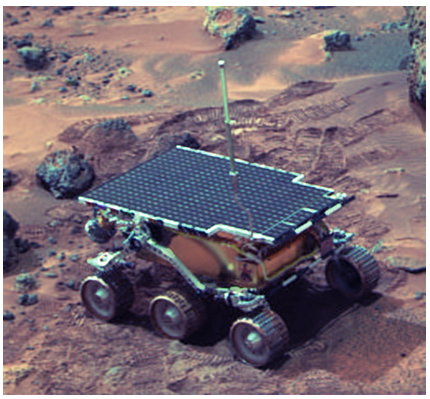
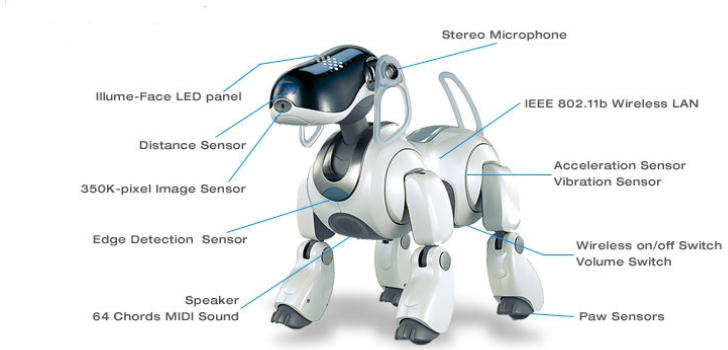
(a)  (b) 

Figura 1.1: Robot industrial de soldadura (a) y robot Sojourner Rover en Marte (b)

Existe una corriente dentro de la robótica que pretende introducir pequeños robots dentro del hogar. Un par de ejemplos actuales en esa dirección, son el robot aspiradora *Roomba* (figura 1.2(a)), que es capaz de limpiar la suciedad del suelo, o el robot mascota *Aibo* (figura 1.2(b)) que según su empresa fabricante Sony "*..verdaderamente tiene emociones e instintos programados en su cerebro.*"[2].

(a) 

(b) 

Figura 1.2: Robot mascota Aibo de Sony y robot de limpieza Roomba

[1] http://marsrovers.jpl.nasa.gov
[2] http://www.sony.net/Products/aibo

1.1. Navegación

En todos los casos indicados en el apartado anterior existe el concepto de desplazamiento, excepto en el brazo mecánico. Cuando existe un desplazamiento de todo el cuerpo del robot es necesario utilizar técnicas de navegación que le permitan moverse por su entorno de un modo autónomo y sin colisionar con los obstáculos.

Un problema muy común al que se enfrenta un sistema móvil con un cierto grado de autonomía es el de navegar por un entorno evitando sus obstáculos hasta alcanzar una meta. Actualmente existen sistemas capaces de navegar por entornos conocidos como industrias, hospitales y oficinas, que son capaces de trasladar material o documentos entre distintos puntos de un edificio. Como ejemplo de uno estos sistemas están los robots *Minerva* (figura 1.3) o *Rhino*, diseñados para guiado autónomo de visitantes a museos [3].

Figura 1.3: Robot guía de museos Minerva

Los sistemas de navegación que contienen un mapa completo de los obstáculos que componen su entorno se engloban dentro del campo de la *navegación global*. En estos sistemas es posible la planificación de una ruta óptima entre diferentes lugares evitando la colisión con los obstáculos del mapa, dado que se conoce de inicio su situación exacta.

Existen distintas técnicas de navegación global como los diagramas de Voronoi [L.Moreno, 2004], que utilizan los espacios que surgen al dejar los obstáculos a ambos lados a una misma distancia. Otra técnica es la del gradiente [Isado, 2005], que asocia a cada punto del espacio un valor relacionado con su cercanía a los obstáculos y su proximidad al punto objetivo teniendo en cuenta paredes, puertas abiertas ... etc. Existe también la técnica de navegación global del grafo de visibilidad, que utiliza los caminos que se forman al unir los vértices de los obstáculos con la condición de que no atraviesen ninguno de ellos a su paso.

Un problema asociado a la navegación global es la localización, para cuya resolución se han propuesto gran número de enfoques y técnicas. Para tener localizada la posición exacta de un robot durante la navegación no es suficiente con medir el espacio recorrido, ya que durante el desplazamiento se pueden acumular errores de los sensores y dar

[3] http://www.cs.washington.edu/ai

medidas erróneas sobre su posición.

Algunas de las técnicas para solucionar este problema son por ejemplo, la localización probabilística con visión local [Dueñas, 2004] en la que una serie de algoritmos calculan las posibles localizaciones del robot mediante los datos recopilados por los sensores. Otra técnica es la de localización mediante redes inalámbricas [Serrano, 2004] que utiliza señales de radio junto con modelos de Markov para determinar la posición del robot en el mapa.

Otro tipo de navegación es el que se realiza en entornos con obstáculos móviles como pueden ser personas, vehículos u otros robots. Para poder evitar estos obstáculos es necesaria la utilización de sensores como el láser, sónar, cámaras de video ..etc. Con estos sensores podemos determinar las distintas posiciones de los obstáculos en el tiempo y evaluar las posibles trayectorias por si existiera peligro de choque con el robot. En este caso habría que modificar la ruta del para evitar la colisión. Este método , consistente en la utilización de los sensores del robot para determinar los obstáculos situados a su alcance, se denomina *navegación local*.

Una de las técnicas utilizadas para la navegación local es la de los campos de fuerza virtuales VFF o *Virtual Force Fields* [Palomino, 2004]. En esta técnica el robot es atraído por la posición objetivo mientras que los obstáculos ejercen una fuerza repulsiva que hace que se aleje de ellos. Otra técnica es la ventana dinámica o *Dinamic Window* [Lobato, 2003], cuya idea principal es predecir todas las posibles velocidades alcanzables en el siguiente intervalo de tiempo, de manera que no se produzca ninguna colisión. Dichas velocidades están enmarcadas por unos límites físicos inherentes a la aceleración máxima del robot. Una vez calculadas todas, maximizamos una función, que nos devuelve la mejor de estas velocidades para ese instante. De esta manera, iterativamente, se obtiene un comportamiento reactivo, capaz de reaccionar ante cambios en el medio.

1.2. Navegación global mediante grafo de visibilidad

Este trabajo se encuadra dentro de un conjunto de proyectos que han abordado el problema de la navegación y localización en robots móviles de interiores, algunos de ellos reseñados en la anterior sección.

El proyecto programa un robot para que entienda el mapa de su entorno especificado en un fichero. Con este mapa debe ser capaz de planificar una ruta entre dos puntos de este entorno y de pilotar el robot por ella. Para ello implementa el grafo de visibilidad y una técnica clásica de planificación de caminos. También se implementan las órdenes necesarias para que el robot recorra el camino planificado anteriormente.

Utilizamos la técnica de navegación global mediante grafo de visibilidad con el robot Pioneer 3, que se muestra en la figura 1.4. Está técnica hace que el robot pueda planificar rutas utilizando los caminos libres de colisión que se forman entre los vértices de los obstáculos. En la ficha web de este proyecto [4] se puede descargar el código, la memoria y la documentación generada durante su realización.

[4] http://gsyc.escet.urjc.es/robotica/pfc/pfc-grafovisibilidad.html

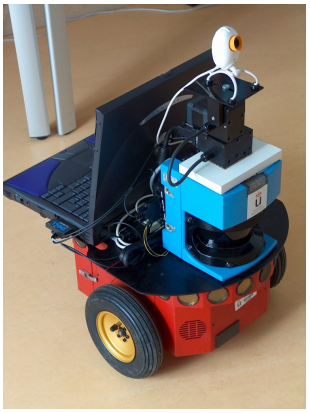


Figura 1.4: Robot Pioneer de la URJC

Esta técnica se programa sobre una plataforma llamada JDE (Jerarquía Dinámica de Esquemas) que se ha desarrolllado íntegramente en el grupo. Esta arquitectura de control se basa en pequeños esquemas o tareas. Cada esquema realiza una acción muy determinada y la unión de varios esquemas dan lugar a un comportamiento complejo.

La memoria se articula en 6 capítulos de los cuales el actual es el capítulo introductorio. En el capítulo 2 se especifican los objetivos y requisitos del proyecto. En el 3 la plataforma software y hardware sobre el que está construido. El capítulo 4 detalla la implementación informática del proyecto, de cuya experimentación hablamos en el capítulo 5. Finalmente el capítulo 6 corresponderá a las conclusiones y lineas de trabajo futuras.

m

Capítulo 2

Objetivos y Metodología

Una vez presentado el contexto de éste proyecto en el capítulo introductorio, se describen aquí los objetivos concretos, requisitos y la metodología empleada durante la creación de este trabajo.

El objetivo principal de este proyecto es la implementación informática de una técnica de navegación global basada en posición. En partícular utilizando un grafo de visibilidad sobre un mapa del entorno del robot. Se utilizará para ello la arquitectura de control JDE desarrollada por el grupo de robótica de la URJC que se describirá en detalle en el capítulo 3.

2.1. Objetivos

El objetivo global que se persigue en este proyecto es la navegación del robot Pioneer por un entorno conocido sin colisionar con ningún obstáculo.

Este objetivo se puede dividir en tres etapas o subobjetivos. Estos son la creación del grafo de visibilidad dado un mapa del entorno, la planificación de la ruta a partir del grafo de visibilidad, y por último la ejecución del plan para que el robot alcance la meta propuesta.

Cada uno de los puntos anteriores contiene como tarea inicial la documentación y puesta al día de las técnicas e investigaciones que se han realizado para este tipo de proyectos. A continuación se detallan cada uno de los tres subobjetivos indicados anteriormente :

1. Creación del grafo de visibilidad :

   Inicialmente disponemos de un mapa completo del entorno en el cual únicamente existirán obstáculos fijos, como son paredes, muebles, puertas, etc. Este mapa ha podido ser creado por el diseñador, o puede haber sido generado por una fase anterior en la que el robot navegara aleatoriamente por el entorno sin llegar a tocar los elementos. En su recorrido habría almacenado por medio de sus sensores la situación exacta de los obstáculos, de los que se asume que permanecerán inmóviles indefinidamente.

Con este mapa, que en este proyecto se traduce como una entrada necesaria y creada previamente, creamos el grafo de visibilidad correspondiente. Éste nos muestra los posibles caminos que el robot podría tomar siempre en línea recta y utilizando los vértices que forman los obstáculos entre sí a una distancia necesaria para que el robot no colisione con ellos.

2. Planificación de la ruta :

   El siguiente subobjetivo consiste en utilizar estos grafos de visibilidad para construir una ruta entre el robot y un destino marcado en el mapa. El destino lo debe poder definir el usuario picando con el ratón sobre un punto en el mapa que se muestra en la interfaz gráfica. Debe ser un destino accesible por el robot, ya que de lo contrario no se podrá generar la ruta hasta éste, como por ejemplo habitaciones cerradas o patios interiores. Un destino se considera accesible cuando el robot puede trazar una ruta recorriendo el grafo de visibilidad sin colisionar con los obstáculos que forman el entorno. Sobre el mismo grafo se deben poder especificar tantos destinos como se quiera a lo largo del tiempo.

   Es importante destacar que planificar una ruta entre dos puntos de un mapa puede ser a veces imposible, esto a debido a indicar la meta en habitaciones cerradas o a situar el punto destino en un punto fuera de los límites del mapa. No se persigue que el robot evite obstáculos imprevistos ya que no usaremos información sensorial, salvo la de los odómetros que nos darán la posición del robot.

   La planificación de la ruta utilizando el grafo debe ofrecer un camino válido para alcanzar la meta propuesta. Se entiende por camino válido aquel que sin tener que ofrecer obligatoriamente el camino más corto, no realice demasiados desvíos innecesarios durante su trayectoria.

3. Desplazamiento del robot :

   Utilizando como entrada la ruta generada en el paso anterior, el robot se desplaza siguiendo las coordenadas que marcan los nodos del nuevo camino siempre a una distancia segura de los obstáculos. Tras finalizar el recorrido el robot debe esperar una nueva meta para volver a calcular una ruta hacia ella e iniciar un nuevo desplazamiento.

2.2. Requisitos

Las tres etapas comentadas anteriormente conforman un único objetivo global y se desarrollan teniendo en cuenta los siguientes requisitos que condicionan su implementación :

1. El algoritmo de navegación debe estar orientado al robot **Pioneer 3** de *ActivMedia*, en concreto a su simulador el **SRIsim** de libre distribución. El Pioneer se considera un robot idóneo para la navegación por interiores y oficinas por maniobrabilidad y la cantidad de sensores y actuadores de los que dispone.

Es necesario un **mapa detallado** del mundo en el que se moverá el robot, dado que la creación del grafo de visibilidad implica conocer la situación exacta de los obstáculos del entorno.

2. El software ha de desarrollarse sobre la plataforma **JDE** que es la plataforma del grupo y simplifica el acceso a los sensores y actuadores desde nuestros programas. Esta plataforma ofrece acceso remoto a los sensores y actuadores del robot a través de dos servidores sockets: otos y oculo, y que se explicará con mayor profundidad en el siguiente capítulo.

   El lenguaje de programación en el que se implementará nuestro software a utilizar debe ser C. Así mismo, es imprescindible que la compilación y ejecución se realicen en el sistema operativo Linux, y partícularmente en su distribución Debian, ya que la plataforma utilizada se ha desarrollado para este sistema.

3. La ejecución del objetivo principal debe ser **vivaz** no obteniendo tiempos muertos en la computación demasiado largos y proporcionando una interfaz amable al usuario.

4. La implementación debe tener carácter de software libre y por tanto tener licencia GPL.

2.3. Metodología empleada

Esta sección describe la metodología utilizada para la realización de este proyecto. Básicamente se basa en iteraciones que se componen de diseño, implementación y experimentos para cada uno de los subobjetivos planteados.

Para la realización de cualquier proyecto informático se deben establecer unas tareas a realizar desde la idea original del proyecto hasta la realización del mismo. Este modelo de desarrollo establece unos requisitos de entrada para producir salidas satisfactorias. El desarrollo de este proyecto se basará en el modelo de desarrollo en espiral basado en prototipos. La elección de este modelo de desarrollo se basa en la necesidad de separar el comportamiento final en varias subtareas más sencillas para luego fusionarlas. Cada tarea finalizada aporta los requisitos e información necesaria para abordar la siguiente iteración del modelo de desarrollo.

La primera iteración, la de creación del grafo de visibilidad, se inicia con la lectura de la posición de los obstáculos que forman el mapa del entorno. De este mapa también se obtiene la posición y orientación iniciales del robot. Tras el proceso de creación del grafo disponemos de una de las entradas necesarias para la segunda etapa de planificación de la ruta. Ésta entrada, junto con la posición indicada por el usuario del lugar hacia el que debe moverse el robot, se utilizarán para calcular una ruta, con cuya obtención finaliza la segunda iteración. La ruta generada y la posición actual del robot deben ser los requisitos de entrada para poder recorrer dicha ruta y finalmente posicionar el robot en el punto indicado.

La gran ventaja de este modelo de desarrollo es la existencia de puntos de control al finalizar cada iteración. Además es altamente flexible en cuanto al cambio de requisitos, hecho muy común en este tipo de proyectos de investigación.

Este tipo de metodología se llama *desarrollo en espiral*. En ella los productos son creados gracias al número de iteraciones que se da en el proceso de vida de software. Cada una de estas iteraciones consta de los siguientes pasos :

1. **Determinar los objetivos.** Los objetivos de un ciclo de desarrollo deben de ser identificados y especificados.
2. **Análisis y desarrollo.** El sistema se analiza previo al desarrollo e implementación del software.
3. **Pruebas.** Las pruebas que testean el cumplimiento de los requisitos fijados.
4. **Planificar.** El proyecto es repasado y la próxima fase de la espiral es planificada.

