Trabajo Final
Sistema de navegación de interiores para plataformas autónomas móviles

Rafael Matías Capdevielle
Javier Valdazo Parnisari
Director: Gabriel Infante Lopez

29 de junio de 2007

Facultad de Matemática, Astronomía y Física
Universidad Nacional de Córdoba
Argentina

I.2.9.Autonomous vehicles

Resumen

El presente trabajo describe un sistema de navegación de interiores para plataformas autónomas moviles. El mismo intenta resolver el problema de trasladar una plataforma de un punto a otro en un ambiente cerrado. El trabajo contempla soluciones para tres problemas: 1. Encontrar un camino optimo entre dos puntos, 2. minimizar errores en el desplazamiento y 3. desplazar la plataforma en ambientes que puedan cambiar. Se presentan cinco algoritmos distintos para resolver el primer problema: dos algoritmos genéticos, un algoritmo dinámico, un algoritmo basado en Dijkstra y uno basado en A^*. Para el segundo problema se presenta una solución basada en primitivas de movimiento que intentan minimizar los errores de hardware. El trabajo muestra un algoritmo que usa las primitivas de movimiento y A^* para resolver el tercer problema. Finalmente, para cada una de las soluciones presentamos evaluaciones empiricas con simulaciones implementadas por nosotros y con una plataforma real.

Palabras clave: robótica, Planning, Genética, navegación, interiores.

Agradecimientos

A Miguel por darnos la oportunidad de trabajar con Omega, a el y su familia por la hospitalidad brindada.
Gracias a nuestras familias por confiar en nosotros y por todos los años de apoyo y paciencia.
A Sabina por su apoyo y afecto.
Gracias al Gabi por el aguante y la buena onda.
A Omega.

Índice general

1. Introducción **1**

2. Algoritmo genético de Path Finding no determinístico **9**
- 2.1. Algoritmos Genéticos (GA) 10
  - 2.1.1. Codificación y función de Fitness 11
  - 2.1.2. Selección 12
  - 2.1.3. Reproducción 13
  - 2.1.4. Terminación y Pseudo código del algoritmo 14
- 2.2. Replica de un GA existente 15
  - 2.2.1. Codificación 16
  - 2.2.2. Fitness 18
  - 2.2.3. Evolución 20
  - 2.2.4. Implementación 21
  - 2.2.5. Comparación de resultados 22
- 2.3. Nuestro GA 23
  - 2.3.1. Codificación 25
  - 2.3.2. Fitness 26
  - 2.3.3. Evolución 28
  - 2.3.4. Implementación 29
  - 2.3.5. Comparación de nuestro GA con la replica 30

3. Algoritmos de Path Finding determinísticos **33**
- 3.1. Algoritmo Dinámico 33
  - 3.1.1. Algoritmo propuesto 33
  - 3.1.2. Resultados 36
- 3.2. Algoritmo basado en Dijkstra 37
  - 3.2.1. Algoritmo Propuesto 37
  - 3.2.2. Resultados 40
- 3.3. Algoritmo basado en A^* 41
  - 3.3.1. Algoritmo Propuesto 41
  - 3.3.2. Resultados 43

3.4. Comparación de los algoritmos 44

4. Path Executing **47**
4.1. Robot utilizado 47
4.2. Problemas 49
4.3. Soluciones 51
4.3.1. Basic Forward 52
4.3.2. Wall Stop 53
4.3.3. Wall Following 54
4.3.4. Turn 56
4.4. Resultados 57

5. Path Planning **60**
5.1. Path Planning propuesto 61
5.2. Simulación de Path Planning 63
5.3. Ejecución de Path Planning 69
5.4. Resultados 70

6. Conclusión **72**

Capítulo 1

Introducción

En la acutalidad la automatización de tareas mecánicas a crecido enormemente, impactando facetas de la vida diaria. Por ejemplo, centrales telefónicas permiten que una comunicación por teléfono se realice de forma casi instantánea entre personas en lugares distantes del mundo. Las centrales telefónicas son un claro avance a conexiones conctretadas por medio de clavijas.

Muchas de estas labores mecánicas involucran el desplazamiento de algún tipo de plataforma, para esto se desarrollan plataformas móviles que no necesitan la presencia de personas para desarrollar sus tareas como por ejemplo aspiradoras, cortadoras de césped, o generalizando, robots. Una gran cantidad de estos robots son utilizados en ambientes cerrados, dándole un papel clave a la navegación automática de interiores.

La navegación de interiores consiste en desplazar una plataforma desde un lugar a otro dentro de un ambiente cerrado. La plataforma debe evitar colisionar con obstáculos presentes en el ambiente. El movimiento del robot debe cumplir con condiciones relacionadas al problema que pretende resolver. Estas condiciones afectan tanto las decisiones que el robot debe tomar antes de recorrer el camino, como las decisiones que toma durante el recorrido del mismo.

En la navegación de interiores se encuentran muchos problemas. El problema fundamental es encontrar un camino entre los lugares de origen y llegada, recorrer ese camino, y evitar colisionar con obstáculos presentes en el ambiente donde se desplaza. También, puede ocurrir que el robot se "pierda" dentro de la habitación, suponiendo que se encuentra en un determinado lugar y encontrándose realmente otro. Un robot puede perderse porque los metodos de ubicación utilizados normalmente son complicados y dependen de los dispositivos con los que cuenta la plataforma, por ejemplo, motores, sensores de ultrasonido, odómetros, etc.

El sistema desarrollado en este trabajo propone una solución a los problemas presentes en la navegación de interiores. El sistema se divide en tres partes, Path Finding, Path Executing y Path Planning. Path Planning y Path Finding resuelven el problema de encontrar un camino y recorrerlo evitando colisiones, mientras que Path Executing se encarga de evitar que el robot se pierda durante la navegación.

Definiciones

Path Finding comprende una clase de algoritmos de enrutamiento que buscan caminos de un punto a otro en el ambiente. Los caminos, además, deben tratar de ser óptimos con respecto a condiciones que son determinadas por cada problema de enrutamiento.

Path Executing es un conjunto sencillo de operaciones que trabajan directamente con los instrumentos de movimiento y ubicación del robot. El trabajo de Path Executing es evitar que el robot se pierda, y asegurar que la plataforma siempre se encuentra en el lugar del ambiente donde el robot (a través de una representación interna del ambiente), cree que se encuentra.

Path Planning es un algoritmo que se encarga de encontrar un camino libre de colisiones en un determinado ambiente utilizando Path Finding, recorre ese camino usando las operaciones de Path Executing y evita colisionar con obstáculos desconocidos del ambiente.

Existen dos clases de Path Planning: Global Path Planning y Local Path Planning. Global Path Planning es un algoritmo que resuelve el problema de encontrar un camino libre de colisiones en un terreno estático, donde todos los obstáculos del ambiente son conocidos. En Local Path Planning, el planeamiento es realizado mientras el robot se encuentra en movimiento, en otras palabras, el algoritmo es capaz de producir un nuevo camino como respuesta de cambios a el ambiente. El Path Planning utilizado en este trabajo es local Path Planning.

Problemas

El problema que Path Finding debe resolver, es encontrar un camino libre de colisiones, que a su vez minimice la distancia y la cantidad de giros que debe efectuar la plataforma para llegar al destino.

A pesar de que el principal objetivo de Path Finding es encontrar un camino desde un lugar a otro en un determinado ambiente, no nos satisface cualquier camino que cumpla con esta condición. Si un camino tiene giros, provoca una demora innecesaria en la ejecución del camino, dado que para hacer un giro la plataforma debe detenerse, girar, y volver a arrancar. Si

hay muchos caminos que resuelven el problema de encontrar una ruta de un punto a otro, tenemos preferencia por aquellos de menor longitud dado que los caminos de longitud más grande también ocasionan demoras innecesarias en la ejecución de un camino.

Para realizar el sistema de navegación consideramos necesario tener una representación del ambiente, la cual tiene que mantener concordancia con el ambiente real donde la plataforma se desenvuelve. También debe conservarse la equivalencia entre la posición y orientación real del robot en la habitación con la posición y orientación del robot en la representación del ambiente. Como ya dijimos, Path Executing esta encargado de preservar esta equivalencia.

Los problemas que debe resolver Path Executing nacen del uso de la plataforma móvil. En esta parte del sistema utilizamos un robot en particular que tiene muchos elementos mecánicos y electrónicos, como motores y sensores de diverso tipo.

La plataforma cuenta con dos ruedas impulsadas independientemente por dos motores. Los motores, a pesar de ser similares, tienen diferentes potencias y la velocidad en cada motor puede ser establecida al margen de la velocidad que tiene el otro motor. La plataforma también cuenta con dos odómetros, estos artefactos se encuentran ubicados al lado de las ruedas. Un odómetro es un dispositivo que indica la distancia recorrida en viaje por un vehículo, en nuestro caso cada odómetro indica la distancia recorrida por la rueda en la que esta ubicado. El robot cuenta con un sensor de ultrasonido. Un sensor de ultrasonido, comúnmente denominado sónar, es un sistema de localización similar a los radares, utiliza ondas de ultrasonido para determinar la distancia a un objeto. El sónar esta montado sobre un servo. Un servo (también llamado servomotor), es un dispositivo capaz de ubicarse en cualquier posición en su rango de operación y mantenerse estable en ese lugar. Combinando el servo con el sónar, podemos detectar objetos en un rango de al rededor de 180 grados. Salvando las diferencias, el rango de detección de objetos con el que contamos es similar al rango de la visión humana.

Estos elementos tienen errores. Por ejemplo, un odómetro nos puede indicar que una rueda se movió veinte centímetros. Si el odómetro tiene como unidad mínima de medida cuatro centímetros, luego de un movimiento podemos creer que la rueda se desplazo veinte centímetros cuando en realidad se desplazo veintitrés centímetros. Lo mismo puede ocurrir cuando el robot gira. La plataforma puede realizar muchos movimientos y giros creyendo terminar en un lugar de la representación del ambiente totalmente diferente al sitio donde se encuentra en realidad.

A pesar de que los errores más importantes son los de movimiento y giro, otros dispositivos como sensores de distancia pueden no ser exactos. En

nuestro caso para que la medición de la distancia entre la plataforma y un objeto realizada por el sensor de ultrasonido sea correcta, es necesario que el objeto se encuentre a una ubicado a lo sumo a 2.5 metros y que su superficie sea grande.

Path Planning, en cambio, tiene que encontrar soluciones a otro tipo de problemas, debe lidiar con elementos que se encuentran en el ambiente pero que están ausentes en la representación de la habitación. Path Planning debe obtener un camino y lograr su correcta ejecución evitando colisionar tanto con elementos que se encuentran en la representación del ambiente, como con los elementos que no están presentes en dicha representación.

Soluciones

El trabajo muestra diversos tipos de algoritmos que resuelven el problema de Path Finding. Estos algoritmos pueden agruparse en determinísticos y no determinísticos.

El grupo de los algoritmos de Path Finding no determinísticos esta conformado por un par de algoritmos genéticos (GA). Un algoritmo genético es una técnica de busqueda utilizada para encontrar soluciones o aproximaciones de soluciones a problemas de busqueda y optimización. Un GA puede devolver diferentes soluciones a un mismo problema, usa técnicas inspiradas en la biología evolucionaria como herencia, mutación, selección y reproducción (crossover).

Para desarrollar los algoritmos genéticos nos basamos en un paper de la universidad de Tulsa [5, kamran H. Sedighi y otros]. En el paper se muestra una descripción de un algoritmo genético que realiza Path Finding. Nosotros propusimos otro algoritmo genético para Path Planning y realizamos una replica del algoritmo propuesto por el paper con el fin de comparar los resultados obtenidos por ambos.

Cabe aclarar que en el trabajo hacemos énfasis en la presentación de los GA, explicando minuciosamente que son y mostrando su funcionamiento.

Un algoritmo basado en programación dinámica, otro basado en el algoritmo de Dijkstra y uno denominado A^* forman el grupo de los algoritmos de Path Finding determinísticos. Un algoritmo deteterminístico es un algoritmo al que dado un determinado input, siempre produce el mismo output, los pasos que ejecuta el algoritmo son siempre los mismos para inputs iguales.

La construcción de los algoritmos fue la siguiente: propusimos una función recursiva que resuelve el problema de encontrar un camino y minimizar longitud y giros del mismo en un determinado ambiente. A partir de esta solución generamos un algoritmo dinámico, el cual mantiene tablas en las que guarda el camino mínimo desde un punto de origen determinado a cualquier

punto del ambiente. Desde esta solución obtuvimos la idea de como realizar algunas modificaciones al algoritmo de Dijkstra para que sea capaz de resolver nuestro problema. A^* es una modificación Heurística de Dijkstra y para llevar a cabo el algoritmo A^*, hicimos una modificación sensilla al algoritmo basado en Dijkstra.

Para cumplir con sus objetivos Path Executing utiliza el sensor de ultrasonido, el servo, las velocidades independientes de las ruedas y los odómetros.

Path Executing propone una solución a los errores que surgen en el desplazamiento del robot, brindando un conjunto de operaciones simples. Estas operaciones utilizadas correctamente y en conjunto sirven para recorrer cualquier camino factible en el ambiente. Las operaciones suponen movimientos simples, pero de la forma que al realiza aprovechan elementos presentes del ambiente para corregir errores que son producto del movimiento de la plataforma. Las operaciones brindadas por Path Executing son cuatro: Basic Forward, Wall Following, Wall Stop y Turn.

Basic Forward es una operación que ajusta la velocidad de cada rueda para intentar lograr un movimiento rectilíneo. Es importante realizar esta corrección porque diferencias de potencias entre los motores, o diferencia en las superficies de desplazamiento de cada rueda, agregan errores al movimiento. A pesar de esta corrección, es inevitable que se cometan errores al desplazarse. Para realizar los ajustes de velocidades el robot se vale de odómetros ubicados en las ruedas.

Wall Following aprovecha las paredes que se encuentran a una distancia relativamente cercana del robot y paralelas al movimiento de la plataforma. Esta operación permite que el robot se mantenga a cierta distancia de la pared durante su movimiento. Wall Following corrige, principalmente, los errores que surgen al realizar giros durante el movimiento. Para seguir la pared el robot se vale de un sensor de ultrasonido.

Wall Stop es la operación que utiliza la información de las paredes perpendiculares al movimiento del robot para detenerse a cierta distancia de las mismas. Esta operación es muy útil para corregir errores de desplazamiento. El robot presupone que se encuentra a una distancia determinada de la pared, y se acerca de a poco utilizando Basic Forward. Una vez que se encuentra cerca de la distancia deseada de la pared se mueve lentamente hacia adelante controlando constantemente la distancia entre la plataforma y la pared. Si el robot se acerca demasiado a la pared, retrocede hasta la distancia deseada. Wall Stop utiliza el sensor de ultrasonido para detenerse cerca de la pared.

La operación **Turn** realiza un giro de aproximadamente noventa grados. El mismo puede ser hacia la derecha o izquierda, según se le indique.

Para llevar a cabo la ejecución de un camino Paht Planning se vale de Path Finding y de Path Executing. Path Planning cuenta con una representa-

ción del ambiente y Path Finding utiliza esa representación de la habitación para encontrar el camino buscado. Una vez que Path Planning tiene un camino empieza a recorrerlo. Como conoce el ambiente por la representación del mismo, utiliza las operaciones provistas por Path Executing intentando aprovechar al máximo las ventajas del ambiente. Dado que pueden existir elementos que se encuentren en el ambiente, pero no así en la representación, Path Planning al momento de toparse con estos obstáculos detiene la ejecución del camino, los marca en la representación, busca un nuevo camino con Path Finding y recorre ese camino. Este proceso se repite hasta que llega al lugar deseado, o se detiene si no encuentra una solución.

Conclusiones

Los algoritmos de Path Finding propuestos nos dieron buenos resultados. Todos los algoritmos encuentran un camino que une el punto de origen y fin, bajo las condiciones especificadas por Path Finding.

Nuestra replica del algoritmo propuesto por el paper [5], supera los resultados expuestos en ese paper, esto confirma que nuestra replica fue buena. Dado que nuestra replica supera a los resultados del paper, nos brinda un buen parametro de comparación para nuestra propuesta. El algoritmo genético que generamos supera a la replica del algoritmo propuesto por el paper. Efectuamos muchas pruebas de ambos algoritmos y nuestro algoritmo genético encuentra más soluciones libres de colisiones que la replica del algoritmo propuesto por el paper, los comparamos en un conjunto de 20 mapas y nuestro algoritmo supera en todos los mapas a la replica del algoritmo propuesto, a su vez, la replica del algoritmo iguala o supera los resultados del paper.

Las pruebas realizadas con los algoritmos determinísticos en nuestro conjunto de mapas de prueba nos mostraron que la modificación de A^* es más rápida que el algoritmo dinámico y la modificación de Dijkstra.

Para implementar los algoritmos genéticos utilizamos una librería desarrollada en el lenguaje de programación Java. Los algoritmos se ejecutan exitosamente pero las ejecuciones toman mucho tiempo en terminar. Con el fin de mejorar estos tiempos implementamos completamente nuestro GA, obtuvimos un algoritmo que funciona bien y hasta diez veces más rápido que el algoritmo que utiliza la librería. A pesar de esto, la diferencia en los tiempos de ejecución entre A^* y el algoritmo genético es grande. Lo que buscamos es una aplicación útil en tiempo real, por eso decidimos usar A^* como el algoritmo de Path Finding.

El resultado obtenido por Path Executing fue bueno. Las primitivas propuestas solucionan muchos de los problemas de desfasaje que ocurren durante el desplazamiento de la plataforma. Las operaciones Wall Following y Wall

Stop utilizan las paredes para corregir errores de desplazamiento y giro. Las primitivas de Basic Forward y de giro intentan evitar agregar errores. Efectuamos muchas pruebas de cada una de las primitivas. Wall Following sigue correctamente las paredes y es notable la corrección de giros que realiza. Si el robot esta mal ubicado y se encuentra muy girado cuando tendría que estar paralelo a la pared, Wall Following realiza una corrección de movimiento muy suave, si la distancia que se realiza Wall Following es pequeña, puede no llegar a corregir los errores. Se espera que esta situación no ocurra, pero puede pasar y es uno de los defectos de Wall Following. De las pruebas realizadas con Wall Stop, podemos determinar que el resultado es bueno, dado que siempre lo deja cerca de la distancia deseada a la pared. A pesar de que Basic Forward agrega errores de giro y desplazamiento, a partir de los experimentos realizados podemos decir que es notable la forma en que disminuyen los errores. La primitivas Turn fue ajustada a través de pruebas. Path Executing es útil, sus mejores operaciones son Wall Following y Wall Stop, como ambas dependen de las paredes presentes en el ambiente, una habitación con pocas paredes complica la corrección de errores a través de Path Executing.

Realizamos Path Planning simulando el ambiente y simulando los movimientos del robot, se agregaron obstáculos no conocidos en la representación, para cada mapa que probamos Path Planning llevo a cabo la ejecución de los caminos sin colisionar tanto con los obstáculos conocidos como con los obstáculos desconocidos del ambiente. Luego cuando llevamos Path Planning al ambiente real, con obstáculos reales, los resultados de Path Planning fueron satisfactorios, obtuvimos ejecuciones buenas y algunas ejecuciones malas, Path Planning utiliza correctamente la información de la representación interna del ambiente para tomar decisiones sobre los pasos que la plataforma debe seguir, hace un uso correcto de las primitivas de Path Executing aprovechando las ventajas que estas brindan. Los malos resultados recaen en la poca cantidad de paredes del ambiente que no permiten corregir errores, y en que la plataforma no cuenta con otro sistema de ubicación mejor que el sónar.

Si bien la estrategia propuesta esta basada en este robot en particular, la misma puede ser generalizada para otros tipos de robots.

Estructura del trabajo

A continuación una breve descripción de los temas incluidos en los capítulos que restan del trabajo.

Capitulo 2

En este capitulo definimos los algoritmos de paht finding y los algoritmos de Path Finding no determinísticos. Definimos algoritmos genéticos.

Describimos detalladamente nuestra replica a un algoritmo propuesto en [5, kamran H. Sedighi y otros], y comparamos los resultados de la replica contra los resultados expuestos por el paper.

Mostramos un segundo algoritmo genético el cual es nuestra propuesta de algoritmos genéticos para Path Finding, además comparamos los resultados de nuestro algoritmo contra los resultados obtenidos por la replica.

Capitulo 3

Presentamos la definición de algoritmos determinísticos para Path Finding. Este capitulo contiene la definición, descripción y análisis de nuestro algoritmo dinámico de Path Finding. También mostramos la descripción de el algoritmo Dijkstra, nuestra modificación al algoritmo de Dijkstra y análisis. Definición, descripción y análisis del algoritmo de Path Finding A^*.

Al final del capitulo realizamos una comparación entre los tres algoritmos.

Capitulo 4

Path Executing es desarrollado en el Capítulo 4. En este capítulo se cubren la definición de Path Executing. Además se muestran los problemas que surgen del desplazamiento de una plataforma móvil en un ambiente cerrado.

Se describen detalladamente las operaciones de Path Executing: Wall Following, Wall Stop, Basic Forward y Turn.

Para cerrar el capitulo exponemos los resultados de Path Executing.

Capitulo 5

Cubre las dificultades del traslado del robot en un ambiente que contiene obstáculos desconocidos para la plataforma y las soluciones a estos problemas. Incluso el desarrollo y los resultados de Path Planning son presentados en el capitulo 5.

Capitulo 6

Concluye el trabajo y discute los resultados obtenidos.

Capítulo 2

Algoritmo genético de Path Finding no determinístico

Path Finding se encarga de la labor de encontrar un camino desde un punto de origen a un punto de destino, el cual debe evitar colisionar con los obstáculos del ambiente. El camino buscado debe cumplir con ciertas restricciones, por lo general se busca el camino más corto entre dos puntos. En nuestro caso buscamos un camino que minimice la cantidad de giros y la distancia recorrida entre el punto de partida y el punto de llegada. Consideramos importante que un camino minimice giros, puesto que para hacer un giro la plataforma debe detenerse, girar y volver a arrancar provocando una demora innecesaria en la ejecución del camino.

El input del Path Finding es un mapa, un punto de origen y un punto de destino. El output es un camino.

Mapa o representación del ambiente: es una grilla que divide en filas y columnas el ambiente donde se encuentra la plataforma. En el mapa se representan los obstáculos a través de celdas ocupadas y los sitios libres de obstáculos por medio de celdas libres. Una celda se considera ocupada si contiene total o parcialmente a un obstáculo, en caso contrario la celda se considera libre. Path Finding supone que todos los obstaculos del ambiente se encuentran representados en el mapa. En la Figura 2.1 mostramos un mapa de ejemplo.

Punto de origen: está representado por los números de fila y columna que indican la celda del mapa en la que se encuentra ubicado el robot inicialmente.

Punto de destino: al igual que el punto de origen, son los números de fila y columna de la celda del mapa donde se desea que el robot termine.