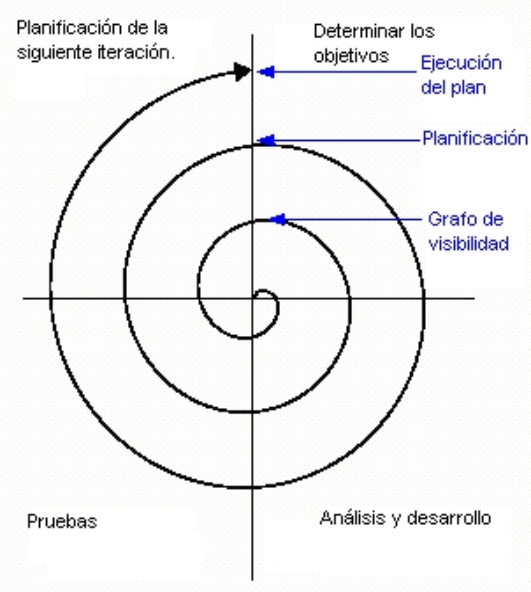


Figura 2.1: Modelo en espiral

En la figura 2.1 se puede observar los cuatro ciclos que forman este modelo de desarrollo: Análisis de requisitos, diseño e implementación, pruebas y planificación del próximo ciclo de desarrollo. Cada iteración corresponderá a cada uno de los subobjetivos marcados.

El modelo que hemos seleccionado se basa en que en cada finalización de iteración se obtiene un nuevo prototipo. Un prototipo es una versión preliminar de un sistema con fines de demostración o evaluación de ciertos requisitos. Se suele estimar la finalización de una iteración en el modelo de desarrollo como la obtención de un nuevo prototipo.

Cierto es que la obtención de un prototipo no implica que se utilice para el producto final, si se cree necesario se puede desechar dicho prototipo.

- **Prototipo 1:** Es capaz de crear un grafo de visibilidad mediante el mapa de entrada
- **Prototipo 2:** Utiliza el grafo de visibilidad para crear una ruta entre la posición origen y la meta
- **Prototipo 3:** Recorre la ruta y posiciona el robot en el punto destino.

Capítulo 3

Plataforma de desarrollo

En el capítulo 2 se describió el objetivo de este proyecto. Antes de explicar en el capítulo 4 su implementación informática, se detallan en el presente capítulo las herramientas software sobre las que nos hemos apoyado para su consecución y una descripción breve del robot al que están orientados los algoritmos implementados.

Por ello en este capítulo se explica en mayor medida el simulador SRIsim y la plataforma JDE utilizada. A un nivel más general se describe la plataforma hardware que disponemos.

3.1. El robot Pioneer

El hardware disponible es el robot Pioneer 3, ideal para entornos de interior. Este robot lo comercializa la empresa norteamericana ActivMedia[1] que ofrece soporte técnico activo. El robot (figura 1.4) dispone de un equipo sensorial para medir el estado de su entorno y unos motores que le permiten moverse.

En cuanto a sensores, el Pioneer dispone de una corona de 16 transductores de ultrasonido que rodean al robot (8 en la parte delantera y 8 en la parte trasera), un sensor laser que realiza un barrido de 180 grados y por último lleva incorporados unos odómetros (*encoders*) asociados a las ruedas para saber cuánto han girado éstas.

Los actuadores principales del robot son dos motores de corriente continua, cada uno asociado a una rueda. Además dispone de una tercera rueda loca sin tracción que unida al movimiento que permiten los dos motores favorece notablemente los giros del robot.

Finalmente, se incorpora un ordenador portátil al robot para ejecutar los programas de navegación. Este ordenador está conectado conectado a una red exterior mediante un enlace inalámbrico, con una tarjeta de red 802.11 que le proporciona una velocidad de 11 Mbps en sus comunicaciones [Robotics, 2003]. De esta forma el programa de control puede correr a bordo del portátil o en cualquier otro ordenador.

Para probar los comportamientos implementados sin transportar la lógica al robot físico, disponemos del simulador SRIsim. Éste nos muestra la posición real de robot y

[1] http://www.activmedia.com

su orientación, así como los obstáculos de su entorno. Para ello, antes de conectar JDE debemos lanzar el simulador y cargar el mapa con los obstáculos de este entorno. En la figura 3.1[a] se muestra la interfaz gráfica de SRIsim en la que se distingue el robot definido como una circunferencia y su orientación que corresponde al radio marcado en la misma. Así mismo, se ven algunos obstáculos a su alrededor y las coordenadas exactas sobre el mapa como son x (abcisas), y (ordenadas) y th (ángulo de orientación). La figura 3.1[b] muestra estas coordenadas de posicionamiento respecto de su entorno.

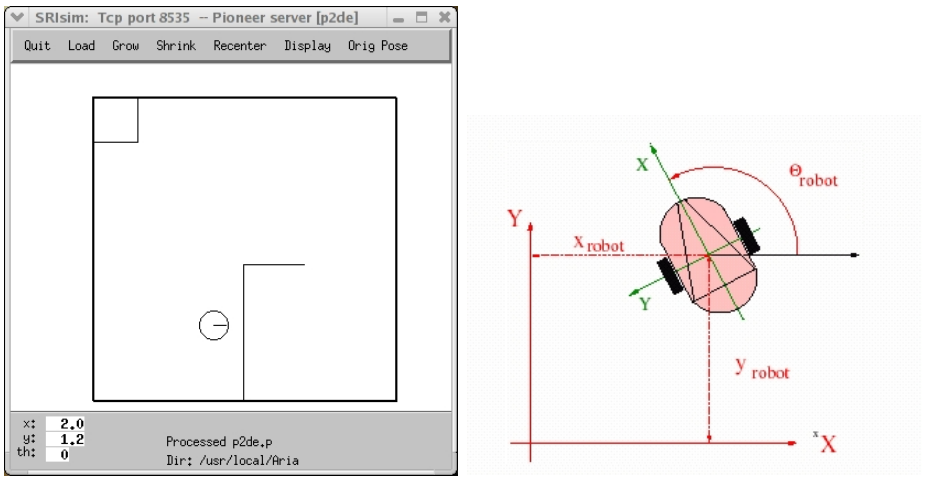


Figura 3.1: Simulador SRIsim [a] y coordenadas de posición del robot [b]

La necesidad de utilizar el simulador del Pioneer en nuestro proyecto en lugar del robot real se debe a que en éste último no conocemos la posición exacta del robot. Los errores de precisión en los odómetros reales se acumulan durante el desplazamiento y hacen que las medidas de posicionamiento sean falsas. El algoritmo implementado en este proyecto realiza básicamente un control de navegación basado en posición y para ello necesita conocer su situación sobre el mapa de manera continua y presisa. Para poder implementar este proyecto en el robot real sería necesario incluir un sistema de localización [Mejías, 2003] que permitiera conocer las coordenadas de posición y orientación reales respecto del entorno.

3.2. Plataforma Software: JDE

El software utilizado para este proyecto se apoya en la plataforma JDE (Jerarquía Dinámica de Esquemas) [Cañas, 2003]. Esta plataforma se ha creado en el grupo de robótica de la URJC y actualmente se desarrollan diversos comportamientos y nuevas funciones para el robot utilizándola como base. JDE hace muy accesible al desarrollador la comunicación con los actuadores y sensores del robot. ARIA es el software que nos proporciona el fabricante del robot y JDE la usa como driver de acceso a sensores y actuadores.

JDE es un entorno de desarrollo que facilita la creación de aplicaciones robóticas sobre el Pioneer. Así mismo, mediante JDE podemos interactuar con el robot Pioneer y sus periféricos como la cámara, cuello mecánico o el láser. Esta plataforma continúa ampliándose con nuevas funcionalidades y debido a su carácter de software libre, se encuentra a disposición en internet [2]. Este entorno de programación, se divide en dos partes relacionadas : Servidores JDE y Esquemas JDE.

3.2.1. Servidores JDE

La arquitectura de servidores y clientes JDE nace de la limitación de que al ejecutar cualquier programa en el robot que utilizara los sensores o los motores, fuera necesario hacerlo directamente en el portátil conectado al robot. Esta arquitectura software se muestra en la figura 3.2.

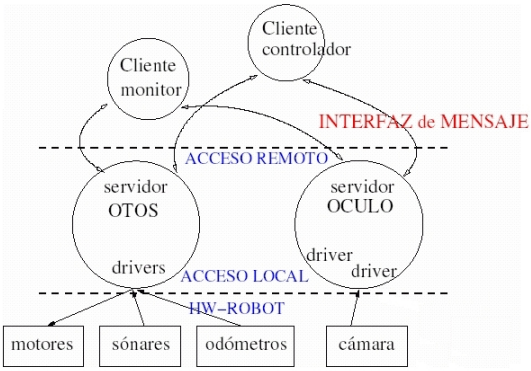


Figura 3.2: Arquitectura software con servidores y clientes de JDE

La tarea de los servidores es ofrecer acceso a los sensores, mótores y periféricos en general, a cualquier programa cliente de la plataforma que los requiera. Cada servidor trabaja con un grupo de sensores y actuadores del robot y ofrece su funcionalidad al conjunto de clientes mediante una *API de mensajes*. Para hacer este acceso, los clientes utilizan la red para intercambiar mensajes con los servidores siguiendo un protocolo.

Se han establecido tres patrones de interacción : el envío bajo demanda de imágenes, suscripción a sensores cortos y las órdenes de movimiento. Entre un cliente y un servidor se establece una conexión `tcp` por la cual ambos se comunican mediante mensajes.

Como se aprecia en la figura 3.2 la principal información que suministra `oculo` es la de las imágenes capturadas con la cámara. Éstas se envían a los clientes a través de un servicio de *imágenes bajo demanda*. Esto significa que cada vez que un cliente necesita una imagen la solicita y el servidor le responde con dos mensajes, uno donde se especifica el tamaño de la imagen, y otro en el que se adjunta dicha imagen.

El servidor `otos` interactúa con los sensores de proximidad como los ultrasonidos y el láser. Se puede conectar tanto al robot real como al simulador y ofrece un *servicio de*

[2] http://gsyc.escet.urjc.es/jmplaza/software.html

suscripción directa a los sensores. Este servicio consiste en que el servidor envía datos de un determinado sensor al cliente cada vez que existan nuevas medidas, previa subscripción del cliente a este sensor. Por ejemplo, en nuestro proyecto utilizamos subscripción directa a los odómetros para conocer la posición y orientación de nuestro robot respecto de su entorno.

Otro tipo de comunicación entre servidores y clientes es el que envía información a los actuadores. Para ello los clientes mandan las *órdenes de movimiento* a los servidores, y éstos transmiten directamente a los actuadores las órdenes explicitas para realizar los movimientos. Para mover los dos motores del robot encargados del desplazamiento y giro del Pioneer, Otos envía órdenes independientes a ambos motores. Esto hace que el robot tenga una tracción diferencial similar a la de los tanques, en la que para cambiar la orientación del robot se le da más velocidad a una rueda que a otra.

Respecto a las variables de actuación, disponemos de las variable v, que comanda el valor de la velocidad lineal y w, que lo hace con la velocidad angular. Con ellas conseguimos abstraer el desarrollo del tipo de tracción del robot y del número de motores que lo forman.

3.2.2. Esquemas JDE

La aplicación robótica dentro de JDE se plantea como un conjunto de hebras llamadas esquemas, que se ejecutan periódicamente a modo de iteraciones. La ejecución concurrente de varios esquemas dan lugar a un *comportamiento*, en nuestro caso un comportamiento de navegación.

Nuestro proyecto se articula en JDE como una colección de esquemas. De un lado los *esquemas de servicio* que incorpora la plataforma y de otro los *esquemas genuinos* que se encargan de la navegación propiamente dicha.

Cara a la programación de esos esqumas genuinos, la plataforma ofrece variables sensoriales y de actuación. Estas variables son utilizadas por varios esquemas a la vez por lo que podemos tener concurrencia. El uso de semáforos haría más fiable el acceso a estas variables pero no se implementan para no producir más retardos y desincronización en las lecturas. Por lo tanto se asume la posibilidad de condiciones de carrera esporádicas.

Algunos de estos esquemas son los *esquemas de servicio* que funcionan como clientes de `otos` y `oculo`. A continuación se detallan algunos de estos esquemas que se utilizan en nuestro proyecto :

Esquema sensationsotos

Este esquema perceptivo ofrece las lecturas de los diferentes sensores del robot actuando como cliente del servidor `otos`. La interfaz de `sensationsotos` consta de un grupo de variables que se actualizan continuamente con los datos sensoriales del robot. La más utilizada en la navegación por posición, y por tanto en nuestro proyecto, es la estructura `robot[5]` que nos ofrece información sobre la odometría y la situación exacta del robot.

Esta estructura consiste en un array de 5 variables con los siguientes datos. Las posiciones robot[0] y robot[1] corresponden a las coordenadas de los ejes x e y respectivamente en el sistema referencial del mundo (mm). La tercera posición robot[2] indica el ángulo de orientación θ también en el sistema referencial del mundo (radianes). Por último robot[3] y [4] corresponden al seno y coseno respectivamente del ángulo de orientación θ.

Estos datos son imprescindibles para el desplazamiento de nuestro robot por la ruta planificada. Sin ellas no sabríamos cuánto hemos recorrido ni hacia dónde dirigimos el robot. Estas medidas son exactas ya que las estamos utilizando sobre el simulador, las cuales carecen deliberádamente de ruido. Esta característica es imposible en el robot real, donde la odometría lleva inevitáblemente a errores. El simulador SRIsim ofrece la posibilidad de incluir cierto grado de ruido sobre la ódometría. Para incluirlo debemos insertar en el fichero de configuración del simulador **p2de.p** los valores *EncodeJitter*, *AngleJitter* y *AngleDrift*. Si les asignamos el valor 0.0 conseguimos que el robot carezca de ruido y que por tanto las medidas de los odómetros sean exactas.

Otra de estas variables con datos sensoriales relevante es `us[16]`. Un array de 16 posiciones que ofrece las medidas de los 16 sensores de ultrasonido. Otra estructura importante es `laser[180]` en la que se guardan las 180 medidas que es capaz de procesar el láser. En nuestro proyecto no las utilizamos dado que no nos valemos de estos sensores durante la navegación si no, como se indicó anteriormente, de la posición que nos indican los odómetros.

Esquema motors

En nuestro proyecto utilizamos este esquema para que nuestra aplicación gobierne al robot en su navegación al destino marcado.

El esquema actuador `motors` envía periódicamente al servidor `otos` la velocidad lineal y angular que se quiere ordenar a los motores del robot en *mm/s*. Estos datos se guardan en las variables globales `v` y `w` que corresponden a las velocidades lineal y angular. La variable `v` indica a los motores que avancen linealmente según su valor dado en mm/s. En nuestro proyecto modificamos esta variable durante el desplazamiento del robot por los distintos tramos de la planificación. Aumentamos `v` para darle una mayor velocidad cuando el robot se encuentre alejado del obstáculo y la disminuimos para ralentizar su acercamiento a un punto en el que debemos realizar un giro. Estos giros se realizan dando un valor a `w`, también en mm/s, o también para corregir las posibles desviaciones del robot.

Esquema guixforms

Este esquema de servicio proporciona una interfaz gráfica al usuario, refrescada periódicamente y mostrando tanto las medidas sensoriales como el efecto de las acciones del robot en tiempo real. Este esquema tiene una gran utilidad en cuanto a la depuración del comportamiento generado, ya que podemos visualizar estructuras internas y activar o desactivar esquemas a voluntad para realizar trazas del comportamiento.