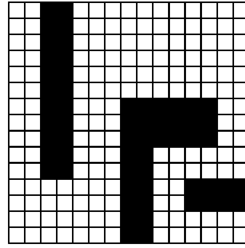


Figura 2.1: Ejemplo de un mapa

Camino: es un conjunto ordenado de celdas vecinas. Decimos que una celda tiene por vecina a otra en una grilla, si comparten la misma fila (o columna) y además la segunda se encuentra en alguna columna (o fila) vecina. Si existen caminos que eviten colisiones, Path Finding intenta devolver el camino con menos giros y más corto entre el punto de origen y el punto de destino. Se considera que un camino no tiene colisiones cuando no contiene celdas ocupadas.

Una de las propuestas de Path Finding esta basada en un algoritmo no determinístico llamado Algoritmo Genético o GA. Un algoritmo no determinístico es un algoritmo que ofrece más de un resultado en función de su entrada. Los algoritmos no determinísticos no permiten saber antes de la ejecución cual sera el resultado a una determinada entrada.

2.1. Algoritmos Genéticos (GA)

En el mundo real, el proceso de selección natural controla la evolución. Organismos mas adaptados a sus ambientes tienden a vivir lo suficiente como para reproducirse, mientras que los organismos menos adaptados frecuentemente mueren antes de reproducirse o producen pocos y/o débiles descendientes.

Un algoritmo genético es una técnica de busqueda que se usa para encontrar soluciones o aproximaciones de soluciones a problemas de busqueda y optimización. Usa técnicas inspiradas en la biología evolucionaria como herencia, mutación, selección y reproducción (crossover).

Una población es un conjunto de individuos, donde cada uno de estos está formado por un conjunto de genes. Los individuos o cromosomas son representados generalmente por cadenas de bits, pero otros tipos de representaciones suelen ser utilizadas. La evolución empieza con una población que generalmente es elegida al azar. En cada generación la aptitud (fitness)

de cada individuo de la población es evaluada. Muchos de los cromosomas de la población actual son elegidos a través de diferentes procesos que combinan azar y el fitness de cada individuo, dando así lugar a la población de la próxima generación por medio de operaciones como crossover o mutación al azar. Luego la nueva población es usada en la siguiente iteración del algoritmo.

Usualmente el algoritmo termina cuando se alcanza un número determinado de generaciones, o cuando un nivel satisfactorio de fitness ha sido logrado por la población. Si el GA termina por haber alcanzado un número determinado de generaciones, puede o no haber encontrado una solución satisfactoria.

Para realizar esta sección del trabajo se utilizaron los libros [1, 2, 4]

2.1.1. Codificación y función de Fitness

Un algoritmo genético requiere que se definan la representación genética del dominio de soluciones y la función de fitness.

Los individuos de una población son llamados también cromosomas. Los cromosomas están formados por un conjunto de parametros (genes) que definen las soluciones propuestas para el problema que el algoritmo genético esta tratando de resolver. Los cromosomas son comúnmente representados a través de una cadena de bits, aunque una amplia variedad de estructuras de datos son también usadas. Algo que hace importante a la representación genética es que los pares de individuos pueden ser fácilmente alineados gracias a que los cromosomas tienen un tamaño fijo, esto facilita las operaciones genéticas que se hacen sobre los cromosomas. También son permitidas las representaciones de tamaño variable, pero la implementación de algunas de las operaciones genéticas se vuelven mas complejas.

El diseño de los cromosomas y sus parametros esta dado por la necesidad especifica del problema a resolver, los operadores genéticos usados por el GA deben tener en cuenta el diseño del cromosoma.

La **función de fitness** es definida sobre la representación genética y mide la cualidad de la solución representada. La función del fitness es siempre problema dependiente.

Por ejemplo, en el problema de la mochila se quiere maximizar la cantidad de objetos que se pueden meter en una mochila de capacidad fija. Una representación del problema puede ser una cadena de bits donde un 0 en la posición iésima del cromosoma significa que el elemento iésimo no esta en la mochila, mientras que un 1 significa que el elemento si se encuentra en la mochila. La función de fitness aplicada a un cromosoma determinado, devuelve la suma de los valores de los objetos en la mochila si no exceden la capacidad, mientras que devuelve 0 en caso contrario.

Una vez que se han definido la representación y la función de fitness se procede a generar la primer generación de la población. Normalmente, las soluciones son generadas al azar para formar la primer generación, pero también se puede comenzar con una población inicial especifica. El tamaño de la población depende del problema a resolver, pero generalmente posee cientos o miles de soluciones posibles.

2.1.2. Selección

Durante las sucesivas generaciones, una proporción de la población existente es elegida para alimentar una nueva generación. Las soluciones mas aptas, según la función de fitness, son normalmente más capaces de ser elegidas. Algunos metodos de selección miden el fitness de cada solución y preferencialmente eligen las mejores soluciones, mientras que otros métodos miden solo una muestra al azar de la población.

La mayoría de las funciones son diseñadas para que una pequeña proporción de los cromosomas con bajo fitness (los menos aptos), sean seleccionados. Esto mantiene la diversidad de la población, previniendo convergencia prematura o soluciones pobres. Algunos de los métodos de selección más populares son roulette wheel selection y tournament selection:

Roulette whell selection El fitness asociado a cada cromosoma por la función de fitness es usado para definir una probabilidad de selección a cada cromosoma. Las soluciones candidatas con mayor fitness van a ser menos probables de ser eliminadas, pero existe la chance de que eso ocurra. Con este proceso de selección existe la posibilidad de que algunas de las soluciones mas débiles sobrevivan al proceso de selección; esto es una ventaja, porque a pesar de ser una solución débil puede tener componentes fuertes que podrían ser útiles en el proceso de recombinación. La analogía de este proceso con la ruleta puede ser vista imaginando una rueda de ruleta en la que cada cromosoma representa un hueco en la rueda; el tamaño de los huecos es proporcional a la probabilidad de selección de la solución. Elegir N individuos de la población es equivalente a jugar N juegos de ruleta.

Tournament selection Este método de selección realiza un torneo entre unos pocos individuos de la población elegidos al azar, luego escoge al individuo que tiene el mejor fitness. La cantidad de cromosomas elegidos al azar esta definida por el tamaño del tournament. Los individuos pueden ser seleccionados más de una vez. Si se elije un tamaño de tournament grande, los individuos más débiles tienen menos probabilidad de ser elegidos.

2.1.3. Reproducción

Dada una generación determinada, se obtiene la siguiente generación a través de operadores genéticos: crossover y mutación.

Los individuos de la nueva generación son generados aplicando al menos una de las operaciones genéticas a sus “padres”, los padres de los nuevos individuos son seleccionados a través de alguno de los métodos de selección descriptos anteriormente. Usualmente la nueva solución creada comparte muchas de las características de sus “padres”. Nuevos padres son escogidos para cada hijo, y el proceso continua hasta que una nueva población de soluciones de un tamaño determinado es generado.

La población obtenida por este proceso difiere de la población inicial, usualmente el fitness promedio de la población crece, esto ocurre porque los mejores cromosomas de la población anterior tienden a ser seleccionados para la reproducción.

Mutación La mutación es un operador genético utilizado para mantener la diversidad genética de una generación de una población a la siguiente. Es análoga a la mutación biológica.

El ejemplo clásico del operador de mutación involucra a la probabilidad de que un bit arbitrario en una secuencia genética sea cambiado de su estado original. Un método común para implementar la mutación involucra generar una variable al azar para cada gen en el cromosoma. Esta variable dice si ese gen en particular será modificado o no.

El propósito de la mutación en los GA es permitir al algoritmo evitar mínimos locales previniendo que los cromosomas de la población se hagan similares. Si se encuentran mínimos locales la evolución se hace mas lenta y hasta incluso se detiene.

Crossover El operador genético de crossover esta basado en el crossover biológico, en el que dos cromosomas intercambian alguna porción de su ADN, generando nuevos cromosomas.

Existen muchas técnicas de crossover y dependen de las diferentes estructuras de datos que los cromosomas tienen. A continuación mostramos las técnicas más comunes, variaciones de estas son realizadas para los individuos que utilizan estructuras de datos complejas.

- **One-point crossover:** Un punto de cruce es elegido en los cromosomas “padres” y toda la información a partir de ese punto de la cadena es intercambiada entre los dos “padres”. Se obtienen como resultados dos cromosomas hijos, como se ilustra en la figura 2.2.

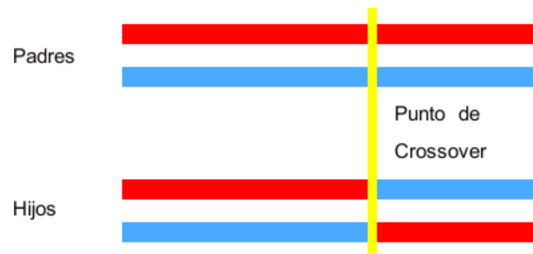


Figura 2.2: One point crossover

- **Two-point crossover:** Se elijen dos puntos de cruce en las cadenas de los cromosomas padres y solo la información que se encuentra entre los puntos de cruce es intercambiada para generar los dos cromosomas hijos. La figura 2.3 muestra la técnica de double-point crossover.

Figura 2.3: Double point crossover

2.1.4. Terminación y Pseudo código del algoritmo

Se obtienen nuevas generaciones hasta que se llega a la condición de terminación del algoritmo. Algunas de las condiciones de terminación mas comunes para un GA son:

- Encuentra una solución que supera un determinado fitness.
- Alcanza un número determinado de generaciones.
- Se termina un plazo de tiempo o se llega a un costo monetario determinado.
- El fitness de la mejor solución es uno tal que las soluciones de las generaciones siguiente no mejoran este resultado.
- Inspección manual.

- Combinaciones de las anteriores.

Pseudo código del algoritmo

1. Elegir población inicial
2. Evaluar el fitness de cada individuo en la población
3. Repetir hasta que se alcance la condición de terminación:
    a) Elegir los mejores individuos para reproducirse
    b) Alimentar la nueva generación a través de crossover y/o mutación
    c) Evaluar el fitness de la nueva población

2.2. Replica de un GA existente

Para desarrollar nuestro algoritmo genético de Path Finding nos basamos en el paper *"Autonomous Local Path Planning for a Mobile Robot Using a Genetic Algorithm"* [5] al cual, de aquí en adelante nos referiremos usando la palabra "paper". En el mismo se resuelve el problema de Path Finding utilizando un algoritmo genético que busca no solo caminos óptimos en longitud, sino también óptimos en giros.

Debido a que nuestro GA se basa fuertemente en el articulo mencionado anteriormente, en esta sección describimos la codificación, el Fitness y la forma de evolución expuestas en el paper. Además explicamos las suposiciones que hicimos al implementar el algoritmo genético propuesto en el paper. A dicha implementación que realizamos la llamamos replica del paper y la comparamos contra la implementación mostrada en el paper en la sección comparación replica.

Para comenzar a describir el GA, consideraremos que siempre se requiere que el robot navegue desde la esquina superior izquierda, a la esquina inferior derecha. Para cumplir con este requisito se definen dos tipos de movimientos que puede realizar el robot: Row-Wised y Column-Wise.

Row-Wised: En esta clase de movimiento la plataforma comienza moviéndose fila por fila desde el punto de partida al punto de llegada. Es decir, se mueve de arriba hacia abajo no pudiendo retroceder hacia arriba. En otras palabras, cada linea horizontal del mapa encuentra el camino solo una vez. (Ver Figura 2.4).