También ofrece la posibilidad de joysticks para teleoperar los motores del robot.

En este proyecto modificamos este esquema para incluir elementos propios de nuestra aplicación y eliminar los que no nos son útiles. En particular ampliamos la ventana de visualización del entorno e incluimos grupos de botones dedicados a tareas concretas de las distintas etapas del proyecto, como explicaremos en el capítulo 4.

Este esquema se ejecuta a una frecuencia de 5 veces por segundo, suficiente para generar un refresco aceptable al usuario.

Como ilustra la figura 3.3 el código de nuestro proyecto se inserta en esta arquitectura software como una aplicación dentro de JDE, articulada en esquemas y sobre el servidor Otos. Es éste último el que finalmente interactúa con el simulador SRIsim para recoger las medidas sensoriales y materializar los comandos de movimiento.

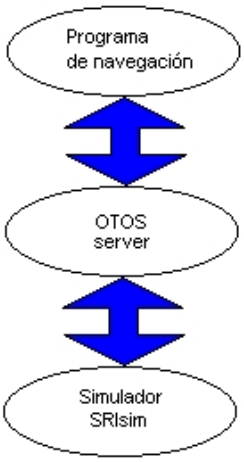


Figura 3.3: Arquitectura software de nuestro proyecto

Capítulo 4

Implementación informática

Sobre la plataforma descrita en el capítulo anterior, hemos implementado los algoritmos encargados de realizar los objetivos planteados en el capítulo 2. Estos algoritmos y su implementación realizada en código C se detallan en este capítulo. El código fuente de esta implementación está formada por un conjunto de ficheros que se añaden a los proporcionados por los de la plataforma JDE (todos estos ficheros se incluyen en el CD del anexo).

4.1. Grafo de visibilidad

En esta sección se explica el método del grafo de visibilidad y se definen los términos que manejamos a lo largo de este capítulo.

Podemos decir que un *grafo de visibilidad* es aquel grafo que, a partir de un mapa que incluye la situación de los obstáculos fijos, tiene como nodos los vértices de estos obstáculos y cuyas aristas son las posibles uniones de cada nodo con el resto, siempre y cuando no crucen otros obstáculos situados en el mapa [Ford, 1984]. En la figura 4.1 se muestra un posible grafo de visibilidad para un mapa de tres obstáculos triangulares.

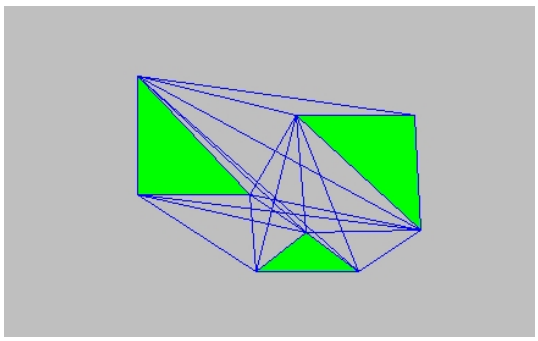


Figura 4.1: Grafo de visibilidad de 3 obstáculos triangulares

Este tipo de grafo se denomina *de visibilidad* porque cada arista generada es la unión de cada nodo con sus nodos visibles. Entendemos por dos nodos visibles entre sí, aquellos que se pueden unir mediante un segmento rectilíneo sin cruzar ningún obstáculo. Sobre esta red de posibles caminos se planificará posteriormente una ruta óptima para navegar desde la posición que ocupa el robot hasta un punto remoto del escenario. Debido a que sabemos que recorriendo estas aristas nunca cruzaremos obstáculos, cada una de ellas serán posibles caminos transitables para la siguiente etapa de planificación.

No podemos aplicar esta técnica directamente sobre los vértices de los obstáculos reales sin haber creado una *zona de seguridad* alrededor de cada obstáculo para evitar que el robot choque con su lateral al pasar muy cerca de ellos.

Si el mundo se describe con polígonos formados por segmentos rectos, cada zona de seguridad está formada por otros cuatro segmentos que rodean al obstáculo, dos paralelos y perpendiculares a los que llamaremos *segmentos de seguridad*. Estos segmentos se encuentran a una distancia *d* que deberá ser mayor que la del radio del robot, por si en la planificación se utiliza alguno de estos como parte de la ruta. De esta forma siempre existirá una separación entre el robot y el obstáculo suficiente para impedir la colisión.

En la implementación realizada esta distancia de seguridad se encuentra definida como constante en el fichero *pintamapa.c* y es la suma del radio del robot con un margen de seguridad añadido, como se muestra en la figura 4.2.

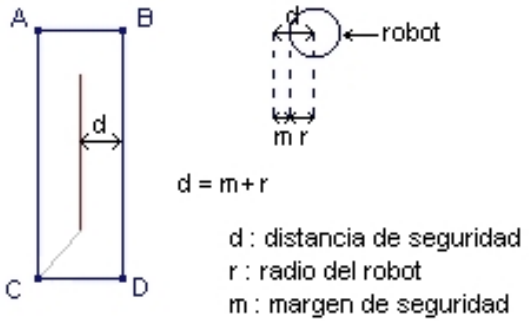


Figura 4.2: Segmento obstáculo con cuatro segmentos de seguridad a una distancia d

Para la creación de estos cuatro segmentos de seguridad se crean inicialmente dos segmentos paralelos a la ecuación de la recta del obstáculo separados por la distancia de seguridad definida anteriormente. A continuación, dos rectas perpendiculares a la recta del segmento obstáculo y que pasen cada una de ellas por sendos vértices del mismo. Se crean también dos paralelas a ambas rectas perpendiculares a una distancia *d* y *-d* de ambos vértices y se calculan los cuatro puntos de intersección de las rectas perpendiculares con las rectas paralelas [de Berg, 1987]. Estos cuatro puntos *A, B, C* y *D* serán los vértices de los cuatro segmentos de la zona de seguridad, como se muestra en la figura 4.2.

Una vez que hemos construido las zonas de seguridad generamos el grafo de visibili-

dad. Cada uno de los vértices de las zonas de seguridad serán los nodos de nuestro grafo. A las aristas que unen estos nodos las llamaremos *segmentos de visibilidad*. Se permite que los segmentos de visibilidad lleguen a tocar segmentos de seguridad sin cortarlos ya que los propios segmentos que forman la zona de seguridad son posibles caminos en el grafo.

4.2. Construcción del grafo de visibilidad

Para programar el almacenamiento de los datos del grafo de visibilidad, hemos creado una serie de tipos de datos que almacenan los segmentos con los que trabajamos y los puntos que los definen.

4.2.1. Implementación directa

Un punto cualquiera en el mapa lo definimos como dos valores reales que equivalen a las variables x e y en un eje de coordenadas. Este tipo de datos viene ya definido en la plataforma JDE, más concretamente en el fichero jde.h como *Tvoxel* :

```
typedef struct voxel{
            float x;
            float y;}Tvoxel;
```

El tipo de dato correspondiente a los segmentos es un registro que esta formado a su vez por otros dos registros de tipo punto correspondientes a sus vértices. Posee dos variables más que almacenan su longitud y su punto medio. Este tipo de datos viene definido en el fichero *lasersegments.h* que forma parte de la plataforma JDE como *tsegmento* :

```
typedef struct segmento {
             Tvoxel p1;
             Tvoxel p2;
             double longitud;
             Tvoxel pmedio; }tsegmento;
```

Una serie de listas enlazadas almacenan las distintos clases de segmentos en cada etapa de creación del grafo :

1. **Lista de obstáculos.** Almacenamos los obstáculos leídos del mapa en una lista enlazada de segmentos de obstáculo.

2. **Lista de seguridad.** Creamos una zona de seguridad con cada uno de estos segmentos de obstáculo. Esto lo realizamos recorriendo la lista de obstáculos y almacenando en esta lista los cuatro segmentos necesarios para definir su zona de seguridad.

3. **Vector de nodos del grafo de visibilidad.** Los vértices de cada segmento de seguridad son los nodos que se ingresan en el vector de visibilidad. Cada nodo tiene Tvoxel para almacenar su posición. Por sencillez hemos decidido asignarles una numeración correlativa, de esta forma podemos identificarlos como números enteros en lugar de como coordenadas reales.

   Esta estructura se rellena a la vez que insertamos segmentos en la lista de seguridad, comprobando antes que no se ha incluido previamente ya un nodo con las mismas coordenadas. Esta comprobación es necesaria porque existen casos en los que al menos dos zonas de seguridad comparten alguno de sus vértices como se muestra en la figura 4.3.

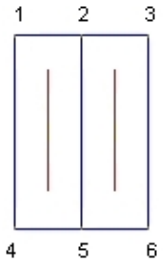


Figura 4.3: Dos segmentos de obstáculo compartiendo dos nodos de sus zonas de seguridad

4. **Lista de segmentos de visibilidad**. Con el vector de nodos podemos comenzar a trazar posibles segmentos de visibilidad entre ellos y comprobar si cumplen las condiciones necesarias para poder considerarlos segmentos válidos de visibilidad.

   Cada par de nodos genera un candidato a segmento de visibilidad. Para rellenar esta lista, es necesario recorrer el vector de nodos y comprobar si se cumplen las condiciones necesarias. Este proceso se realiza sin repetir comprobaciones de tal manera que se comprobará el nodo número 1 con el resto, el nodo 2 con el resto (sin repetir la comprobación con el nodo 1), el nodo 3 con el resto (sin repetir la comprobación con el nodo 1 ni con el 2), y así sucesivamente hasta comprobar cada pareja de nodos.

   Las condiciones que deben cumplir una pareja de nodos para poder formar parte de un segmento de visibilidad válido las englobamos como una única *condición de descarte*. Estas condiciones son las siguientes:

   a) El segmento formado por estos dos nodos no debe cortar ningún segmento de seguridad.
   b) Este mismo segmento no debe tocar o cortar ningún segmento de obstáculos.

   Hemos programado las funciones que comprueban geométricamente si dos segmentos se cortan, se tocan o ninguna de las dos cosas. Consideramos que dos segmentos

se *cortan* cuando intersectan en un punto sin que ninguno de los vértices forme parte del otro segmento. Por el contrario entendemos que dos segmentos *tocan* cuando uno de los vértices de un segmento forma parte del otro segmento.

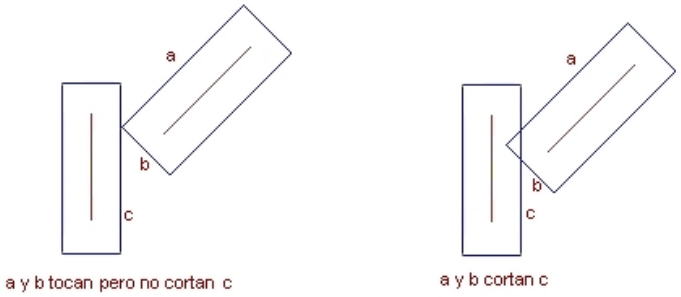


Figura 4.4: Dos segmentos obstáculo cuyos segmentos de seguridad tocan o cortan

Con estas dos condiciones podemos asegurar que los segmentos de visibilidad insertados en la lista de visibilidad no pasan demasiado cerca de los obstáculos ni atraviesan las zonas de seguridad creadas alrededor de los mismos.

El grafo de visibilidad se almacena por lo tanto en la lista de segmentos de visibilidad y en el vector de nodos. Además, para facilitar los algoritmos de planificación también se almacena en una tabla de conectividad, que guarda por cada segmento de visibilidad una conexión recíproca entre los nodos. De este modo, cada nodo que esté conectado con otro, tiene su conexión inversa en la tabla (Si A está conectado con B, B está conectado con A).

Esta tabla, define el grafo de visibilidad completo ya que contiene el conjunto de nodos y aristas necesario. La implementación de esta tabla es un modo más compacto de expresar la lista de visibilidad haciendo accesibles más rápidamente qué segmentos de visibilidad parten o llegan a cada nodo del grafo. En la figura 4.5 se muestra un mapa con los nodos del grafo de visibilidad numerados y su correspondiente tabla de conectividad en la tabla 4.1:

Tras explicar la implementación del proceso de creación del grafo de visibilidad y las estructuras utilizadas, se realizan ciertas consideraciones sobre el algoritmo y mejoras realizadas sobre el mismo.

Consideraciones

El orden de complejidad del algoritmo es muy alto ($O(n^2)$), ya que es necesario comprobar la posible unión de cada nodo con el resto. Esto hace que el proceso sea muy lento para un número elevado de nodos.

Nodo	Lista de conexiones
1	31 , 17 , 14 , 13 , 6 , 5 , 2
2	32 , 17 , 15 , 14 , 13 , 8 , 1
3	29 , 20 , 19 , 16 , 15 , 7 , 5 , 4
4	20 , 19 , 16 , 15 , 7 , 3
5	29 , 6 , 3 , 1
6	31 , 5 , 1
7	30 , 20 , 19 , 16 , 15 , 8 , 4 , 3
8	32 , 15 , 13 , 7 , 2
9	27 , 24 , 18 , 17 , 10
10	22 , 9
11	23 , 20 , 18 , 12
12	26 , 21 , 20 , 11
13	32 , 17 , 15 , 14 , 8 , 2 , 1
14	19 , 17 , 16 , 13 , 2 , 1
15	32 , 20 , 19 , 16 , 13 , 8 , 7 , 4 , 3 , 2
16	20 , 19 , 17 , 15 , 14 , 7 , 4 , 3
17	32 , 27 , 19 , 18 , 16 , 14 , 13 , 9 , 2 , 1
18	27 , 24 , 23 , 20 , 17 , 11 , 9
19	20 , 17 , 16 , 15 , 14 , 7 , 4 , 3
20	24 , 23 , 21 , 19 , 18 , 16 , 15 , 12 , 11 , 7 , 4 , 3
21	26 , 22 , 20 , 12
22	28 , 21 , 10
23	25 , 24 , 20 , 18 , 11
24	27 , 23 , 20 , 18 , 9
25	23
26	21 , 12
27	24 , 18 , 17 , 9
28	22
29	5 , 3
30	7
31	6 , 1
32	17 , 15 , 13 , 8 , 2

Cuadro 4.1: Tabla de conectividad

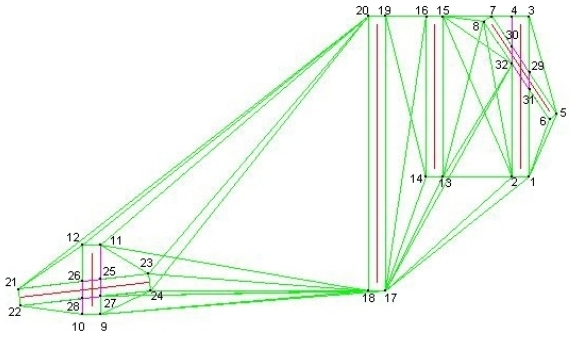


Figura 4.5: Nodos numerados y tabla de conexiones

Puede parecer que en la condición de descarte no hace falta incluir la condición de que un segmento de visibilidad corte a un segmento de tipo obstáculo. Pero como hemos indicado anteriormente, para formar el grafo de visibilidad se unen todos los vértices de los segmentos de seguridad con el resto incluyendo la zona de seguridad en la que está inmerso el obstáculo. Por lo tanto, si unimos los vértices de los segmentos de seguridad de un único segmento obstáculo sin tener en cuenta esta condición, comprobaremos que todos ellos se unen cortando el obstáculo en dos ocasiones, como se muestra en la figura 4.6.