Column-Wise: Aquí el robot comenzara moviéndose hacia el destino columna por columna siempre a la derecha. Es decir, que puede moverse de

izquierda a derecha sin poder retroceder a la izquierda. En otras palabras, cada linea vertical del mapa encuentra el camino solo una vez. (Ver Figura 2.4)

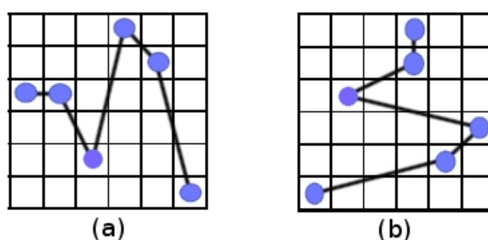


Figura 2.4: (a) movimiento Column-Wised . (b) movimiento Row-Wised.

2.2.1. Codificación

Cada cromosoma puede ser considerado un camino y ser pensado como una solución potencial al problema de Path Finding. Es decir que cualquier cromosoma puede ser traducido a un camino que recorre el mapa, no necesariamente evitando obstaculos. Además un cromosoma puede ser visto como un arreglo, donde cada posición del arreglo es un gen, el cual puede tener un tipo de dato diferente a los demás genes del mismo cromosoma.

La codificación expuesta en el paper consiste de cuatro variables: Path-Flag, Path-Location, Path-Direction, y Path-Switch. Donde cada variable es un conjunto de genes según lo indica la figura 2.5.

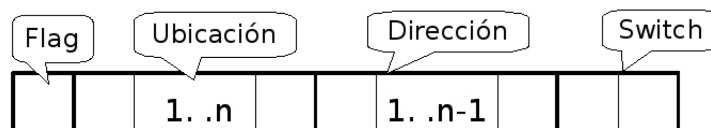


Figura 2.5: Variables de un cromosoma de la replica del paper.

Path-Flag es codificado con un solo gen de tipo binario (0 o 1). La responsabilidad de este bit consiste en indicarle al robot si comienza con un tipo de movimiento row-wise o column-wise (0 o 1 respectivamente).

Path-Location es codificado con n genes de tipo entero donde cada gen puede tener valores de 1 a n, recordemos que n esta determinado por

el tamaño del mapa (nxn). Esta variable define las n ubicaciones del robot en el mapa. Se utiliza tanto la posición de un gen (en el arreglo) como el valor del mismo para obtener un punto de ubicación. De modo que en un movimiento column-wise el valor de x de la coordenada (x, y) es igual a la posición del gen, mientras que el valor de y de esa coordenada se obtiene del valor del gen. En cambio en row-wise el valor de x se toma del valor del gen, y el valor de y se obtiene de la posición. Por ejemplo, si el robot se encuentra en un movimiento row-wise (Path-Flag = 0) la posición de un gen de Path-Location dentro del cromosoma corresponde al número de fila mientras el valor del mismo gen corresponde al número de columna. (véase figura 2.6)

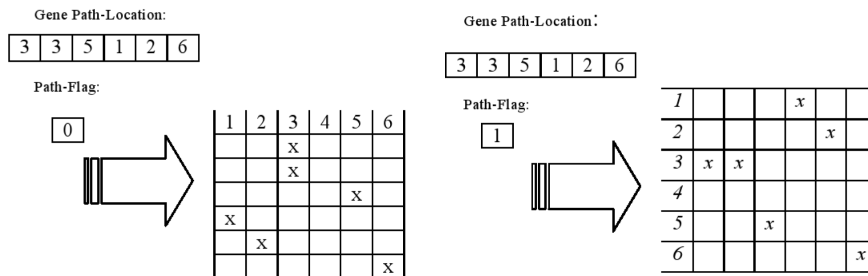


Figura 2.6: Ejemplos de Path-Location, row-wised y column-wised respectivamente.

Path-Direction es codificado con $n - 1$ genes de tipo binario (0 o 1). Esta variable es necesaria porque hasta ahora la estructura del cromosoma descripta solo representa esquinas o pasos intermedios en el camino. Para mover el robot por medio de una linea recta de un Path-Location a otro, hace que la plataforma cruce por muchas celdas adyacentes, necesitando que todas estas estén libres.

Un mejor enfoque es ir primero hacia un costado (horizontalmente), girar, y luego ir hacia abajo (verticalmente), o viceversa (véase Figura 2.7). Para indicar cual de las dos estrategias debe tomar el robot se agregan los genes de Path-Direction, los cuales pueden tomar el valor 0 o 1 según comience horizontal o verticalmente. Los puntos de dirección son uno menos que los puntos de Path-Location ya que no es necesaria una instrucción de dirección para la última ubicación del robot.

Path-Switch se codifica con dos genes de tipo entero con valores de 1 a n. Esta variable permite al robot cambiar (hacer un switch) entre un movimiento row-wised (r.w.) a column-wised (c.w.) o viceversa en el trascurso de la decodificación de un camino. Como hay dos genes de switch esto permite

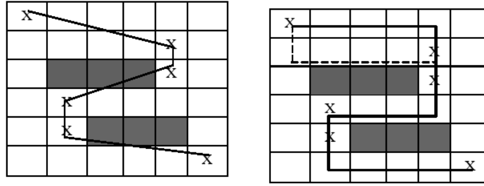


Figura 2.7: Problema del movimiento en diagonal, y solución al movimiento en diagonal.

hacer un máximo de dos cambios o switchs. Los valores de cada gen de tipo Path-Switch indican en que parte del cromosoma se debe hacer el cambio de r.w. a c.w. o al revés, según el tipo de movimiento que se traía.

El hecho de que el cromosoma tenga dos Path-Switch no significa que halla dos puntos de cambio, ya que en el caso que i es el valor del primer punto de switch y j es el segundo, siendo el mapa de dimensión nxn, se supone lo siguiente:

- No hay puntos de switch si $i = j = n$.
- Un punto de switch si $i \neq j = n$.
- Un punto de switch si $i = j \neq n$.
- Dos punto de switch si $i \neq j \neq n$.

2.2.2. Fitness

La población de caminos es evaluada en cada generación. La evaluación se basa en el fitness de cada camino y depende de cuan bien se ajuste la solución (camino) al problema (mapa). En una evaluación preliminar, los valores para el largo de un camino, el número de giros, y el número de pasos no realizables (colisiones) son determinados para cada camino en la población. Estos valores son seteados en relación a la población entera y son representados como valores fraccionales de 0 a 1, donde 1 indica el valor de fitness óptimo. El camino más corto en la población corresponde a $f_{length} = 1{,}0$; el camino más largo en la población corresponde a $f_{length} = 0$. El número más grande de pasos no realizables (colisiones) corresponde a $f_{collisions} = 0$; el menor número de pasos no realizables corresponde a $f_{collisions} = 1{,}0$, y lo mismo para $f_{numberOfTurns}$ en el caso de la cantidad de giros. f_{length} es el valor de fitness asociado a la longitud de un camino, $f_{collisions}$ es el valor de fitness

asociado con el número de pasos no realizables (colisiones), y $f_{numberOfTurns}$ es el valor de fitness asociado con el número de giros en el camino.

Según la importancia de cada parte del fitness de un camino (largo, cantidad de pasos no realizables y cantidad de giros) se le otorga un peso, dependiendo del objetivo del algoritmo. La parte más importante para un camino es la cantidad de pasos no realizables, y debe tener la mayor influencia en el fitness del camino. Entonces, las otras dos partes son multiplicadas por el valor de $f_{collisions}$. Recordemos que los tres valores son números entre 0 y 1. Los dos fitness de longitud y giro son multiplicados por un factor de peso (el valor de estos factores es determinado mediante experimentación) y sumados. Esta suma es dividida por la suma de los pesos de los factores en orden de terminar con un fitness global con valores entre 0 y 1. Finalmente, la función entera de fitness es multiplicada por 100 para terminar con un valor de fitness entre 0 y 100.

$$f_{path} = f_{collisions}.[L.f_{length} + T.f_{numberofTurns}].\frac{100}{L+T}$$

f_{path} es el valor de fitness para el camino completo, L es el factor de peso para f_{length} y T es el factor de peso para $f_{numberOfTurns}$. Tanto para el GA propuesto por el paper y para la replica del paper, L esta seteada a 1 y T a 2, para enfatizar el número de giros sobre la longitud del camino. Entonces penalizando el fitness de giros más que el fitness de longitud, reducimos el número total de giros tomando el riesgo de tener caminos más largo. Esta decisión se basa en la suposición de que un giro toma más tiempo que hacer algunos pasos extras, ya que el robot necesita disminuir la marcha, girar y luego acelerar de vuelta, para completar un giro.

Una penalidad es también aplicada a las soluciones que contienen pasos no realizables, a fin de enfatizar la necesidad de que todos los pasos estén libres de colisiones, para que el camino pueda ser utilizado. Basada en este criterio, la función de fitness es modificada como se muestra a continuación, para todas las soluciones que contienen pasos no realizables:

$$f_{path} = 0{,}1 * \frac{f_{path}}{(número\ de\ colisiones)^2}$$

2.2.3. Evolución

La evolución como se explicó anteriormente solo se valdrá de selección, crossover y mutación para mejorar las poblaciones. A continuación mostramos los procesos de crossover y mutación.

Proceso de Crossover

Tanto en el GA del paper como en la replica del paper, dos cromosomas padres son combinados aplicando un punto de cruce simple (single-cross-point). Por cada operación de crossover se producen dos cromosomas descendientes. Esto se consigue usando la información de los genes,que no fueron usados para construir el primer descendiente, en orden de construir el segundo descendiente. La operación de crossover actúa solo en los puntos de ubicación y dirección (con el path-flag incluido) de los cromosomas, así como no tiene efecto en los dos puntos de switchs. Esta función es ilustrada en la figura 2.8.

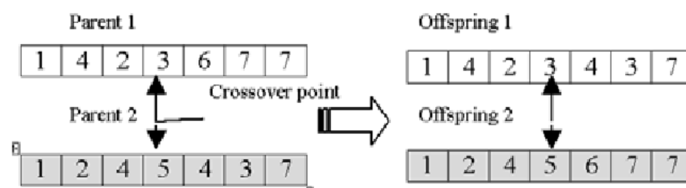


Figura 2.8: Proceso de crossover

Como la operación de crossover especificada recién no tiene efecto en los puntos de switch, produce que se pierda el sentido mismo del crossover que es unir el tramo inicial de un camino con el final de otro y viceversa. Este problema ocurre cuando se realiza la copia de los genes de ubicación de un padre a otro, ya que los Path-Location dependen del movimiento (row-wised o column-wised) que tiene el robot al ser leídos. Como este movimiento es afectado por los puntos de switch y estos no son modificados al realizar el crossover, producen que los path-locations sean leídos de otra forma rompiendo así la intuición de unir el tramo inicial de un camino con el tramo final de otro.Para evitar el problema del crossover al implementar la replica del paper decidimos utilizar los switch points para que crossover haga efectivamente la unión del tramo inicial de un camino con el tramo final de otro.

Ahora tenemos otro problema, la cantidad de switch points necesarios para realizar la unión de dos caminos pueden ser mayor que la cantidad disponible en la representación.

Por ejemplo, supongamos que crossover genera un cromosoma el cual tiene los genes de un primer padre hasta el punto de cruce y los genes de un segundo padre desde el punto de cruce hasta el final. Supongamos además que el cromosoma del primer padre hace uso de los puntos de switch (o switch points) antes del punto de cruce del crossover y el segundo padre hace uso de los switch points después de este punto de cruce. Para mantener el camino generado por esta unión de caminos necesitaríamos al menos cuatro switch points. Además puede ocurrir que no coincida la orientación del camino en el punto de cruce, es decir que el primer tramo termine row-wise y el segundo empiece column-wise o viseversa, por lo tanto es necesario un switch point mas.