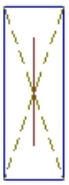


Figura 4.6: Segmento obstáculo con zona de seguridad, cuyos segmentos de visibilidad cortan el obstáculo

4.2.2. Método mejorado

Uno de los inconvenientes que hemos observado de esta implementación directa es que si para crear el conjunto de nodos del grafo tenemos en cuenta únicamente los vértices formados por los segmentos de seguridad, comprobaremos que hay zonas en el mapa donde no se crea el grafo de visibilidad. Como se muestra en la figura 4.7, esto se debe a que los vértices de los segmentos de seguridad están dentro de la zona de seguridad de

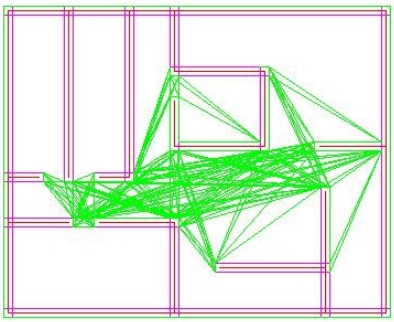


Figura 4.7: Grafo de visibilidad generado por el método simple

otro segmento y por lo tanto no son visibles desde un vértice externo.

Para un mapa que tenga varias habitaciones comprobaremos que el grafo se creará en el pasillo que las recorra y en alguna habitación en particular, pero existirán zonas en las que los vértices de los segmentos de seguridad no sean visibles entre sí y por lo tanto no se generen segmentos de visibilidad entre ellos.

Como hemos visto necesitamos modificar ligeramente el planteamiento para poder acceder a todas las partes en las que el método simple no puede crear el grafo. Hasta ahora hemos generado los segmentos de visibilidad utilizando como nodos únicamente a los vértices de los segmentos de seguridad.

La solución que hemos diseñado y programado respecto a este problema es que antes de generar el grafo, cuando se están construyendo los segmentos de seguridad, no tengamos en cuenta únicamente los vértices de los segmentos de seguridad para generar el grafo de visibilidad. Se consideran también nodos posibles del grafo los puntos donde se corten estos segmentos entre sí ya que en todo momento se encontrarán a la distancia de seguridad definida para el robot, y por tanto no habrá peligro de colisión.

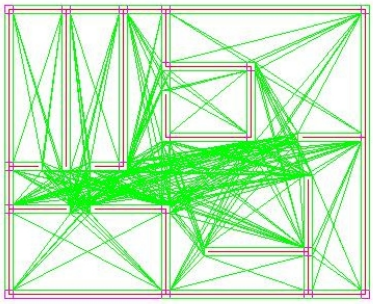


Figura 4.8: Grafo de visibilidad generado por el método mejorado

Incluyendo esta modificación generamos el nuevo grafo de visibilidad sobre el mismo

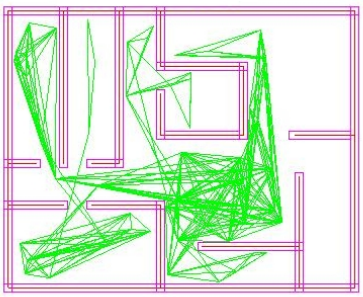


Figura 4.9: Grafo de visibilidad generado por el método del azar

mapa que utilizamos para el método anterior y comprobamos que ahora las habitaciones sí son accesibles por el grafo, como se puede ver en la figura 4.8.

4.2.3. Azar

Además de los dos métodos descritos anteriormente para la creación del grafo, hemos estudiado un método adicional para utilizar un número menor de nodos que en el método mejorado y poder acceder igualmente a la mayoría de las zonas del mapa. Esto se traduce en un menor coste computacional y por tanto, aunque manteniendo la complejidad $O(n^2)$, acelerar la ejecución.

Este método en lugar de tomar como nodos los vértices de los segmentos de seguridad y los puntos en los que intersectan, utiliza el azar para generar los nodos del grafo de visibilidad. Teniendo en cuenta las dimensiones del mapa se pueden crear una serie de nodos aleatoriamente que caerán como una distribución homogénea sobre toda su superficie.

Se crean 4 nodos por cada segmento obstáculo teniendo en cuenta que con el método anterior se creaba como mínimo el mismo número para cada segmento de tipo obstáculo debido a los 4 segmentos de tipo seguridad que lo rodean. Esta condición es plenamente heurística y podemos modificar en el código este número para poder crear un número diferente de nodos.

Este método es más rápido que el método mejorado y accede a más zonas que el método simple, pero también tiene sus desventajas. Puede darse la situación en la que la distribución no sea tan homogénea como deseamos y que haya zonas en las que no existan nodos. Esto supone que haya espacios en los que no se creen segmentos de visibilidad y a los que posiblemente no se pueda planificar una ruta ya que existan puntos no visibles desde otro nodo. Una posible solución sería crear más nodos por cada obstáculo del mapa aumentando la constante que define esta relación. Otra solución sería distribuir los nodos de tal manera que no quedaran demasiado cercanos unos de otros y por tanto el grafo accedería a más zonas del mapa.

Otro punto a tener en cuenta es la posibilidad de que la planificación de la ruta

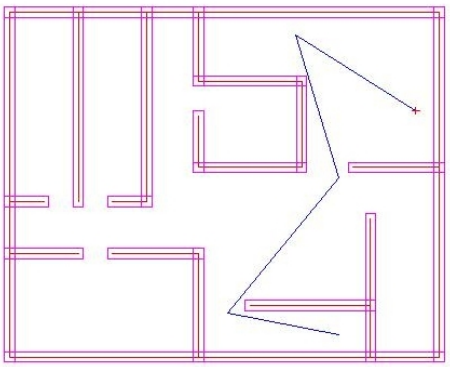


Figura 4.10: Ruta planificada en un grafo generado por el método del azar

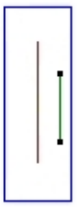


Figura 4.11: Isla de nodos dentro de una zona de seguridad

tenga que realizar giros o desvíos innecesarios. Esto sucede cuando un nodo es únicamente accesible desde otro nodo lejano y éste sólo tenga un grupo de nodos visibles muy cercanos a él. Para llegar a este grupo de nodos puede ser necesario alejarse de la meta y así poder acceder al único nodo visible desde la meta, como se muestra en la figura 4.10

En el caso en el que caigan nodos dentro de una zona de seguridad éstos no se descartan y forman parte del grafo, aunque no conectarán con ningún nodo externo a esta zona puesto que cumplirá una de las condiciones de descarte que es la de cortar a un segmento seguridad. Por esto, los nodos inmersos en zonas de seguridad formarán "islas"que podrían tener conexiones otros nodos incluidos en su misma zona de seguridad, pero que estarán aislados del resto del grafo. Esta circunstancia se muestra en la figura 4.11

4.3. Planificación

Una vez que el grafo de visibilidad ha sido generado por alguna de las tres técnicas descritas podemos utilizarlo para trazar posibles rutas recorriendo sus aristas entre la posición del robot y el destino marcado. El cálculo de estas rutas requiere de una etapa de planificación para generar la ruta entre los puntos de origen y meta. Se han implementado dos algoritmos de búsqueda independientes como son Dijkstra y A*. De esta forma podemos comparar los resultados de ambos sobre un mismo problema.

La mayor diferencia entre estos algoritmos está en la velocidad de respuesta siendo por regla general más rápido el algoritmo A* por crear un único camino entre el robot y la meta marcada. Por el contrario, el algoritmo de Dijkstra crea todos los caminos mínimos entre el robot y sus nodos conexos en el grafo pero tiene la ventaja de ofrecer el camino mínimo entre dos puntos.

Las características de ambos algoritmos, así como un ejemplo de cada uno de ellos se explica en los siguientes apartados.

4.3.1. Algoritmo de Dijkstra

También llamado algoritmo de caminos mínimos, se trata de un algoritmo para la determinación del camino más corto entre el nodo origen y el resto de los nodos conexos en un grafo. Su nombre se refiere a su desarrollador holandés *Edsger Wybe Dijkstra*.

La idea básica de este algoritmo de búsqueda consiste en ir explorando todos los caminos que parten del vértice de origen y que llevan a todos los demás vértices. Cuando se obtiene el camino mínimo para todos los nodos del grafo conexos desde el nodo origen, se detiene el algoritmo. [Cormen, 1990]

Para implementar este algoritmo sobre los nodos del grafo de visibilidad hemos definido como coste de cada arista, la distancia euclídea entre sus dos vértices, de los que disponemos sus coordenadas exactas en la tabla de conectividad.

Este algoritmo tiene como premisa que el grafo sobre el que se realizará la búsqueda sea conexo, es decir que desde cada nodo se pueda alcanzar al resto siguiendo las aristas que los unen en uno o varios pasos, o dicho de otra manera, que no existan islas de nodos desconectados del resto del grafo. Por esta razón hemos incluido una condición adicional de finalización para cuando se haya terminado de calcular el camino mínimo hacia todos los vértices conexos al vértice origen. Se desprecian los vértices que no pueden conectarse con el nodo origen y que forman "islas"de vértices a las que el robot desde su posición actual no puede alcanzar.

El algoritmo de Dijkstra es de complejidad cuadrática O(n^2), donde n es el número de vértices) ya que realiza la búsqueda desde su posición origen hasta el resto de los nodos.

El encaminamiento de paquetes de datos en redes de ordenadores es formalmente análogo al cálculo de trayectorias de un robot dentro de un grafo de visibilidad. A continuación se describe la implementación realizada para este algoritmo.

Implementación

Como nodos origen y destino utilizamos respectivamente las posiciones del robot y el destino marcado por el usuario (su definición se detallará en la siguiente sección), incluyéndolas en la tabla de conexiones temporalmente. Para ello debemos añadir estos dos nuevos puntos como nuevos nodos del grafo de visibilidad y actualizar las conexiones de todos los nodos con ambos. Posteriormente, cuando se haya calculado el camino mínimo entre ellos, se eliminarán del grafo y se volverá a actualizar las conexiones del resto de los nodos.

La estructura principal sobre la que se calcularán los caminos mínimos será un vector cuyas posiciones se corresponden con cada uno de los nodos del grafo. A este vector lo denominamos *vector de caminos mínimos* y mostramos un ejemplo de cómo se rellena durante la planificación en la figura 4.13. Para cada una de las posiciones del vector se incluyen los siguientes valores :

- El camino calculado desde el nodo origen hasta el nodo correspondiente a esa posición del vector. Se representa como un vector de enteros cada uno de ellos corresponde a un nodo del grafo de visibilidad por el que hay que pasar para llegar al nodo correspondiente a la posición del vector de caminos mínimos. A este vector lo llamaremos *vector de camino calculado.*

- El coste del camino calculado, que será un valor real con la suma de las distancias entre los nodos del vector de camino calculado.

- Una variable booleana para determinar que el camino incluido en el vector de camino calculado es realmente el camino mínimo desde el origen hasta este nodo. A este indicador lo denominamos como *indicador de nodo visitado.*

El cálculo de la ruta mínima entre nodo origen y nodo destino según el algoritmo de Dijkstra consiste en los siguientes pasos que se ejecutan iterativamente:

1. Inicializa el vector del camino calculado de cada nodo del vector de caminos mínimos a $\emptyset$, excepto el del nodo inicial, cuyo camino estará formado por el propio nodo y cuyo coste será 0. Inicialmente el coste hacia cada uno de los nodos será igual a ∞ y su indicador de nodo visitado igual a FALSO.

   Al comienzo consideramos el nodo origen como *nodo actual*, al que denominaremos así durante toda la iteración. El valor del nodo actual cambiará en cada iteración conteniendo a cada uno de los nodos del grafo conexos con el nodo origen.

2. Si aún queda por calcular el camino mínimo de algún nodo conexo al nodo origen continúa con la siguiente iteración, en caso contrario finaliza. Esta condición la implementamos mediante una función que calcula la cantidad de nodos del vector de caminos mínimos para los que no se ha terminado de calcular el camino mínimo, es decir que tengan su indicador de camino visitado a FALSO.

   Si esta cantidad no varía en una de las iteraciones respecto a la anterior finalizará el algoritmo puesto que habrán quedado islas dispersas del nodo a los que es imposible

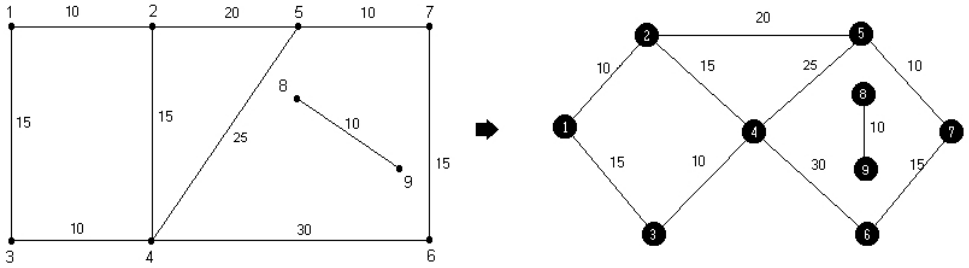


Figura 4.12: Ejemplo algoritmo de Dijkstra. Grafo inicial

llegar. Otra condición de finalización es que tan solo quede un nodo por calcular el camino mínimo, este será por lo general el nodo más alejado del origen y ya contendrá el camino mínimo dado que todos sus nodos adyacentes lo tendrán.

3. Calcula el coste del camino desde el nodo actual hasta sus nodos adyacentes, teniendo el cuenta el coste del camino calculado desde el origen hasta el nodo actual.

4. Por cada uno de los nodos adyacentes al nodo actual : si el coste del nuevo camino hasta el nodo adyacente es menor que su vector de camino calculado, actualiza este vector con el nuevo camino y su coste.

5. Actualiza el indicador de nodo visitado para el nodo actual a CIERTO.

6. Cambia el nodo actual al nodo que menos coste tenga en su camino mínimo y cuyo indicador de nodo visitado sea FALSO.

7. Vuelve al paso 2

El algoritmo retorna el vector de caminos mínimos con las rutas más cortos desde el nodo inicial hasta los nodos conexos a este. Estas rutas están contenidas en cada vector de camino calculado. En el caso en el que hubiera nodos no conexos con el nodo inicial, estos mantendrán un coste infinito ya que se consideran inaccesibles desde la posición del robot.

Ejemplo

A continuación se muestra un ejemplo del funcionamiento del algoritmo de Dijkstra para una figura geométrica cuyos costes equivalen a la distancia euclídea entre los vértices. Para mayor sencillez se transforma la figura geométrica en otra figura topológica equivalente como se muestra en la figura 4.12.

Partimos del nodo inicial 1, el cual tendrá desde el comienzo un coste 0 y como camino mínimo el propio nodo. El resto de los nodos no tendrán caminos mínimos asignados y el coste de los mismos será infinito como se muestra en la inicialización de la figura 4.13.

Calculamos los caminos desde el nodo inicial hasta los dos nodos adyacentes numerados como 2 y 3. Asignamos como camino mínimo hasta el nodo 2 el camino : 1,2 con coste 10, dado que este valor es menor que el coste inicial (∞). Igualmente para el nodo 3 su camino mínimo pasará a ser 1,3 con coste 15. El estado de los costes y de los caminos asignados se muestra en la figura 4.13 en la primera iteración.

Marcamos el nodo 1 como 'tratado', marcando a CIERTO el indicador booleano de camino mínimo, y pasamos al nodo 2 por ser el nodo no tratado con menor coste (10 frente a los 15 del nodo 3). Los nodos adyacentes al nodo 2 son el nodo 4 y el nodo 5. Para el nodo 5 calculamos una ruta con un coste de 30 atravesando los nodos 1,2,5. De igual manera para el nodo 4 se calcula una ruta con coste 1,2,4 con coste 25. El grafo queda como muestra la figura 4.13 en la segunda iteración.