En el peor caso, la unión de dos caminos en un determinado punto de cruce necesita cinco switch points, esto excede a la representación del camino que posee solo dos switch points. Si un camino requiere mas de dos switch points no puede ser contemplado por la representación y decimos que hay una falta de expresividad para la representación de dicho camino. Por esta falta de expresividad de la representación no siempre es posible evitar que crossover rompa caminos al unirlos. Por lo tanto modificamos crossover de forma tal que asegure que la primer parte del camino, que necesita a lo sumo dos switch points, sea conservada. De sobrar switch points y de ser necesario se usa uno en el puntos de cruce para que los caminos se unan correctamente. Si aun quedan switch points disponibles se utilizan para el tramo final del camino.

Proceso de Mutación

Toda operación que cambie el orden de los genes dentro de un cromosoma o cambie el valor de un gen (como la ubicación o dirección) es un operador de mutación válido.

El operador de mutación válido usado aquí chequea una probabilidad para cada gen y decide si este debe o no ser mutado. En el caso de que se decida que el gen debe ser mutado un número al azar entre 1 y el total de filas o de columnas es asignado a los puntos de ubicación y de switchs, mientras un 1 o 0 es asignado a los puntos de dirección y al path-flag.

2.2.4. Implementación

En la implementación de la replica del paper utilizamos un librería desarrollada en java [3], a la cual debimos programarle la función de fitness, y las operaciones de crossover y mutación expuestas anteriormente. De forma experimental tuvimos que ajustar varios parámetros no especificados en el

paper. Para esto nos valimos de un conjunto de veinte mapas entre los cuales están los siete mostrados en el paper.

Mediante experimentación determinamos que el algoritmo genético funciona mejor con poblaciones de aproximadamente 10 mil individuos, de esta manera se aumenta la posibilidad de que en la población inicial (la cual es creada al azar) haya individuos con pedazos de caminos que contribuyen a la solución.

La probabilidad de mutación para cada gen la establecimos en $0,06$. Con probabilidades mayores las poblaciones tendían a modificar mucho a los individuos provocando que la evolución de sucesivas generaciones no mejoren el fitness de los individuos. Por otro lado con probabilidades menores las poblaciones tendían a estancarse rápidamente sin conseguir buenas soluciones.

Además decidimos elegir a los cromosomas a ser evolucionados mediante el método de Tournament Selection. Es decir eligiendo 7 individuos al azar, y de estos tomando los dos de mejor fitness para luego realizarles las operaciones de crossover y mutación (en ese orden). Se debe aclarar también que cada nueva generación que se crea en la replica del paper es alimentada solo por descendientes de individuos a los que se les realizó crossover y mutación.

Por ultimo fijamos el número de generaciones en 40. Ya que en todos los mapas en los que experimentamos obteníamos que el GA se estancaba antes de dicha generación, por lo que no era necesario seguir computando nuevas generaciones.

2.2.5. Comparación de resultados

En esta sección comparamos los resultados mostrados en el paper contra los obtenidos por nuestra replica del paper. Para lograr esto, comenzamos agregando nuestros resultados a una tabla expuesta por el paper, en la que comparan sus resultados con trabajos anteriores. Luego mostramos algunos caminos expuestos en el paper junto con los obtenidos por nuestra replica.

En la tabla de la Figura 2.9 se muestran los porcentajes de caminos libres de colisiones que obtuvieron los GA de dos versiones anteriores del paper, la propuesta en la publicación en cuestión y por ultimo nuestra replica. Para realizar esta comparación realizamos 20 ejecuciones de la replica sobre cada mapa, mientras que en el paper se realizaron 15 ejecuciones por mapa. En la tabla se puede ver que no pudimos comparar nuestros resultados en el mapa SPSet03. Este mapa no es presentado en el paper. La tabla compara la cantidad de soluciones, y no hace referencia a la calidad de soluciones, es decir, en la tabla no importa si los caminos son óptimos o se encuentran cerca de serlos, solo importa si los resultados evitan colisiones.

A continuación contrastamos algunos caminos mostrados por el paper con

Search space	Success Rate (%)			
	Geisler's GA	Hermanu's GA	Paper GA	Replica GA
SPSet01	100	93.3	93.3	100
SPSet02	0.00	86.7	100	100
SPSet03	73	86.7	100	
SPSet04	53	80	100	100
SPSet05	0.00	100	100	100
SPSet06	0.00	20	100	100
SPSet07	0.00	0.00	86.7	100
SPSet08	0.00	0.00	73.3	95

Figura 2.9: Comparación de porcentajes de éxitos entre los resultados expuestos en el paper y los resultados obtenidos por nuestra replica

caminos obtenidos con nuestra replica. Mostramos tres mapas y presentamos dos soluciones para cada mapa. Los mapas mostrados son los mapas SPSet06, SPSet07 y SPSet08 del paper.

La primer solución de cada mapa es la presentada por el paper, y la segunda solución de cada mapa es la nuestra

En la Figura 2.10 se pueden apreciar tres caminos obtenidos del paper que recorren los mapas SPSet06, SPSet07 y SPSet08 respectivamente. Como se puede observar, los caminos mostrados resuelven los mapas, es decir, se encuentran libres de colisiones. Además, la cantidad de giros y la longitud de estas soluciones no es la mínima que se puede obtener en cada mapa.

En la Figura 2.11 se presentan tres caminos obtenidos con nuestra replica que resuelven los mismos mapas. A diferencia de los resultados expuestos en la Figura 2.10, las soluciones aquí mostradas minimizan la cantidad de giros y la longitud del camino.

Si bien con nuestra replica no siempre obtenemos caminos sin colisiones, podemos afirmar que según la experimentación que realizamos el índice de caminos exitosos que obtuvimos es mayor a la expuesta por el paper.

2.3. Nuestro GA

Presentamos a continuación nuestro algoritmo genético, el cual intenta ser una mejora al algoritmo presentado en la Sección 2.2. La idea para mejorar el GA anterior consiste en cambiar la codificación de los individuos, de forma que lo mismos tengan:

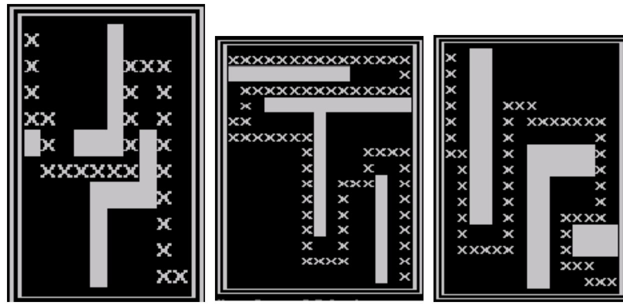


Figura 2.10: Caminos obtenidos por el paper en los mapas SPSet06, SPSet07 y SPSet08 respectivamente.

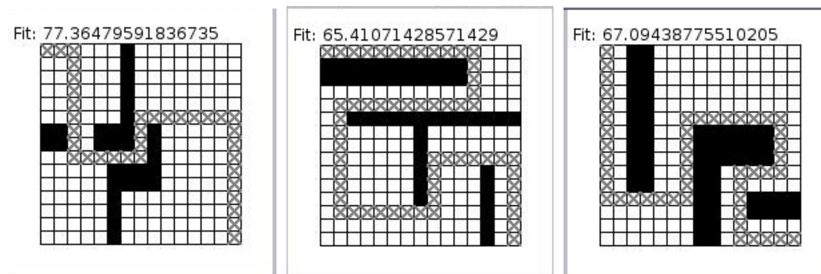


Figura 2.11: Caminos obtenidos por la replica en los mapas SPSet06, SPSet07 y SPSet08 respectivamente.

- Mayor expresividad, es decir se incremente la cantidad de caminos que pueden ser representados mediante un cromosoma.
- Menor cantidad de variables, para minimizar la cantidad de cromosomas diferentes que se pueden crear con la codificación, disminuyendo de esta forma el espacio de soluciones posibles.

Como esta modificación repercute en todo el GA, presentamos los cambios que realizamos a las operaciones de evolución (crossover y mutación), junto a los cambios en la función de fitness y los parametros elegidos.

Por último comparamos nuestro GA con la replica del paper de la Sección 2.2. Recordemos que al momento de comparar los resultados del paper con la replica que realizamos del mismo, obtuvimos que nuestra implementación superaba la expuesta por ellos. Es por esto que la comparación solo la realizamos contra la replica del paper y no la hacemos contra los resultados expuestos por el paper.

En este algoritmo genético también consideraremos que siempre se requiere que el robot navegue desde la esquina superior izquierda, a la esquina inferior derecha del mapa. Para cumplir con este requisito se vale de dos tipos de movimientos que puede realizar el robot: Row-Wise y Column-Wise. Además, al igual que el GA anterior, supondremos que los mapas tienen dimensiones cuadradas (nxn).

2.3.1. Codificación

La codificación en nuestro GA solo consiste de dos variables: Tipos de movimiento y Ubicación. (véase Figura 2.12):

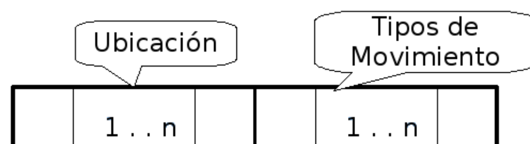


Figura 2.12: Variables de un cromosoma de nuestro GA.

Tipos de movimiento La variable está codificada con n genes de tipo binario (0 o 1). Cada gen que compone esta variable indica que tipo de movimiento (row-wised o column-wised) debe tomar el robot en cada punto de ubicación. Por ejemplo, si un gen ubicado en la posición i de la variable tipo de movimiento contiene un 0, el robot tomará un movimiento row-wised en el lugar indicado en la posición i de la variable de ubicación.

Ubicación La variable de se encuentra codificada por n genes de tipo entero con valores comprendidos entre 1 y n, y define los n puntos de ubicación del robot en el mapa. Para obtener un punto de ubicación se utilizan tanto la posición de un gen en la variable de ubicación, como el valor que se encuentra almacenado en ese gen de la variable de ubicación. En un movimiento column-wise la coordenada x es igual a la posición del gen y la coordenada y se obtiene del valor del gen, en cambio, si el movimiento es row-wise se toma x como el lugar en la variable de ubicación, mientras que el valor existente en ese gen de ubicación es considerado como y. En la Figura 2.13 mostramos un ejemplo de la decodificación de un cromosoma en un camino del mapa, con cruces negras se observan los puntos de ubicación obtenidos por la codificación, y con cruces grises se muestra el camino generado por esta codificación.

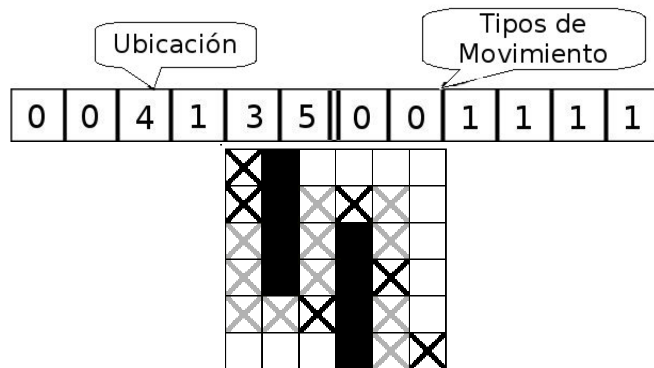


Figura 2.13: Ejemplo de un camino obtenido con nuestra codificación

Como se puede ver en la codificación presentada no se indica que dirección (horizontal y luego vertical o viceversa) debe tomar el robot en cada punto de ubicación. Estas decisiones se toman al momento de decodificar el camino contenido en el cromosoma. Esto se consigue utilizando el mapa del ambiente. La estrategia consiste en evaluar arbitrariamente alguna de las dos direcciones en cada punto de ubicación, si esta elección no provoca que el camino colisione entre dicho punto de ubicación y el siguiente, será la dirección escogida. En cambio, si se detecta una colisión en el tramo del camino en cuestión se opta por la otra dirección siempre y cuando no colisione. Si ambas colisionan se prefiere la dirección que agrega menos colisiones al camino. Al eliminar la variable de la dirección de una ubicación a otra en el camino (anteriormente llamada Path-Direction), podemos agregar los tipos de movimiento a la codificación sin aumentar el espacio de soluciones.

2.3.2. Fitness

La función de fitness mostrada a continuación es similar a la presentada en la Sección 2.2.2. Decidimos usar esta función porque fue pensada para calificar el camino que representa el cromosoma y es exactamente lo que necesitamos para este algoritmo.

En cada generación se necesita evaluar el fitness de cada uno de los individuos de la población. Para evaluar el fitness de cada inividuo realizamos un calculo previo para determinar los valores del largo de un camino, el numero de giros, y el numero de pasos no realizables (colisiones) en cada camino en la población. Estos valores son establecidos con valores fraccionales de 0 a 1 donde 1 indica el valor de fitness óptimo.

El camino más corto en el ambiente corresponde a $f_{length} = 1{,}0$, por ejemplo en un mapa de n filas y m columnas el camino mas corto es aquel que tiene longitud de $n+m$; el camino más largo esta asociado a $f_{length} = 0$, y es el camino que recorre todas las celdas del mapa, por ejemplo, en un mapa de n filas y m columnas el camino de longitud $n*m$ es el más largo. El camino que presenta colisiones en todas las celdas del mapa tiene asociado $f_{collisions} = 0$; el camino que no presenta colisiones tiene asociado $f_{collisions} = 1{,}0$. Lo mismo ocurre para $f_{numberOfTurns}$ que es el numero asociado a los giros del camino. f_{length} es el valor de fitness asociado a la longitud de un camino, $f_{collisions}$ es el valor de fitness asociado con el número de pasos no realizables (colisiones), y $f_{numberOfTurns}$ es el valor de fitness asociado con el número de giros en el camino.

Como queremos obtener caminos libres de colisiones, le damos mayor importancia a la cantidad de pasos no realizables, para que eso ocurra debe tener mayor influencia en el fitness del camino. Recordemos que los tres valores son números entre 0 y 1. Los fitness de longitud y giro son multiplicados por un factor de peso (el valor de estos factores es determinado mediante experimentación) y sumados. Para que las colisiones sean influyentes multiplicamos la suma anterior por el valor de $f_{collisions}$. Este valor lo dividimos por la suma de los pesos de los factores asociados a los giros y longitud con el objetivo de finalizar con un fitness global con valores entre 0 y 1. Finalmente, la función entera de fitness es multiplicada por 100 para terminar con un valor de fitness entre 0 y 100, por lo tanto un camino con fitness de 100, es un camino óptimo.

$$f_{path} = f_{collisions}.[L.f_{length} + T.f_{numberOfTurns}].\frac{100}{L+T}$$

f_{path} es el valor de fitness para el camino completo, L es el factor de peso para f_{length} y T es el factor de peso para $f_{numberOfTurns}$. Con el objetivo de dar mas peso al número de giros que a la longitud del camino, asociamos L a 1 y asociamos T a 2. Entonces penalizando el fitness de giros más que el fitness de longitud, reducimos el número total de giros tomando el riesgo de tener caminos más largos. Esta decisión se basa en la suposición de que un giro toma más tiempo que hacer algunos pasos extras, ya que el robot necesita disminuir la marcha, girar y luego acelerar nuevamente, para completar un giro.

A fin de enfatizar la necesidad de que todos los pasos estén libres de colisiones, otra penalidad es aplicada a las soluciones que contienen pasos no realizables. Basada en este criterio, para todas las soluciones que contienen pasos no realizables, modificamos la función de fitness de la siguiente manera:

$$f_{path} = 0{,}1 * \frac{f_{path}}{(número\ de\ colisiones)^2}$$

2.3.3. Evolución

Con el objetivo de evolucionar la población inicial a una mejor población, el proceso de evolución de un algoritmo genético se vale de las operaciones genéticas *crossover* y *mutación*. Como la codificación de nuestro algoritmo genético es distinta a la codificación de la replica del paper, y como las operaciones genéticas dependen de la codificación de los individuos, las operaciones de crossover y mutación de nuestro algoritmo son distintas a las expuestas en la replica del paper (Sección 2.2.3).

Proceso de Crossover Al igual que el GA anterior en esta operación dos cromosomas padres son combinados aplicando un punto de cruce simple (single-cross-point) mostrado en la Sección 2.1.3 . Por cada operación de crossover se producen dos cromosomas descendientes. Esto se consigue usando la información de los genes que no fueron usados para construir el primer descendiente, en orden de construir el segundo descendiente.

Recordemos que en la replica del algoritmo del paper, la posición y orientación no solo esta definida por un determinado gen, sino que también depende del tipo de movimiento trae de los genes anteriores a este y de la ubicación de los puntos de switch. Esa codificación complica la operación de punto de cruce simple y solo garantiza que la primer parte de un camino puede ser mantenida luego de un crossover, pudiendo perder así la segunda parte del camino. En cambio, la codificación de nuestro algoritmo nos permite unir dos tramos de dos caminos de forma simple.

El procedimiento de crossover para este algoritmo consiste en tomar dos cromosomas, elegir una posición al azar entre los genes de ubicación de un cromosoma y utilizar esa posición para crear a los dos cromosomas hijos. El primer cromosoma hijo tiene el tramo inicial de la variable de ubicación del primero de los padres, hasta la posición anteriormente determinada a través del azar. Además, a partir de esa posición, tiene el tramo final a partir de la variable de ubicación del segundo padre. Para conseguir la variable tipos de movimiento del primer hijo, se toma el tramo inicial de esa variable del primero de los padres hasta el punto elegido al azar y desde ese lugar hasta el final de la variable se toman los genes ubicados en esos lugares del segundo cromosoma padre. El segundo cromosoma hijo es obtenido utilizando los

genes que no fueron tenidos en cuenta para la creación del primer cromosoma hijo. Este proceso de crossover esta graficado en la Figura 2.14.

El motivo por el que se copian los genes de la variable de posición y los genes ubicados en el mismo lugar de la variable de tipo de movimiento, es que cada gen de la variable de posición depende al gen ubicado en el mismo sitio de la variable de tipo de movimiento, esto fue previamente explicado cuando definimos la codificación de este GA.

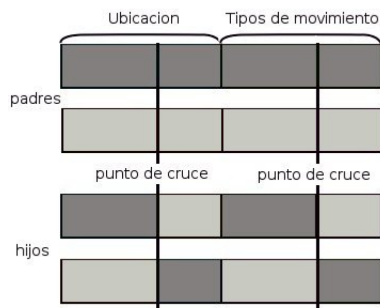


Figura 2.14: El proceso de crossover del GA

Proceso de Mutación El operador de mutación usado aquí chequea una probabilidad para cada gen y decide si este debe o no ser mutado. En el caso de que se decida que el gen debe ser mutado, un número al azar entre 1 y el total de filas o de columnas es asignado a los puntos de ubicación, mientras un 1 o 0 es asignado a los puntos de tipo de movimiento.

2.3.4. Implementación

Para implementar este algoritmo genético nos valimos de la librería [3], utilizada también en la Sección 2.2.4 para el desarrollo de la replica del paper. La ventaja de usar esta librería radica en que solo es necesario programar la función de fitness y el operador de crossover, mientras que las demás partes del algoritmo son especificadas como parámetros a la librería.

La función de fitness y el operador de crossover fueron implementados de forma directa y como fueron explicados en esta sección. Para determinar los parámetros a especificar en la librería, experimentamos utilizando los mapas presentados en la Sección 2.2.4. El proceso de experimentación fue largo y tuvimos que determinar varios parámetros. Ellos fueron:

- El número de individuos por población, establecido finalmente en 10 mil. Esta cantidad nos asegura que en la población inicial tendrá pedazos de caminos que aportan a encontrar la solución.
- Tamaño de Selección, lo fijamos en 7. Con este número mas la elección del método de selección Tournament selection, nos aseguramos de mantener una buena diversidad en la población, previniendo de esta forma una convergencia prematura o soluciones pobres.
- Determinar un orden de ejecución para los operadores de evolución. Por medio de este parámetro le indicamos a la librería primero hacer crossover y luego mutar los descendientes.
- Probabilidad de mutación para cada gen, con un valor de 0,06. Con probabilidades mayores la mutación modifican mucho a los individuos provocando que la evolución de sucesivas generaciones no mejoren el fitness de los individuos. Por otro lado, con probabilidades menores las poblaciones tienden a estancarse rápidamente sin conseguir buenas soluciones.
- Fijar el número máximo de generaciones a evolucionar, finalmente establecido en 35 generaciones. En los 20 mapas obtenemos que el GA se estanca antes de la generación 35, por lo que no tiene sentido seguir con la evolución.

2.3.5. Comparación de nuestro GA con la replica

A continuación contrastamos los resultados de nuestro algoritmo contra el GA implementado en la Sección 2.2.4.

La tabla de la Figura 2.15 muestra la cantidad de éxitos en 20 ejecuciones de cada GA en cada mapa. Se puede apreciar en esta, que la tasa de éxitos de nuestro GA es superior a la del algoritmo replicado anteriormente. Esta superioridad se puede apreciar en los mapas 1, 2, 16 y 19, entre los cuales recalcamos el caso del mapa 16, donde la diferencia radica entre no poder resolver el mapa (0 % de exitos) y si poder resolverlo (60 % de exitos).

A continuación, en el gráfico de barras de la Figura 2.16 mostramos la cantidad de caminos óptimos obtenidos por los dos algoritmos en las 20 ejecuciones anteriores. En este se ve que no siempre nuestra modificación obtiene más caminos óptimos que la replica, sino que estas cantidades se equiparan. Aún así nuestra modificación encuentra mayor cantidad de caminos óptimos en 10 de los 20 mapas, mientras que produce menos caminos óptimos solo en 6 mapas.

Mapa	GA anterior	Nuestro GA
1	5%	100%
2	85%	100%
3	100%	100%
4	100%	100%
5	100%	100%
6	100%	100%
7	100%	100%
8	100%	100%
9	100%	100%
10	100%	100%
11	100%	100%
12	100%	100%
13	100%	100%
14	100%	100%
15	100%	100%
16	0%	60%
17	100%	100%
18	100%	100%
19	10%	100%
20	100%	100%

Figura 2.15: Cantidad de éxitos obtenidos por cada algoritmo al correrlo 20 veces en cada mapa.

Es importante observar que si bien nuestro GA en varios casos no obtiene caminos óptimos (como en el mapa 8), este si obtiene caminos muy cercanos al óptimo (ver Figura 2.17). Esto se debe a que en los cromosomas (caminos) no está especificada la dirección que se debe tomar en cada punto de ubicación, sino que esta es elegida al momento de la decodificación, sin tener en cuenta el hecho de que al elegir una dirección u otra, se puede agregar o eliminar giros. La dirección solo se elige de forma de evitar colisiones, y no de minimizar giros, esto a veces produce que no se consigan caminos óptimos sino caminos muy parecidos a los óptimos. Dado que se espera que el algoritmo funcione en tiempo real, y que los resultados obtenidos son superiores a la replica, decidimos no tomar ninguna estrategia para evitar los problemas de elegir la dirección que debe tomar con el fin de evitar giros, disminuyendo así los tiempos de calculo del algoritmo.

Con estos resultados pudimos ver que nuestras modificaciones en la representación del GA de [5] logran darle más expresividad a los caminos pudiendo así solucionar una variedad de mapas más grande, además, gracias a la disminución de la cantidad de variables, resuelve los mapas precisando menos tiempo de ejecución.