En la tercera iteración del bucle elegimos el nodo 3 como nodo actual por ser el nodo no tratado con menor coste. Los nodos adyacentes al nodo 3 y que no han sido tratados son el 4 y el 6. Recalculando un camino hacia el nodo 4 vemos que el coste del camino 1,3,4 con 25 es igual que el coste de su camino ya asignado 1,2,4 que es de 25. Por lo tanto, no actualizamos el camino a este nodo. Se calculan los caminos hacia estos nodos y se actualizan sus costes.

En la cuarta iteración partimos desde el nodo 4. Sus nodos adyacentes no tratados son el 5 y el 6. Recalculando los caminos para ambos nodos, no encontramos una ruta con un coste menor al de los costes respectivos, por lo que marcamos el nodo 4 como tratado y continuamos con la siguiente vuelta del bucle.

Pasamos al nodo 5 por ser el nodo no tratado con menor coste asignado. Desde este nodo calculamos una nueva ruta para el nodo 7 que es el único nodo adyacente no tratado. La ruta asignada 1,2,5,7 a este nodo tendrá un coste de 40.

Marcamos el nodo 5 como tratado y continuamos con el nodo 6 que tiene un coste de 35. El único nodo adyacente a este y que no ha sido tratado es el nodo 7, pero la ruta calculada para este 1,2,4,6,7 tendría un coste de 70 que es mayor que el coste de la ruta ya asignada 1,2,5,7, con lo que no la modificamos. Se muestran los caminos generados hasta el momento en la figura 4.13 para la quinta iteración.

Tras marcar el nodo 6 como tratado nos queda únicamente el nodo 7 por tratar. Esta es una condición de final de algoritmo, ya que cualquier nodo adyacente a este ya estaría tratado. La figura 4.13 muestra en su última iteración los costes y caminos mínimos finales.

4.3.2. Algoritmo A*

A* (léase A asterisco o A estrella) es un algoritmo de búsqueda heurística del camino más corto desde un nodo inicial a un nodo meta. Ambos nodos deben estar incluidos en un grafo a cuyos arcos se les ha asignado un coste, que este caso será la distancia euclídea. A diferencia de Dijkstra, no calcula la ruta de un nodo al resto, si no desde un nodo origen a un único nodo destino.

Cada nodo contendrá un valor heurístico acerca de la distancia estimada hasta la meta. También tendrán un enlace paterno que apunte al mejor nodo desde el que se ha

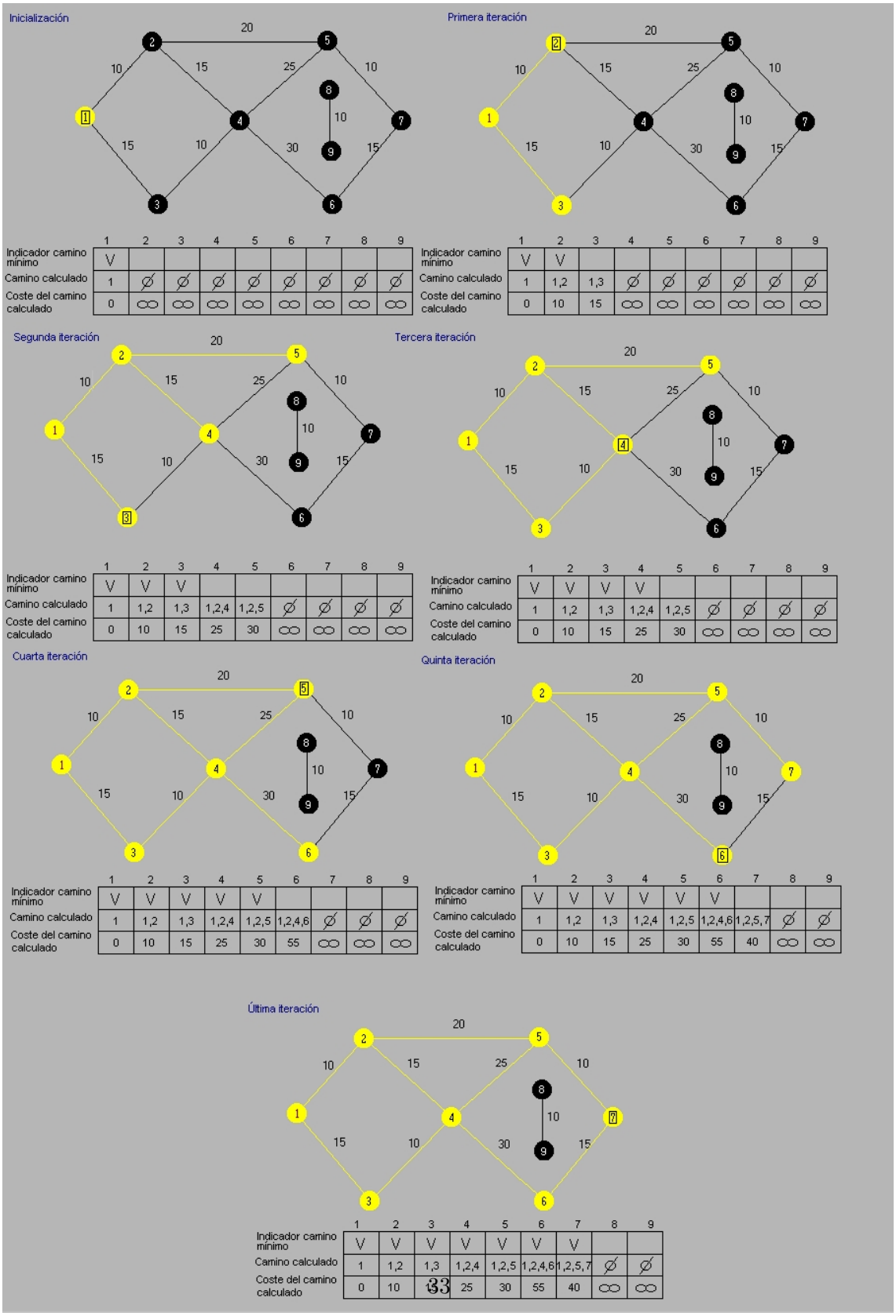


Figura 4.13: Ejemplo algoritmo de Dijkstra. Iteraciones

llegado al actual. El enlace paterno hace posible restablecer un camino entre el nodo origen y el nodo meta una vez que se ha encontrado dicha meta.

La función heurística calculada permitirá que el algoritmo busque en primer lugar los caminos con más probabilidad de éxito de generar un camino válido hacia la meta. Llamemos a esta función f. Se define esta función como la suma de dos componentes, que llamaremos g y h. La función g es una medida del coste de ir desde el estado inicial hasta el nodo actual. La función h es una estimación del coste adicional de llegar desde el nodo actual al nodo meta. [J. Mira, 1995]

Necesitaremos usar dos listas de nodos, a las que llamaremos '*lista abierta*' y '*lista cerrada*'. La lista abierta contiene los nodos que podrían formar parte del camino que queremos tomar, pero que quizás no lo hagan. Básicamente, esta es una lista de los nodos que necesitan ser comprobados. La lista cerrada contiene los nodos que ya han sido examinados y que no hace falta volver a examinar.

Implementación

Durante la implementación observamos que podemos utilizar la misma tabla de conexiones que para el algoritmo de Dijkstra incluyendo nuevos campos que sólo se utilizarán para A*. Se añade a cada nodo de la tabla de conexiones un valor entero que indica el padre del nodo. Además, y también por cada posición se incluyen tres valores correspondientes a los valores f, g y h de cada nodo.

Por otro lado implementamos dos listas enlazadas de valores enteros para utilizarlas como las listas abierta y cerrada de nodos.

De igual manera que para Dijkstra, el nodo origen se corresponde con la situación del robot y el nodo destino con la posición marcada por el usuario. Se incluyen temporalmente estos nodos en la tabla de conexiones estableciendo la visibilidad de los mismos con el resto. Al finalizar el algoritmo se eliminarán estos dos nodos del grafo de visibilidad y se eliminarán todas sus conexiones con del resto de los nodos.

El siguiente pseudocódigo explica por pasos cual es la lógica del algoritmo y cómo se actualizan las estructuras implementadas [1]:

1. Inicializar a vacío las dos listas, abierta y cerrada.
2. Añadir el nodo inicial a la lista abierta con sus correspondientes valores f, g y h.
3. Repetir lo siguiente:
   a) Buscar el nodo con el coste f más bajo en la lista abierta. Nos referimos a éste como el nodo actual.
   b) Mover el nodo actual a la lista cerrada, eliminándolo de la lista abierta.
   c) Comprobamos los nodos adyacentes al nodo actual mediante la tabla de conexiones. Para cada uno de ellos :

[1] http://elthanar.iespana.es/elthanar/tutoriales/articulo1.htm

1) Si no es transitable o si está en la lista cerrada, se ignora. En cualquier otro caso continuar con los dos siguientes pasos :
2) Si no está en la lista abierta, añadir a la lista abierta. El nodo actual debe ser el padre de este nodo. Almacenar los costes f, g y h del nodo.
3) Si ya está en la lista abierta, comprobar si el camino para ese nodo es mejor usando el coste g como baremo. Un coste g menor significa que éste es un mejor camino. Si es así, cambiar el padre del nodo actual al nodo adyacente y recalcular g y f del cuadro.

d) Si se añade el nodo meta a la lista abierta el camino ha sido encontrado, no obstante en el caso de que la lista abierta esté vacía no hay camino posible entre ambos nodos. Finalmente, si se ha incluido el nodo meta en la lista abierta generamos el camino encontrado buscando el padre de cada nodo empezando por la meta y finalizando en el origen.

Ejemplo

A continuación se muestra un ejemplo del algoritmo A* con el siguiente grafo de 5 nodos :

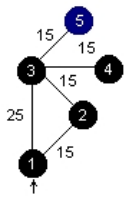


Figura 4.14: Ejemplo algoritmo A*, grafo original

Utilizamos como nodo inicial el nodo 1, y como nodo meta el nodo 5. El algoritmo debe darnos como resultado un camino óptimo entre ambos nodos. Cabe reseñar que a diferencia del algoritmo de Dijkstra, el A* no nos asegura como resultado el camino más corto entre ambos nodos, pero sí aproximarse lo máximo posible dando una solución 'buena' y más rápida que Dijkstra. La eficiencia en A* depende de la función heurística empleada, que en este caso es la distancia euclídea.

Inicialmente ningún nodo tiene asignado un nodo padre. No obstante, al finalizar el algoritmo sólo los nodos explorados tendrán asignado un nodo padre. Al comienzo la lista cerrada estará vacía y sólo la lista abierta contendrá el nodo origen número 1.

Movemos el nodo origen (1) a la lista cerrada y los adyacentes (2 y 3) a la lista abierta. Hacemos al nodo actual padre de sus adyacentes y almacenamos sus costes f, g y h. El valor f será la suma entre g (coste del camino calculado desde el origen hasta el nodo actual) y h (función heurística, distancia estimada desde el nodo actual a la meta).

Como ninguno de los nodos adyacentes estaba anteriormente en la lista abierta no comprobamos nada más y pasamos a la siguiente vuelta del bucle. El estado actual del grafo se muestra en la primera iteración de la figura 4.15:

Elegimos el nodo 2 como nodo con menor f ya que su g es de 15 y esta es menor que la g del nodo 3 que es de 25. Además, como f es la suma entre g y h, el coste estimado desde el nodo 2 hasta el nodo meta es menor que la del nodo 3 ya que este último está más alejado.

Desde el nodo actual tenemos como nodos adyacentes que no estén ya en la lista cerrada únicamente al nodo 3. Hacemos al nodo 2 padre de este nodo, y calculamos sus funciones f, g y h. Respecto al nodo 3, como ya estaba en la lista abierta, debemos comprobar si podría existir un camino más corto hacia este.

Para ello comprobamos que el coste g de llegar al nodo 3 desde el nodo 2 , es mayor que yendo directamente desde el nodo 1. Por lo que cambiamos el padre del nodo 3 al nodo 1 y recalculamos sus funciones f y g. Movemos el nodo 2 a la lista cerrada eliminándolo de la lisa abierta. El estado del grafo se muestra en la figura 4.15 para la segunda iteración:

Mantenemos el nodo 3 como nodo actual e incluimos en la lista abierta a los nodos adyacentes 4 y 5. Al incluir el nodo 5 en la lista abierta y asignar al nodo actual como su padre conseguimos una de las condiciones de final del algoritmo.

Finalmente, para encontrar el camino final buscamos el padre del nodo 5 que es el nodo 3, y a su vez el padre de este es el nodo origen , por lo que tenemos como camino resultante el 1,3,5 que en este caso es el camino mínimo entre el nodo origen y la meta.

4.4. Ejecución del plan

Una vez planificada la ruta por cualquiera de los dos algoritmos implementados debemos enviar las órdenes correspondientes al robot para recorrer la ruta creada.

Tanto por A* como por Dijkstra se consigue una ruta desde la posición del robot hasta la indicada por el usuario como meta. Esta ruta es una secuencia de nodos del grafo de visibilidad, implementada como un vector de nodos. Este vector se vacía cada vez que se inicia una etapa de planificación y se rellena con la ruta entre el robot y la meta siempre y cuando haya sido posible establecer una ruta entre ambas posiciones. A este vector lo llamaremos *vector de trayectoria*.

El movimiento del robot se controla con un hilo de ejecución paralelo al de creación del grafo. Para realizar dicho movimiento, la interfaz existente solo nos permite enviar órdenes respecto a la velocidad de traslación (v) y de rotación (w) de los motores. Recorriendo el vector de nodos de la trayectoria implementamos un algoritmo para conseguir que mediante una serie de tramos rectos el robot se vaya posicionando en cada uno de los nodos del vector y se posicione finalmente en la meta indicada.

Utilizamos un control basado en la posición del robot, por lo que debemos conocer la posición de este respecto del mapa en todo momento y corregir errores en el desplazamiento para evitar que el robot se desvíe de la trayectoria correcta. Dado que únicamente podemos enviar órdenes de velocidad y giro al robot y que existen márgenes de error

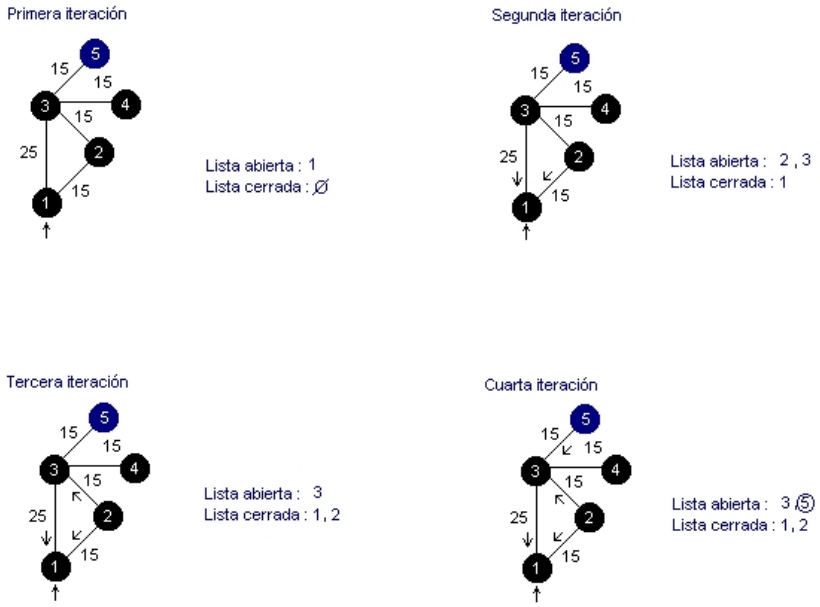


Figura 4.15: Ejemplo algoritmo A*, iteraciones.

que pueden hacer al robot perder la trayectoria se implementa un método de control de errores durante el desplazamiento.

Para explicar este control es necesario entender cual es la lógica para recorrer todos los nodos del vector de trayectoria. El siguiente proceso se repite por cada uno de los tramos que componen dicho vector, es decir, la trayectoria planificada.

Por cada uno de los nodos desde el nodo inicial hasta el nodo meta se ejecuta la siguiente secuencia:

1. Enviar al robot la orden de rotar hasta que el frontal del robot y el nodo siguiente estén en frente uno de otro. La dirección de la rotación será la que necesite menos giro para alcanzar el ángulo deseado.

2. Detiene la rotación del robot.

3. Avanza hasta el nodo siguiente reduciendo la velocidad según el robot se vaya acercando a este nodo.

4. Detiene el avance del robot.

Las coordenadas de la posición del robot nos las dan sus odómetros por dos variables globales llamadas robot[0] y robot[1]. El ángulo de giro respecto al eje X nos lo da una tercera variable robot[2], que nos ofrece dicho ángulo en radianes.

El control de errores durante la trayectoria antes mencionado pretende paliar la poca exactitud de los giros y de las aceleraciones del robot. Teniendo en cuenta que el algoritmo de desplazamiento se repite por cada par de nodos, podemos explicar este método con el trayecto desde un nodo hasta otro recorriendo una arista que los une. Esta arista sería el recorrido ideal para el robot , ya que es el camino más corto. No obstante durante la rotación para alinearlo con el nodo destino, podemos encontrar errores que hagan desviarse al robot desde el comienzo.

Esta primera rotación contiene un margen de error, ya que no podemos decirle al robot que se detenga justo cuando forme un ángulo de, por ejemplo, 75.34748 grados con el eje X. Este margen viene dado porque la orden que se le da al robot es que gire en el sentido en el que menos le cueste enfrentarse al nodo siguiente hasta que forme un ángulo mayor que (α + margen)y menor que (α - margen), donde α es el ángulo que forma el robot con el nodo destino.

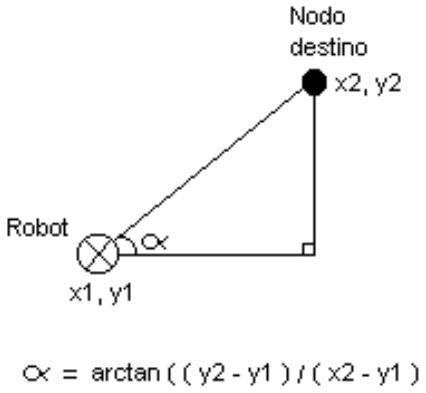


Figura 4.16: Ángulo formado por el robot y el nodo a alcanzar

Por esto, una vez que iniciamos el tramo rectilíneo, debemos recalcular el ángulo formado por el robot y el nodo siguiente para recuperar durante la trayectoria la orientación respecto a las coordenadas destino. Si detectamos que el robot ha perdido la alineación con el nodo destino giramos levemente la trayectoria para conseguir que el robot vuelva a quedar enfrentado con este nodo. De esta forma obtenemos un leve movimiento de cabeceo en la trayectoria, pero que a la vez evita que el robot se desoriente.

La velocidad de avance del robot es diferente según la distancia que le separe del nodo. Si esta distancia está fuera de cierto umbral definido como constante, la velocidad permanecerá invariable. No obstante si esta distancia está dentro del umbral, reducirá la velocidad de manera lenta y constante hasta situarse en menos de un tercio de su velocidad anterior.

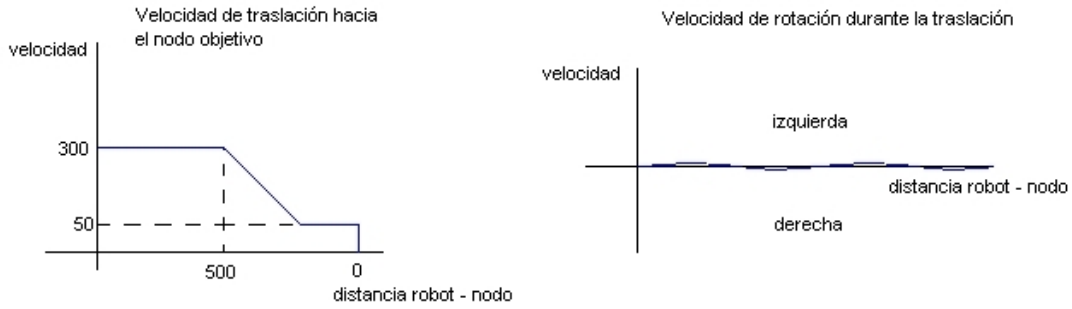


Figura 4.17: Gráficas de velocidad de traslación y rotación del robot durante el trayecto hacia el nodo objetivo.

4.5. Interfaz gráfica

Una vez descritas las tres etapas de las que consta este proyecto : creación del grafo de visibilidad, planificación y ejecución del plan; se explica la interfaz gráfica implementada.

La visualización en este proyecto se realiza mediante un hilo de ejecución paralelo a los de construcción del grafo y ejecución del plan. En el centro de la ventana de ejecución (figura 4.18), aparece una pantalla en donde se puede visualizar además del robot, cada uno de los segmentos que intervienen en el proceso, sean obstáculos del mapa, grafo de visibilidad o zonas de seguridad.

Todos estos segmentos se pueden visualizar de manera independiente o conjunta según el tipo. Así podemos tener una visualización de un mismo mapa pero con diferentes combinaciones como por ejemplo : obstáculos y zonas de seguridad, obstáculos y grafo de visibilidad ... etc. Los botones dispuestos en la parte inferior izquierda de la interfaz se encargan de esta visualización :

Mundo : Muestra de color rojo los segmentos de tipo obstáculo leídos del mapa del entorno.

Seguridad : Muestra de color magenta los segmentos que forman la zona de seguridad de cada uno de los obstáculos.

Visibilidad : Muestra de color verde los segmentos que forman el grafo de visibilidad

Ruta : Muestra de color azul la ruta planificada desde la posición del robot a la posición destino indicada por el usuario.

Trayecto : Muestra un punto por segundo en la posición en donde se encuentre el robot. Esta función nos sirve para visualizar la trayectoria descrita durante la ejecución del plan.

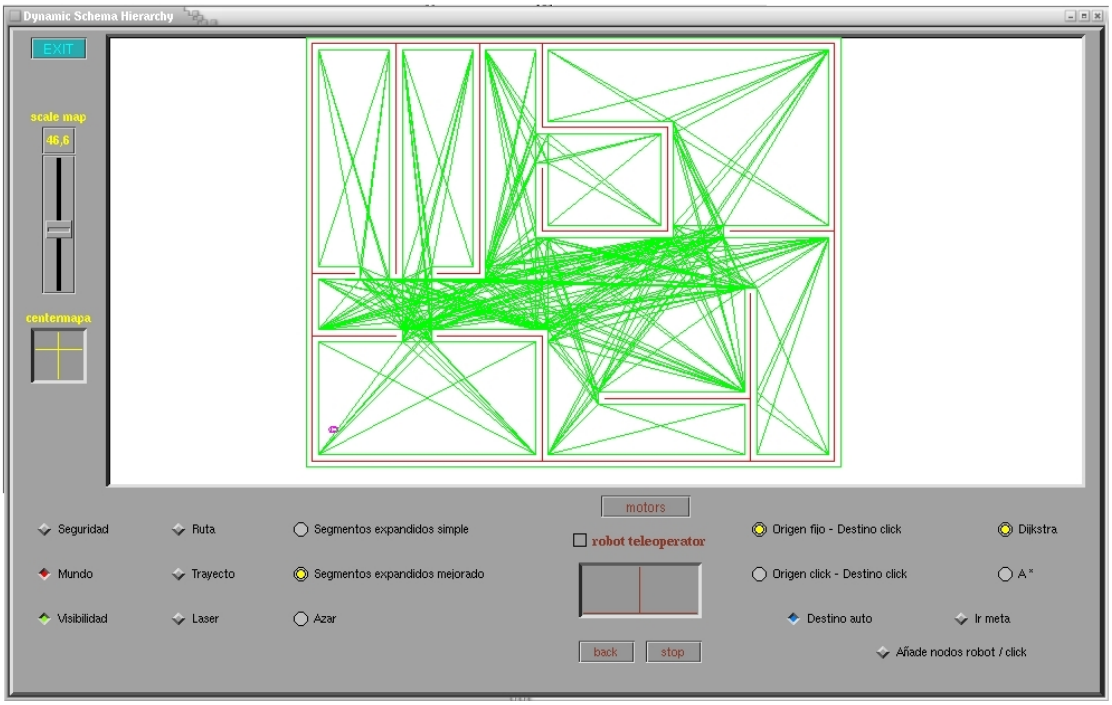


Figura 4.18: Interfaz de la aplicación

Siguiendo con el grupo de visualización, a la izquierda de la ventana encontramos una barra vertical para hacer **zoom** al mapa visualizado. Debajo de este tenemos un **eje de visualización** para mover a pequeños saltos la pantalla del mapa según la posición del click sobre este eje.

Además de poder visualizar cada una de estas funciones en pantalla, podemos elegir el método de creación del grafo a partir del mapa leído. Tenemos a la derecha del grupo de botones de visualización, tres botones con la descripción de cada uno de los métodos de creación : simple, mejorado y azar. Todos ellos explicados al comienzo de este mismo capítulo.

Una vez que tenemos el grafo creado podemos pasar realizar la planificación de la ruta desde la posición del robot hasta la posición definida por usuario.

Para definir la posición destino hacia donde debe planificarse una ruta desde el robot, primero se ha de optar por uno de los métodos de planificación implementados : Dijkstra o A*.

Una vez que este método ha sido elegido el usuario debe hacer click con el ratón en un punto del mapa. Este punto será el destino hacia el que se realiza la planificación desde el robot. A continuación aparecerá un aspa roja en el punto marcado por el usuario y, si el botón 'Ruta' está marcado, se describirá mediante trazos rectilíneos el trayecto que ha de seguir el robot hasta llegar a la posición del aspa.

Podemos realizar pruebas de planificación antes de comenzar a mover el robot, optando por elegir el origen y el destino de la ruta mediante el ratón (botón 'Origen click -Destino click'), o mantener como origen la posición del robot (botón 'Origen fijo - Destino click').

Tan solo queda hacer que el robot recorra la ruta planificada, para ello tenemos la opción de dirigir al robot manual o automáticamente como se muestra en el siguiente grupo.

En la parte inferior central de la pantalla se implementa un **joystick** para poder conducir el robot por el mapa sin tener que seguir la ruta planificada. Esto puede ser muy útil para posicionar el robot en un punto de partida sin planificar una ruta hacia ese punto recorriendo el grafo de visibilidad. Para poder activar este joystick debe estar pulsado el botón ' **robot teleoperator**' y el botón '**motors**' que activa los motores del robot. En la parte inferior de este joystick disponemos de dos botones para dar marcha atrás al robot('**back**') o detenerlo en seco ('**stop**').

Si el botón 'motors' está pulsado y pulsamos '**Ir meta**', el robot comenzará a recorrer la ruta planificada desde el nodo origen y pasando por todos los nodos intermedios hasta llegar a la posición de meta.

Para visualizar el movimiento del robot durante la traslación en pequeños instantes de tiempo disponemos de un vector circular de 20 posiciones, cuyas posiciones son de tipo punto, en el que vamos informando de la posición del robot cada segundo de ejecución. El resultado de mostrar el contenido de este vector es una estela de puntos que indican las últimas posiciones del robot durante el desplazamiento. Como ya se indicó en el grupo de visualización para visualizar estos puntos es preciso que esté pulsado el botón 'Trayecto' en la parte inferior izquierda de la ventana.

Capítulo 5

Experimentación

Una vez expuesta la implementación informática desarrollada, se han realizado diversos experimentos y pruebas para calcular el rendimiento de los algoritmos utilizados y así poder compararlos.

Por orden de ejecución, se expondrán en este capítulo los cálculos realizados durante las diversas etapas, indicando las correspondientes valoraciones y conclusiones en el capítulo 6.

5.1. Ejecución típica

El equipo utilizado para el cálculo de medidas durante la fase de experimentación fue un Pentium 4 a 2.6 MHz con 512 Megas de RAM y S.O Linux Debian kernel 2.6.8

Sobre este equipo se lanza la plataforma JDE necesaria para este proyecto y descrita en el capítulo 2, compuesta por el simulador SRIsim y el servidor otos conectados por el puerto 8101. Ejecutamos nuestro cliente JDE, que se conecta al servidor otos por el puerto 3001 y ya tenemos la arquitectura software necesaria para realizar las pruebas y los cálculos sobre la ejecución. Esta estructura de cliente-servidor-simulador se mostró en la figura 3.3.

Una vez que tenemos conectado nuestro cliente JDE al servidor otos, y éste a su vez con el simulador SRIsim, podemos ejecutar nuestra aplicación para que gobierne la navegación del robot de un punto a otro.

Recordamos que las etapas que forman la ejecución son por este orden : leer el mapa, construir las zonas de seguridad, construir el grafo, planificación de ruta y ejecución del plan navegando por la ruta creada.

El mapa leído es el de la planta 1 del edificio departamental II, y más concretamente el ala izquierda donde se encuentra el laboratorio del grupo de robótica. En la figura 5.1 se puede apreciar el mapa de segmentos (a), las zonas de seguridad (b) y el grafo creado (c).

Nuestra aplicación analiza el fichero con la descripción del mapa en forma de colección de segmentos. Como puede apreciarse en la figura (b) se forma una isla de segmentos de visibilidad en la parte central del mapa. Esta zona corresponde al patio central de

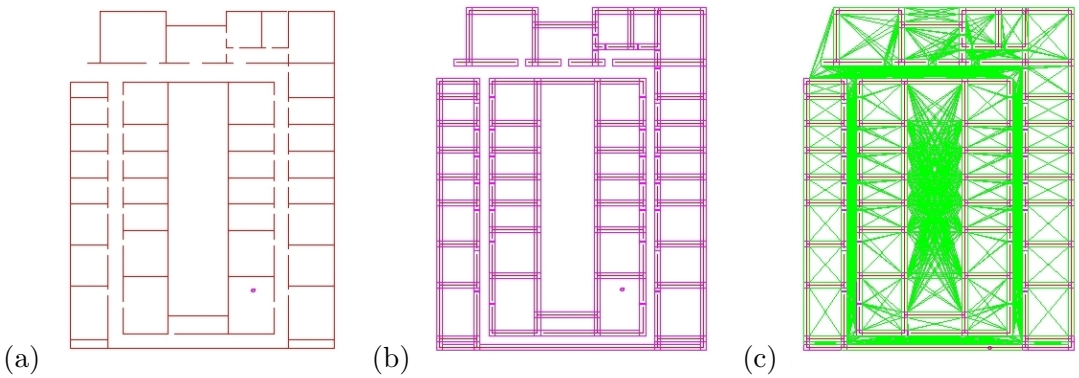


Figura 5.1: Departamental II, mapa (a), zonas de seguridad (b) y grafo de visibilidad por el método mejorado (c)

la planta a la que el robot no podrá acceder. Aun así, como el mapa sólo incluye los obstáculos y no las zonas accesibles, el grafo cubre esta zona aunque nunca se pueda llegar a ellas.

Probamos a generar para este mismo mapa el grafo de visibilidad por los tres métodos implementados. El resultado se muestra en la figura 5.2.