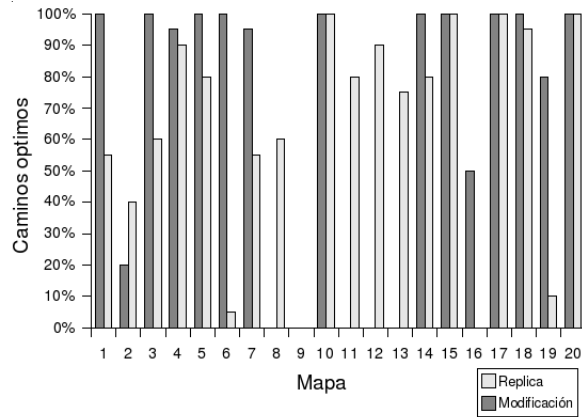
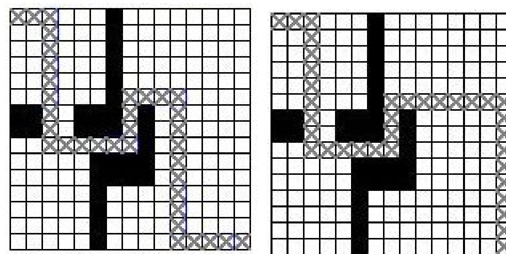


Figura 2.16: Cantidad de caminos óptimos obtenidos por cada algoritmo al correrlo 20 veces en cada mapa.



Mapa 8

Figura 2.17: A la izquierda un camino no óptimo obtenido por nuestro GA en el mapa 8; a la derecha un camino óptimo obtenido por nuestra implementación del GA presentado en [5].

Capítulo 3

Algoritmos de Path Finding determinísticos

Un algoritmo deteterminístico es un algoritmo para el que dado un determinado input, siempre produce el mismo output, los pasos que ejecuta el algoritmo son siempre los mismos para un mismo input. Dicho de otra forma, si se conocen las entradas del algoritmo siempre producirá la misma salida, y la máquina interna pasará por la misma secuencia de estados.

Un modelo simple de algoritmo determinístico es la función matemática, de esta forma se puede establecer el siguiente paralelismo: la función extrae la misma salida para una entrada dada, al igual que los algoritmos determinísticos. La diferencia es que un algoritmo describe explícitamente como la salida se obtiene de la entrada, mientras que las funciones definen implícitamente su salida.

3.1. Algoritmo Dinámico

Una solución para mejorar los tiempos de ejecución de funciones recursivas de alta complejidad son los algoritmos dinámicos. Un algoritmo dinámico resuelve problemas solucionando versiones más chicas del problema, guardando estas soluciones para luego combinarlas y resolver el problema más grande.

3.1.1. Algoritmo propuesto

Nuevamente suponemos que se requiere navegar en un mapa representado por una matriz M de dimensiones nxm, partiendo desde la posición $(1, 1)$ y llegando a la posición (n, m). Para esto, propusimos una función f recursiva

que resuelve el problema de minimizar los giros y la longitud de un camino en M. f toma los siguientes parámetros:

- (i) un número entero que indica la fila donde se encuentra el punto al que se quiere llegar.
- (j) un número entero que indica la columna donde se encuentra el punto al que se quiere llegar.
- (c) un número entero donde se indica la cantidad de giros que debe contener el camino.
- (p) una letra o carácter que indica la orientación en la que debe terminar mirando el robot al llegar al punto de destino.

Por orientación entendemos los puntos cardinales tomados con respecto al mapa, donde Este (E) significa mirar a la derecha, Oeste (O) a la izquierda, Norte (N) arriba y Sur (S) abajo. Esta función recursiva encuentra un camino entre el punto de inicio ($(1,1)$) y el punto de fin ((i,j)), en caso de que existe un camino la función devuelve la longitud del camino, o devuelve infinito (∞) en el caso de que no exista un camino libre de colisiones. A continuación presentamos la definición de la función f.

$$\begin{array}{lcll} f(i,j,c,p) & = & 0 & si \quad i=j=c=0 \\ f(i,j,c,p) & = & \infty & si \quad i \notin [0,n-1] \vee j \notin [0,m-1] \vee c < 0 \end{array}$$

$$\begin{array}{lcll} f(i,j,c,E) & = & min(& f(i,j-1,c,E), \\ & & & f(i-1,j,c-1,S), \\ & & & f(i+1,j,c-1,N) \quad)+1 \end{array}$$

$$\begin{array}{lcll} f(i,j,c,O) & = & min(& f(i,j+1,c,O), \\ & & & f(i-1,j,c-1,S), \\ & & & f(i+1,j,c-1,N) \quad)+1 \end{array}$$

$$\begin{array}{lcll} f(i,j,c,N) & = & min(& f(i+1,j,c,N), \\ & & & f(i,j+1,c-1,O), \\ & & & f(i,j-1,c-1,E) \quad)+1 \end{array}$$

$$\begin{array}{lcll} f(i,j,c,S) & = & min(& f(i-1,j,c,S), \\ & & & f(i,j+1,c-1,O), \\ & & & f(i,j-1,c-1,E) \quad)+1 \end{array}$$

El algoritmo dinámico usa la función f para llenar una tabla de 4 dimensiones. La tabla es usada para evitar calcular más de una vez la misma llamada a f. Además, esta tabla es utilizada para recuperar el camino una vez que el algoritmo ha terminado.

Entonces, para resolver un mapa, el algoritmo dinámico llama varias veces a f hasta obtener un resultado. Inicialmente ejecuta a f con 0 giros ($c = 0$) en las cuatro direcciones (E, O, N y S), si ninguna de las cuatro llamadas obtuvo algún valor distinto de ∞, incrementa c y vuelve a hacer las cuatro llamadas a las diferentes direcciones. Esto se repite mientras f devuelva infinito (∞). Además en el algoritmo se establece como cota el valor $n*m$ para la cantidad de giros. Se elige este valor como cota porque un mapa de dimensiones nxm no puede contener un camino con más de $n*m$ giros.

A continuación mostramos el algoritmo en pseudo código:

```
c := 0
l := INFINITO;

while ( l >= INFINITO and c < n*m ) do
        l :=  min( f(n,m,c,E),
                   f(n,m,c,O),
                   f(n,m,c,N),
                   f(n,m,c,S))

        c := c+1
od
```

Con esto logramos que el algoritmo encuentre un camino, siempre que exista una solución para el mapa. Además, el camino encontrado será la solución con menor cantidad de giros de todas las soluciones posibles, dado a que comenzamos a buscar caminos con 0 giros y los vamos incrementando hasta conseguir alguna solución. También, el camino obtenido será el de menor longitud con dicha cantidad de giros (c), debido a que tomamos el mínimo de las distintas soluciones de f con c cantidad de giros.

El procedimiento recién mostrado, sirve también para establecer la dirección en la que el robot se debe encontrar en la posición de origen y la posición que debe terminar para minimizar así la cantidad de giros. Si la dirección de llegada del robot se encuentra fija, por ejemplo la dirección E, se debe realizar el mismo ciclo pero solo llamando a la f con E. En pseudo código se escribe de esta forma:

```
c := 0
l := INFINITO;

while ( l >= INFINITO and c < n*m ) do
        l :=  f(n,m,c,E)
        c := c+1
od
```

3.1.2. Resultados

Todas las ejecuciones de el algoritmo dinámico sobre nuestro set de 20 mapas obtuvieron resultados positivos. El algoritmo encuentra caminos libres de colisiones en todos los mapas ingresados. Los caminos encontrados además cumplen las exigencias de ser mínimos en giros y en longitud (son óptimos para cada mapa). En la Figura 3.1 presentamos cinco mapas resueltos por el algoritmo. En esta figura también se puede apreciar el tiempo de ejecución que tomó cada mapa en ser resuelto.

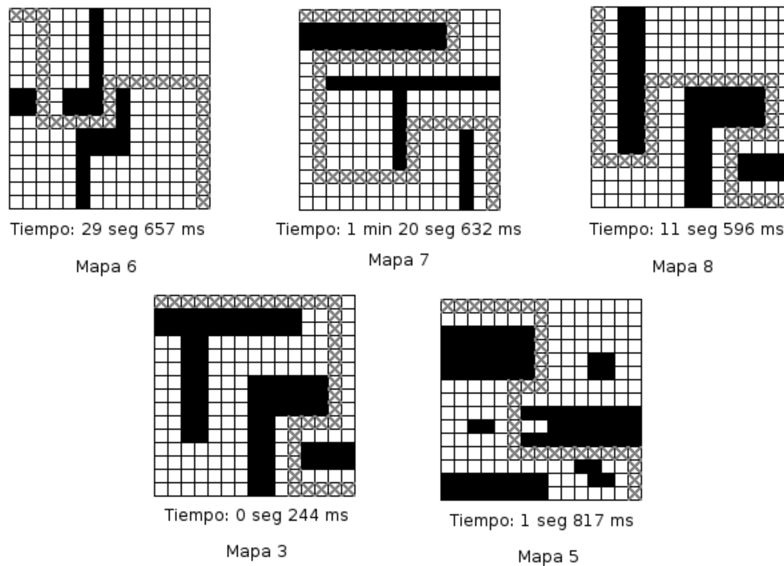


Figura 3.1: Caminos obtenidos por el algoritmo dinámico junto con el tiempo de ejecución de cada uno.

Si se analizan los tiempos de ejecución en cada mapa, se pueden apreciar grandes variaciones de tiempo entre las ejecuciones. Estas variaciones están

dadas por la cantidad de giros que tiene que realizar el camino y por la cantidad de casilleros ocupados en la matriz que representa el mapa.

La cantidad de giros que necesita realizar un camino para resolver un mapa sin colisionar es determinante en el tiempo de ejecución. Mientras mayor sea la cantidad de giros que hay que realizar mayor es la cantidad de llamadas a la función recursiva f que utiliza el algoritmo dinámico. Esto se da a causa de que el algoritmo inicialmente llama a f para buscar caminos que tengan cero cantidad de giros, en el caso de no encontrar ninguno continua llamando a f con dos giros y así sucesivamente.

La cantidad de casilleros ocupados en la matriz que tiene que resolver el algoritmo dinámico impacta directamente en el tiempo de ejecución. Dado que una vez que la función recursiva f es llamada desde un cierto casillero, esta se vuelve a llamar así misma desde los casilleros "vecinos"(respecto a movimientos) libres únicamente. Así si en vez de haber ningún casillero "vecino" ocupado en una llamada de f hay dos, el algoritmo evita hacer dos llamadas de f, disminuyendo esto el tiempo de ejecución.

Con este algoritmo nos aseguramos de encontrar un camino libre de colisiones en un mapa siempre que este exista. Además el camino encontrado será óptimo en giros y en longitud.

3.2. Algoritmo basado en Dijkstra

El algoritmo de Dijkstra, también llamado algoritmo de caminos mínimos, es un algoritmo para la determinación del camino más corto de un vértice origen al resto de vértices en un grafo dirigido y con pesos en cada arista. Su nombre se refiere a Edsger Dijkstra, quien lo describió por primera vez en 1959.

La idea subyacente en este algoritmo consiste en ir explorando todos los caminos más cortos que parten del vértice origen y que llevan a todos los demás vértices; cuando se obtiene el camino más corto desde el vértice origen, al resto de vértices que componen el grafo, el algoritmo se detiene. El algoritmo es una especialización de la búsqueda de costo uniforme, y como tal, no funciona en grafos con aristas de costo negativo.

3.2.1. Algoritmo Propuesto

Como dijimos, el algoritmo de Dijkstra determina el camino más corto en un grafo, como nosotros necesitamos encontrar el camino más corto y con la mínima cantidad de giros en un mapa (una matriz), transformamos el mapa en un grafo y ejecutamos el algoritmo de Dijkstra tal como se lo conoce (sin

modificaciones) sobre el grafo obtenido. El resultado de esta ejecución es un camino en el grafo, luego, transformamos el camino obtenido en el grafo en un camino del mapa.