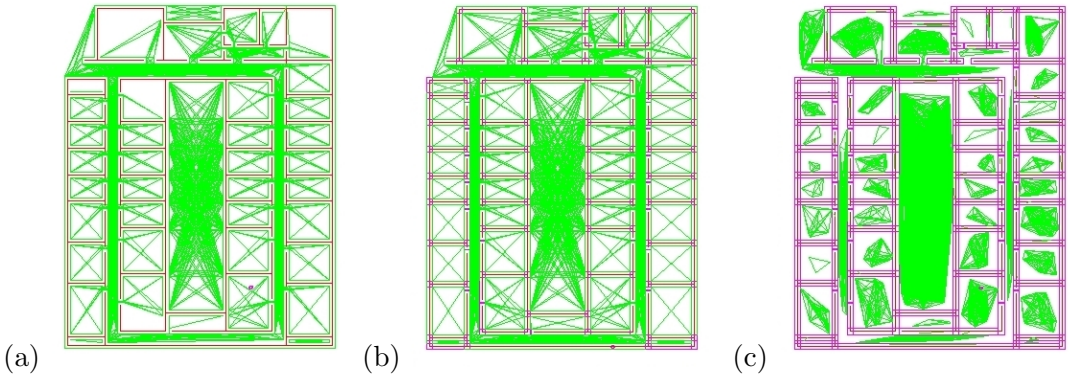


Figura 5.2: Grafos de visibilidad por el método simple (a), mejorado (b) y al azar (c)

Comparamos las tres figuras y vemos que, por ejemplo, entre el método simple y el mejorado existen pocas diferencias. Por ejemplo difieren en las dos esquinas superiores del mapa. En estas habitaciones existen más segmentos de visibilidad en el método mejorado que en el simple, tal y como pretendimos durante la implementación.

Si comparamos la figura generada al azar con los otros dos métodos, observamos que se crean muchas más islas, lo que puede llegar a hacer que durante la etapa de planificación no podamos llevar el robot a estas zonas. Sin embargo, podemos ver que en esta etapa de creación del grafo los tres métodos consiguen acceder a casi todas las

zonas del mapa. Esta es una circunstancia particular para el método simple, ya que una de las limitaciones que encontramos durante su implementación fue precisamente la accesibilidad a zonas del mapa como se pudo ver en la figura 4.7. En este caso la forma de este mapa en concreto hace que el método simple pueda crear segmentos de visibilidad en todas las habitaciones del mapa. Esto se debe a que en el fichero del mapa las paredes que limitan el interior de la planta con el exterior no están construidas como un único segmento, si no por el alineamiento de los distintos segmentos de las habitaciones que limitan con el exterior.

Para probar los algoritmos de planificación, utilizamos el grafo generado por el método mejorado. Optamos por probar una planificación por el método de Dijkstra en este grafo, de una ruta desde el laboratorio de robótica hasta la sala de reuniones obteniendo la ruta que se muestra en la figura 5.3.

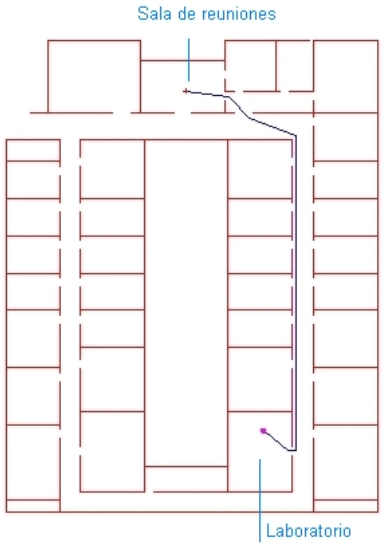


Figura 5.3: Ruta desde el laboratorio de robótica hasta la sala de reuniones. Método Dijkstra

Como podemos ver en la ruta planificada, el robot no roza en ningún momento las paredes del mapa. Esto se debe a las zonas de seguridad implementadas previamente a la creación del grafo, que hacen que el robot mantenga una distancia mínima prudencial de los obstáculos.

Sobre la trayectoria generada en la figura 5.3, podemos comprobar que es la más corta de todas las posibles. Si Dijkstra no nos garantizara el camino más corto, el algoritmo podría haber planificado girar hacia la derecha al salir de la primera habitación y desplazarse por los pasillos inferior e izquierdo para poder acercarse a la meta. Sin embargo haber utilizado este método hace que obtengamos el camino más corto entre el robot y el destino.

Merece consideración que este mapa ha sido diseñado con todas las puertas abiertas para poder experimentar con las distintas zonas del mapa durante esta etapa. Como consecuencia de esto, si en la realidad existen habitaciones que están cerradas y esta circunstancia no ha sido indicada en el fichero del mapa, el robot no podrá saber que no es posible acceder a dicha habitación.

Seguidamente activamos los motores del robot y el esquema que hace que éste se desplace siguiendo la ruta planificada. En la figura 5.4 se muestra el trayecto que describe el robot durante la ejecución.

Podemos diferenciar en esta figura cómo el destino de la ruta se muestra con una cruz de color rojo. Así mismo, durante el desplazamiento del robot (en color azul) , mostramos una estela de puntos negros para indicar la trayectoria seguida por el robot.

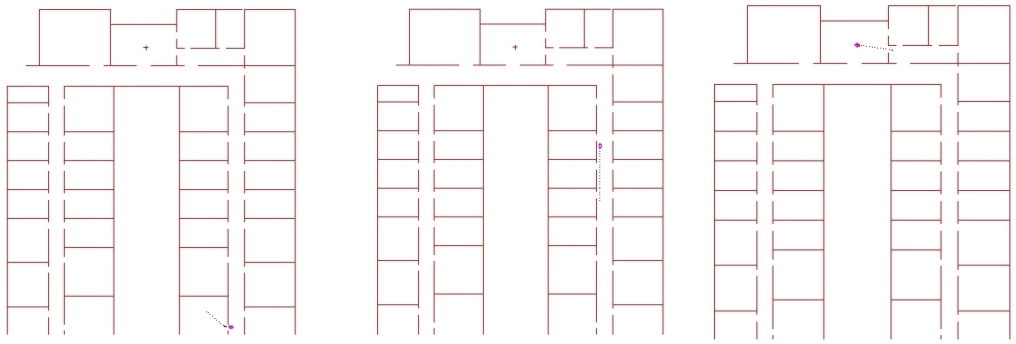


Figura 5.4: Trayecto de la ruta planificada desde el laboratorio de robótica hasta la sala de reuniones.

La interfaz gráfica implementada nos permite modificar el método de planificación en tiempo de ejecución. Por ello, una vez posicionado el robot en la sala de reuniones, cambiamos el método de planificación por el de A*. En esta ocasión, probamos a trazar una ruta desde esta sala de reuniones hasta el cuarto de los servidores e impresoras. En la figura 5.5 se visualiza la nueva planificación por A*.

Las rutas planificadas por uno u otro método tienen el mismo aspecto indistintamente y el algoritmo de pilotaje las utiliza. No obstante pero existen diferencias en el tiempo de planificación y en la garantía de obtener el camino mínimo con cada uno de ellos.

Podemos ver en la figura 5.6 que modificando ligeramente la posición destino el algoritmo es capaz de realizar una planificación por zonas distintas del mapa para conseguir, o al menos acercarse, al camino más corto posible.

5.2. Métodos de creación del grafo

En el capítulo 4 vimos tres técnicas de construcción del grafo : simple, mejorado y azar. En esta sección mostramos las pruebas realizadas sobre tres mapas de diferentes

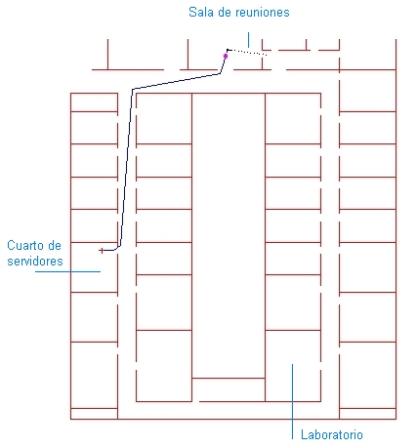


Figura 5.5: Ruta desde la sala de reuniones hasta el cuarto de servidores. Método A*

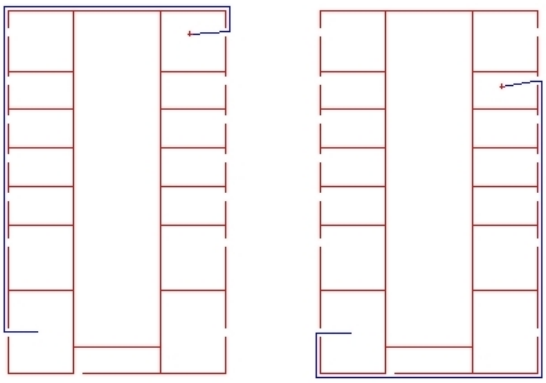


Figura 5.6: Rutas planificadas desde un mismo destino hasta habitaciones contiguas entre si.

Mapa	Método	Nodos	Seg. obstáculo	Seg. seguridad	Seg. visibilidad
Pequeño	Simple	66	19	76	332
	Mejorado	118			804
	Azar	76			360
Mediano	Simple	187	65	260	1568
	Mejorado	388			6204
	Azar	260			5267
Grande	Simple	481	167	668	6283
	Mejorado	1008			21025
	Azar	668			7739

Cuadro 5.1: Número de segmentos por método y mapa

tamaños y así poder comprobar en ellos la eficiencia de los métodos empleados. Los tres mapas son los que se muestran en la figura 5.7, uno pequeño, otro mediano y otro grande.

El mapa considerado como grande ya ha sido utilizado en esta misma sección, el mediano corresponde a las habitaciones interiores del mapa grande, y el pequeño simula la disposición de las paredes y habitaciones de una casa particular.

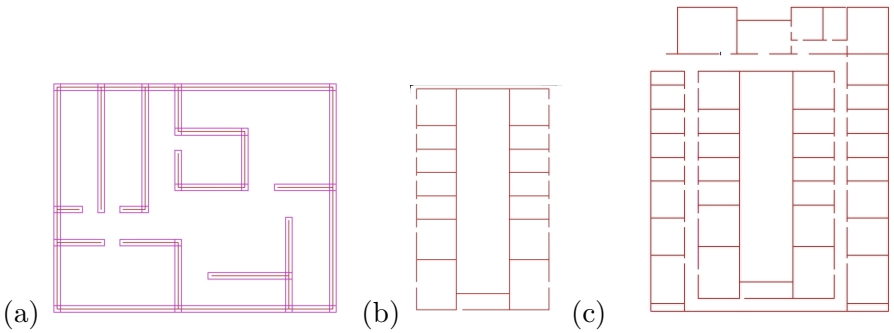


Figura 5.7: Mapas pequeño (a), mediano (b) y grande [c]

En la tabla 5.1 se muestra el número de nodos y de segmentos de obstáculo, seguridad y visibilidad generados por cada uno de los métodos implementados. Como podemos ver en esta tabla, el método utilizado sólo varía el número de segmentos de visibilidad del mapa, ya que el número de obstáculos y de segmentos de seguridad pertenece a una proceso previo al de creación del grafo. El método mejorado es el que más segmentos de visibilidad crea ya que es el método que más nodos genera sobre el mapa. De hecho, en el mapa grande se han creado casi tres veces más segmentos de visibilidad con el método mejorado más que con el azar.

El número de nodos para el método simple siempre será igual o menor que el número de segmentos de seguridad. Esto se debe a que incluimos durante la implementación una condición por la cual no creamos más de un nodo con las mismas coordenadas. Si

Mapa	Método	Nodos	Tiempo de creación del grafo (seg.)
Pequeño	Simple	66	0.08
	Mejorado	118	0.38
	Azar	76	0.12
Mediano	Simple	187	0.75
	Mejorado	388	2.21
	Azar	260	1.18
Grande	Simple	481	3.9
	Mejorado	1008	11.6
	Azar	668	5.03

Cuadro 5.2: Tiempo de creación del grafo por método y mapa

para este método existen menos nodos que segmentos de seguridad es porque hay algún vértice de estos segmentos que coincide con algún otro vértice de otra zona de seguridad.

Sin embargo para el método al azar, este número de nodos es idéntico al número de segmentos de seguridad. Esto es porque el heurístico empleado crea 4 nodos por cada segmento obstáculo. Esta proporción es modificable en el código como se comentó en el capítulo 4.

También podemos observar en los datos recogidos, cómo el método al azar genera más segmentos de visibilidad que el método simple. Esto se debe a que, aunque estamos creando el mismo número de nodos con ambos métodos, la disposición es totalmente distinta. Con el método simple sabíamos que por cada obstáculo que hubiera en el mapa se iban a crear 4 nodos en los cuatro vértices de sus segmentos de seguridad, y por tanto las diagonales no iban a ser visibles por atravesar el obstáculo. Con el método al azar esto ya no tiene por qué ocurrir, ya que aunque seguimos creando 4 nodos por obstáculo, éstos tienen mayor visibilidad entre ellos debido a su cercanía unos con otros. Esto también influye en el tiempo de creación del grafo y en la planificación de la ruta. Indicar que esta distribución aleatoria hace que el número de segmentos de visibilidad creados varíe en cada ejecución, mientras que en los métodos simple y mejorado este número permanece fijo.

Medimos los tiempos para cada uno de los métodos de creación del grafo implementado. En la tabla 5.2 se muestra la relación entre el número de nodos de cada uno de los mapas y el tiempo de ejecución necesario para su creación. Los tiempos indicados en esta tabla corresponden a la media de 5 medidas por cada uno de los casos que se indican.

Como se puede comprobar en los tres mapas probados, el método al azar es más rápido que el mejorado, pero más lento que el simple. Esto se debe a que, aunque teniendo el mismo número de nodos que con el método simple, los nodos suelen caer muy cercanos unos a otros por lo que sus conexiones son mayores. Esto equivale a más segmentos de visibilidad y por tanto mayor tiempo de procesado de los datos. El método mejorado consume más tiempo que cualquier otro debido al gran número de segmentos de visibilidad y de nodos que genera. El tiempo de creación es proporcional al número de

segmentos de visibilidad y éste influye igualmente en el tiempo que se tarda en planificar una ruta, como veremos en la tabla 5.3.

En todo caso el tiempo obtenido en cada caso es directamente proporcional al número de nodos generado.

5.3. Métodos de planificación de la ruta

En esta sección mostramos las pruebas realizadas con cada uno de los métodos de planificación, Dijkstra y A*. De las pruebas realizadas hemos seleccionado dos rutas con un mismo origen y destino, generadas por cada uno de los métodos y sobre cada uno de los mapas. Se muestran las rutas creadas por métodos distintos en las figuras 5.8 y 5.9.

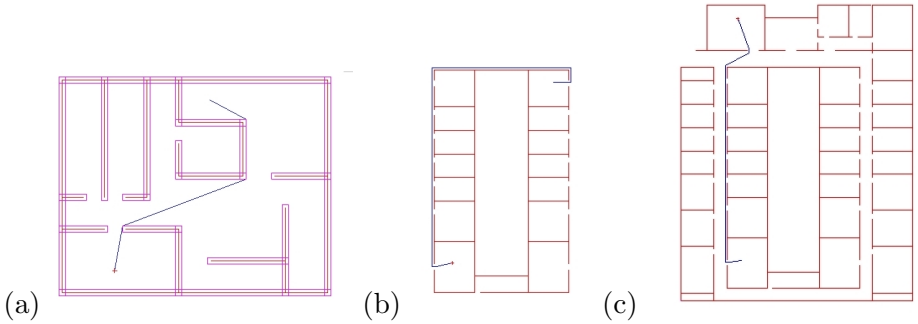


Figura 5.8: Rutas en mapas pequeño (a), mediano (b) y grande (c) por el método de Dijkstra

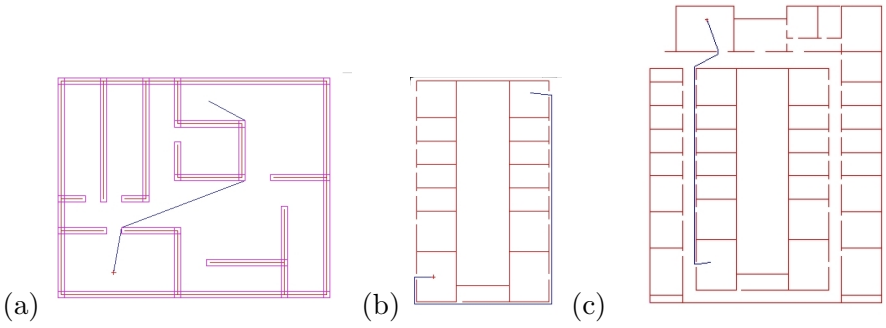


Figura 5.9: Rutas en mapas pequeño (a), mediano (b) y grande (c) por el método A*

Como podemos ver en las figuras 5.8(b) y 5.9(b), Dijkstra y A* crean rutas idénticas en el mapa pequeño y grande, pero diferentes en el mediano. Éste es un claro ejemplo en el que Dijkstra genera la ruta más corta y A* un camino que se aproxima al óptimo hasta la meta. Esto se debe a las características propias de cada uno de los algoritmos empleados,

Mapa	Método	Tiempo de creación de la ruta (seg.)
Pequeño	Dijkstra	0.45
Pequeño	A*	0.32
Mediano	Dijkstra	1.90
Mediano	A*	0.88
Grande	Dijkstra	2.42
Grande	A*	1.70

Cuadro 5.3: Tiempo de planificación por método y mapa

que en muchas ocasiones coincidirán en sus rutas y otras harán una planificación distinta pero igual de aceptable.

Los tiempos calculados para cada una de estas 6 rutas se muestran en la tabla 5.3 y corresponden a la media de 5 medidas por cada uno de los casos que se indican. Comprobamos como el método A* es regularmente más rápido que Dijkstra planificando rutas idénticas como en los casos de los mapas pequeño y grande. Esto también ocurre aunque se planifiquen rutas distintas como ocurre con el mapa mediano. El tiempo de la planificación con Dijsktra está acotado entre 1'4 y 2'2 veces el tiempo de planificación A*. Los tiempos medidos varian dependiendo del número de nodos que contenga la ruta, pero la proporción entre ambos métodos suele ser la indicada anteriormente siendo aproximadamente el doble para Dijkstra que para A*.

Capítulo 6

Conclusiones y líneas futuras

Una vez descrita la aplicación de navegación y los experimentos realizados en ella, repasaremos los objetivos marcados al inicio del proyecto así como los requisitos y el grado de cumplimento de los mismos. Además, en este capítulo se explicarán las conclusiones a las que hemos llegado durante la fase de experimentación y varias líneas de trabajo propuestas para futuros proyectos.

6.1. Consecución de objetivos y requisitos

Respecto de los objetivos planteados en el capítulo 2 podemos concluir que se han realizado en su totalidad. El objetivo de diseñar y programar una aplicación de navegación se ha satisfecho como se ha puesto de manifiesto en las pruebas realizadas.

El objetivo principal se articula en 3 subobjetivos, los cuales se enumeran a continuación junto con un breve resumen de cómo los hemos satisfecho.

1. **Creación del grafo de visibilidad** : Se han implementado tres métodos para crear distintos grafos de visibilidad : Método simple, mejorado y al azar. Cada uno de ellos ofrece una solución distinta y aplicable a diferentes entornos. La implementación de varios métodos nos ha permitido experimentar con cada uno de ellos y buscar la mejor solución disponible para un mismo problema.

   El método simple crea un grafo que sólo tiene en cuenta como nodos del grafo los vértices de los segmentos de seguridad creados a una distancia definida de los segmentos de seguridad.

   Con el método mejorado, cuya implementación se definió en el capítulo 4.2.2, añadimos al conjunto de nodos del método simple las intersecciones de los segmentos de seguridad, resolviendo así el problema de accesibilidad a todas las zonas del mapa.

   Por último implementamos un método de generación del grafo de visibilidad disponiendo aleatoriamente los nodos sobre el mapa. Con este método pretendimos mejorar la velocidad de ejecución del método mejorado y resolver de una manera distinta el problema de accesibilidad del método simple, tal y como fue explicado en el capítulo 4.2.3.

2. **Planificación de la ruta** : Hemos implementado dos métodos de planificación de ruta para poder comparar y experimentar con ambos. Inicialmente se implementó el algoritmo de Dijkstra (4.3.1), que encuentra las rutas más cortas entre un nodo y el resto. Posteriormente se añadió el método A* (4.3.2), por ser más rápido que el anterior y que encuentra heurísticamente una ruta óptima entre dos nodos cualesquiera del grafo.

3. **Ejecución de la ruta planificada** : Hemos podido observar durante la experimentación cómo el método implementado para que recorrer la ruta planificada hace que el robot pase por cada uno de los nodos que forman la ruta hasta concluir en el nodo final o meta. Este método se basa en la repetición de una secuencia de órdenes de giros y desplazamientos desde un nodo a su siguiente en la ruta planificada hasta posicionarse en el destino marcado por el usuario. Este control iterativo es el método más sencillo de todos los posibles, ya que el objetivo de esta etapa es simplemente llevar al robot por la ruta planificada.

En la sección 2.2 se definió una serie de requisitos obligatorios que condicionaban el desarrollo. A continuación se enumera la manera en la que hemos cumplido con cada uno de ellos :

El software desarrollado sobre la plataforma JDE : El proyecto se ha servido de la plataforma en cuanto a la interfaz inicial aportando una nueva funcionalidad en cuanto al tipo de navegación y la búsqueda de rutas. Existen distintos tipos de navegación global desarrollados en diversos proyectos del grupo [Isado, 2005], y que están creados sobre esta misma plataforma, la cual supone una herramienta muy útil en este tipo de trabajos.

Ejecución vivaz : Este resultado depende mucho del tamaño del mapa de entrada, de los métodos empleados y de la potencia de la máquina en la que se ejecute la aplicación. El algoritmo de creación del grafo por definición tiene una complejidad cuadrática y por tanto es inevitable que el tiempo de ejecución se dispare para mapas de cierto tamaño como el del departamental II. Aun así consideramos que los tiempos recogidos durante la experimentación son aceptables ya que no suponen tiempos de espera demasiado largos teniendo en cuenta la complejidad del algoritmo empleado.

Planificación de un camino óptimo : Los dos métodos implementados generan una ruta que consideramos óptima ya que siempre obtenemos un camino cercano o igual al camino de distancia mínima entre el robot y el destino marcado por el usuario.

Implementación en C y sistema operativo Linux : Se ha utilizado este lenguaje para implementar la lógica de todos los objetivos planteados. En particular se ha utilizado una versión la versión 3.3.2 del compilador gcc para este lenguaje en concreto. Respecto al sistema operativo se ha probado con éxito la implementación de este proyecto en las distribuciones de Linux : Debian y Mandrake 9.2.

6.2. Conclusiones sobre los métodos empleados

Respecto al método de navegación utilizando el grafo de visibilidad, hemos podido observar que la complejidad hace que su ejecución sea relativamente lenta para escenarios medianos o grandes. En este tipo de escenarios, con aproximadamente unos 60 segmentos de obstáculo en el mapa, hemos comprobado que el proceso tarda varios segundos en crear el grafo. Por ello, concluimos que este método debe utilizarse en entornos con pocos obstáculos o con tiempo disponible para poder procesar el grafo completo y planificar una ruta posteriormente.

Sobre los métodos de creación de los nodos del grafo (simple, mejorado o al azar), no podemos concluir que uno de ellos sea mejor que el resto, ya que cada uno de ellos resulta más conveniente que los otros en determinados entornos y tendrá usos diferentes según el mapa de entrada. Hemos comprobado que el método mejorado puede alcanzar más zonas que el método simple, pero que esto tiene un coste computacional al comparar muchos más nodos entre sí. Sin embargo podemos comprobar que en ciertos escenarios como el del mapa del departamental II, el método simple es capaz de llegar a todos las zonas del mapa donde llega el mejorado generando un grafo más pequeño y más sencillo de utilizar. Esto se debe a que una serie de paredes del mapa no están definidas en el mapa como un solo segmento sino como varios consecutivos y pertenecientes cada uno de ellos a una habitación distinta. Esto hace que se generen nodos del grafo en cada una de las habitaciones y que por tanto el robot pueda planificar una ruta hacia estas zonas.

El tercer método implementado, el del azar, ha resultado más lento de lo que inicialmente planteamos. Esto se debe a que este método, aunque el número de nodos es el mismo que en el simple (ya que lo hemos dispuesto así durante la implementación), el número de estos nodos que son visibles entre sí aumenta. Esto se debe a que estos nodos ya no están dispuestos sobre los 4 extremos de cada obstáculo y por tanto de cada 4 nodos que teníamos por grafo existían al menos 2 uniones que no iban a ser posibles entre ellos, ya que hubieran cortado al obstáculo. Esta regla no existe en el método al azar por la disposición aleatoria de los nodos que hacen que se genere un mayor número de segmentos de visibilidad.

De los métodos implementados para la fase de planificación, Dijkstra y A*, podemos concluir que para la aplicación que nos ocupa este proyecto es más eficiente A* que Dijkstra debido a su rapidez. Dijkstra calcula demasiadas rutas que no se van a utilizar y por tanto tarda más tiempo que A* que solo calcula la ruta entre dos nodos del grafo.

Un escenario en el que Dijkstra nos podría ser mucho más útil que A*, sería implementar una opción para que el robot recorriera toda la superficie del mapa como si fuera, por ejemplo, un robot que limpiara la suciedad del suelo. En este caso, bastaría con una ejecución del algoritmo de Dijkstra para obtener las rutas más cortas desde el robot hasta *todos* los nodos conexos a su posición en el mapa. Estas rutas se ordenarían de menor a mayor coste y tendríamos una lista de nodos, que si la recorriéramos, pasaríamos por toda la superficie que ocupan los nodos conexos al nodo inicial.

6.3. Líneas de trabajo futuras

Una mejora al método del azar que pudiera resolver las excesivas conexiones que se hacen entre los nodos, sería que mientras se distribuyen los nodos por toda la superficie del mapa hubiera una lógica implementada que sólo aceptara distribuir un nodo en un punto concreto si este se encuentra como mínimo a una distancia definida anteriormente del nodo más cercano. Con esto lograríamos llegar a todas las partes del mapa y no tener nodos del grafo excesivamente juntos que disparen el número de segmentos de visibilidad. Este método se podría llamar *azar con repulsión*.

Por otra parte para ganar rapidez en la planificación de rutas, y más particularmente en el algoritmo de Dijkstra, se podría incluir una condición por la cual el algoritmo se detuviera en cuanto se calculara la ruta más corta hacia el nodo destino. Esto beneficiaría sobretodo a las rutas entre nodos más cercanos, pero obtendríamos tiempos semejantes a los actuales si se trazan rutas hacia los nodos más lejanos al robot.

Respecto a la ejecución de la planificación y el desplazamiento del robot se propone utilizar un método de navegación mediante lógica borrosa que suavizaría la trayectoria del robot.

Por último, para poder utilizar los métodos implementados para este proyecto en el robot físico es necesario incluir un sistema de localización que permita conocer la situación exacta en la que se encuentra el mismo. Esto se debe a que los odómetros tienen un determinado margen de error que hace que el robot pierda el conocimiento de su posición durante el desplazamiento.

Bibliografía

[Borenstein y Koren, 1989] J. Borenstein y Y. Koren. Real-time obstacle avoidance for fast mobile robots. *IEEE Journal of Robotics and Automation*, 1989.

[Cañas, 2002] Jose María Cañas. Dynamic schema hierarchies for an autonomous robot. *Universidad de Sevilla*, Noviembre 2002.

[Cañas, 2003] Jose María Cañas. Jerarquía dinámica de esquemas para la generación de comportamiento autónomo. *Tesis Doctoral*, Diciembre 2003.

[Cañas, 2004] Jose María Cañas. Manual de programación de robots con jde. *Universidad Rey Juan Calos*, pages 1–36, abril 2004.

[Cascales y Lucas, 2000] Bernardo Cascales y Pascual Lucas. Latex, una imprenta en sus manos. *AD*, 2000.

[Cormen, 1990] Rivest Cormen, Leiserson. Introduction to algorithms. *The MIT Press–Mc Graw Hill*, 1990.

[Crowley, 1985] James L. Crowley. Navigation for an intelligent mobile robot. *IEEE Journal of Robotics and Automation*, 1(1):31–41, March 1985.

[de Berg, 1987] M. de Berg. Computational geometry algorithms and applications. *Springer Verlag*, 1987.

[Dueñas, 2004] Mª Angeles Crespo Dueñas. Localización probabilística en un robot con visión local. *Universidad Rey Juan Carlos-PFC*, 2004.

[Feyrer y Zell, 2001] Tefan Feyrer y Andreas Zell. Robust real-time pursuit of persons with a mobile robot using multisensor fusion. *Wilhelm-Schickard-Institute, Department of Computer Architecture. University of Tübingen*, 2001.

[Ford, 1984] D.R Ford, L.R. y Fulkerson. Graphs and algorithms. *Wiley*, 1984.

[García y Bustos, 2001] M.C. García y P. Bustos. Complex behaviour generacion on autonomouse robots: A case study. *Instituto de Automática Industrial*, 2001.

[GSyC, 2004] GSyC. Tema de navegación del temario de robotica. *Universidad Rey Juan Carlos-PFC*, 2004.

[Isado, 2005] José Raúl Isado. Navegación global utilizando método del gradiente. *Universidad Rey Juan Carlos-PFC*, 2005.

[J. Mira, 1995] J. G. Boticario y F. J. Díez J. Mira, A. E. Delgado. Aspectos básicos de inteligencia artificial. *Sanz y Torres*, 1995.

[L.Moreno, 2004] B.L.Boada; D.Blanco; L.Moreno. Symbolic place recognition in voronoi-based maps by using hidden markov models. *Journal of Intelligent and Robotics Systems. Vol. 39*, 2004.

[Lobato, 2003] David Lobato. Evitación de obstáculos basada en ventana dinámica. *Universidad Rey Juan Carlos-PFC*, 2003.

[Mejías, 2003] Raúl Benitez Mejías. Sistema de localización basado en la detección de segmentos para robot móviles. *Universidad Rey Juan Carlos-PFC*, 2003.

[Palomino, 2004] Roberto Calvo Palomino. Comportamiento sigue persona con vision direccional. *Universidad Rey Juan Carlos-PFC*, 2004.

[Robotics, 2002] ActivMedia Robotics. Aria reference manual. *Technical Report version 1.1.10*, Noviembre 2002.

[Robotics, 2003] ActivMedia Robotics. Pioneer 3 pioneer 2 h8-series operations manual. *Technical Report version 3*, Agosto 2003.

[Serrano, 2004] Oscar Serrano Serrano. Localización mediante redes inalámbricas. *Universidad Rey Juan Carlos-PFC*, 2004.

[Worz, 1997] Christian Schlegel Jorg Illmann Heiko Jaberg Matthias Schuster Robert Worz. Vision based person tracking with a mobile robot. *Institute for Applied Knowledge Processing (FAW)*, 1997.