La principal complejidad en este algoritmo es la transformación de una Matriz a un grafo, dado que esperamos obtener el camino mínimo con el algoritmo de Dijkstra utilizando el grafo, y que ese camino pueda ser decodificado a un camino en la matriz que minimiza giros y longitud.

Para resolver esto nos basamos en la idea del algoritmo Dinámico de la sección anterior, que consiste en mantener la orientación y la cantidad de giros acumulada en cada celda de la matriz utilizada como cache. Con esta idea lo que hacemos es construir un grafo donde a cada casilla del mapa le corresponden cuatro nodos, uno para cada orientación que puede tener el robot (N, S, E y O). Luego unimos los nodos "vecinos" mediante aristas. Un nodo tiene por vecino a otro si puede llegar a este a través de algún movimiento. Por ejemplo, si el robot se encuentra en la celda (i_0, j_0) mirando al Este en el grafo se ubica en el nodo $[i_0, j_0, E]$, los movimientos que puede realizar desde esta ubicación son:

- Desplazarse desde la celda (i_0, j_0) y quedar mirando al Este en $(i_0, j_0 + 1)$, esto sería moverse del nodo $[i_0, j_0, E]$ al $[i_0, j_0 + 1, E]$, lo que hace que estos nodos sean "vecinos" entre si.

- Otro movimiento posible es girar en la celda (i_0, j_0) y quedar mirando al Norte en la misma celda, esto sería moverse del nodo $[i_0, j_0, E]$ al $[i_0, j_0, N]$.

- O por ultimo, girar a la derecha en la celda (i_0, j_0) y quedar orientado hacia el Sur, lo que implicaría moverse del nodo $[i_0, j_0, E]$ al $[i_0, j_0, S]$.

Esto crea un grafo que puede ser pensado como con cuatro capas, donde cada capa esta compuesta de nodos del mismo tipo de orientación. Estos están ubicados de la misma forma que los casilleros de una matriz (ver figura 3.2 (a)). Cada nodo además de tener como vecino a uno de la misma capa también tiene como vecinos a dos nodos de diferentes capas, según lo muestra la imagen 3.2 (b).

Un matriz puede contener celdas ocupadas, por las cuales el robot no puede transitar. Esto es reflejado también en el grafo mediante la eliminación de las aristas que llegan a los nodos que representan celdas ocupadas. De esta forma nos aseguramos que Dijkstra no encuentre caminos no realizables, dado que estos nodos al no estar conectados con el resto del grafo no podrán formar parte de ningún camino.

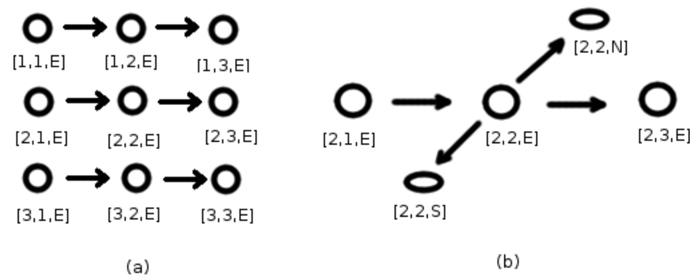


Figura 3.2: (a) Una de las cuatro capas del grafo utilizado por Dijkstra para resolver el Path Finding. (b) Posibles conexiones de un nodo.

Una vez generado el grafo, corremos el algoritmo de Dijkstra para caminos mínimos tal como se lo conoce (sin modificaciones). Por ejemplo, si queremos llegar desde la celda $(1,1)$ mirando hacia el Este y terminar en la celda (n,n) con el robot orientado hacia el Este, lo que buscamos es el camino desde el nodo $[1,1,E]$ al nodo $[n,n,E]$. Este algoritmo nos asegura que el camino encontrado es el más corto que resuelve el grafo. Para que Dijkstra resuelva el grafo hay que indicarle un nodo de partida y otro de llegada, y como por cada celda de la matriz hay cuatro nodos (uno para cada orientación), entonces podemos seleccionar la orientación inicial que tiene el robot al comenzar la navegación y también la dirección a la que terminará mirando el robot.

Finalmente transformamos el camino contenido en el grafo a otro contenido en la matriz. Este proceso es sencillo gracias a que, como ya sabemos, a cada celda de la matriz le corresponden cuatro nodos del grafo, y a cada nodo presente en el grafo le corresponde una sola celda. Por lo tanto, es cuestión de mapear los nodos del camino a celdas y borrar las celdas repetidas. Estas ultimas se dan en el caso de los giros, dado que el robot no cambia de celda, sino que solo cambia de dirección.

En la implementación de este algoritmo de Path Finding decidimos darle el mismo peso a las aristas que realizan giros como a las que no, de forma que el algoritmo encuentre de los caminos más cortos que resuelven el mapa, el que tiene menos cantidad de giros. Si aumentaramos el peso de la aristas que representan giros a costo 3, Dijkstra preferiría desplazarse por dos aristas de costo 1 (movimientos que no agregan giros) antes que tomar una arista de costo 3 (movimiento que agrega giro). De esta forma priorizariamos aún más minimizar giros a minimizar longitud.

3.2.2. Resultados

Todas las ejecuciones de este algoritmo sobre nuestro set de 20 mapas obtuvieron resultados positivos, dado que en todos los mapas el algoritmo basado en Dijkstra encuentra caminos libres de colisiones. Estos caminos además son mínimos en cantidad de giros y longitud en cada mapa, por lo tanto, pueden ser considerados óptimos. En la Figura 3.3 se pueden apreciar cinco caminos obtenidos por este algoritmo. En ellos están especificados también los tiempos que tardaron para ser ejecutados una vez. Entre diferentes ejecuciones los tiempos no varían por más de 15 milisegundos, por esto los tiempos mostrados en la figura son muy representativos.

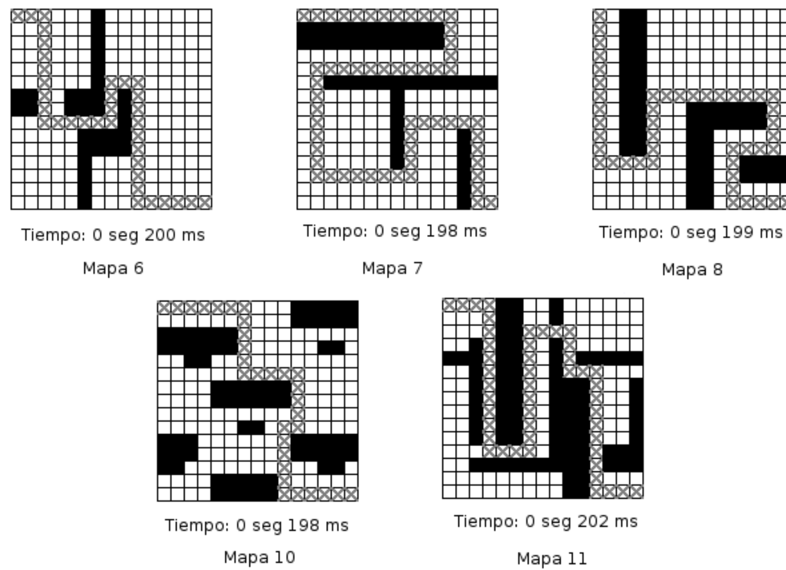


Figura 3.3: Caminos obtenidos por el algoritmo basado en Dijkstra junto con el tiempo de ejecución de cada uno.

En las ejecuciones de la Figura 3.3 le especificamos al algoritmo que se deseaba llegar al casillero $(15, 15)$ quedando el robot orientado al Este, siendo que el robot partía desde la casilla $(1, 1)$ mirando al Este. Esto provoca caminos diferentes a los mostrados en la Figura 3.1, dado que el algoritmo de Dijkstra crea el camino de forma de comenzar mirando al Este y terminar orientado con esa misma dirección y sentido, y a diferencia de esto, en el algoritmo dinámico, las direcciones de origen y finalización no son determinadas previamente, teniendo así la libertad de elegirlas pudiendo minimizar aún más giros.

3.3. Algoritmo basado en A^*

El algoritmo de búsqueda A^* (A Estrella) se clasifica dentro de los algoritmos de búsqueda en grafos y es básicamente una optimización heurística del algoritmo de Dijkstra. Presentado por primera vez en 1968 por Peter E. Hart, Nils J. Nilsson y Bertram Raphael, el algoritmo encuentra, siempre y cuando se cumplan unas determinadas condiciones, el camino de menor costo entre un nodo origen y uno destino.

El problema de algunos algoritmos de búsqueda en grafos, como puede ser el algoritmo voraz, es que se guían exclusivamente por la función heurística, la cual puede no indicar el camino de menor costo, o por el costo real de desplazarse de un nodo a otro, pudiéndose darse el caso en que sea necesario realizar un movimiento de mayor costo para alcanzar la solución. Por esto, es bastante intuitivo el hecho de que un buen algoritmo de búsqueda debería tener en cuenta ambos factores, el valor heurístico de los nodos y el costo real del recorrido.

Así, el algoritmo A^* utiliza una función de evaluación $f(n) = g(n)+h(n)$, donde $h(n)$ representa el valor heurístico del nodo a evaluar, y $g(n)$, el costo real del camino recorrido para llegar a dicho nodo. A^* mantiene dos estructuras de datos auxiliares, que podemos denominar abiertos, implementado como una cola de prioridad (ordenada por el valor $f(n)$ de cada nodo), y cerrados, donde se guarda la información de los nodos que ya han sido visitados. En cada paso del algoritmo, se expande el nodo que esté primero en abiertos, y en caso de que no sea un nodo objetivo, calcula la $f(n)$ de todos sus hijos, los inserta en abiertos, y pasa el nodo evaluado a cerrados.

El algoritmo es una combinación entre búsquedas BFS (Breadth First Search) con DFS (Depth First Search): mientras que $h(n)$ tiende a DFS, $g(n)$ tiende a BFS. De este modo, se cambia de camino de búsqueda cada vez que existen nodos más prometedores.

A^* es un algoritmo completo: en caso de existir una solución, siempre dará con ella.

3.3.1. Algoritmo Propuesto

Como A^* es un algoritmo de busqueda en grafos, nuevamente utilizamos la misma estrategia que en Dijkstra. Armando primero un grafo de la misma forma que en la sección 3.2.1. Donde en vez de ejecutar Dijkstra corremos A^* especificándole una función heurística h para optimizar la elección de nodos. Finalmente tomamos el resultado encontrado y lo transformamos de un camino en un grafo a otro representado en una matriz.

Como ya explicamos con más profundidad en la sección 3.2.1 el grafo

que armamos a partir de la representación del ambiente (una matriz) tiene cuatro nodos por cada celda de la matriz, uno por cada ubicación. Además, cada nodo esta conectado a otro por medio de una arista en el caso que estos son "vecinos" respecto a movimientos que puede realizar el robot desde el primer nodo. Estas conexiones forman el grafo que puede ser pensado como cuatro capas, donde cada una esta formada por los nodos de una determinada orientación. Cada nodo perteneciente a una capa tiene por vecinos a uno de la misma capa, que se encuentra ubicado en la dirección que indica la capa y a otros dos de diferentes capas (ver Figura 3.2 (b)). Esto sugiere la idea de una ubicación de los nodos igual a la de la matriz (ver Figura 3.2 (a)). Un casillero ocupado de la matriz, es reflejado en el grafo borrando todas las aristas que conectan cualquier nodo con los cuatro nodos que representan ese casillero. De esta forma evitamos que se encuentre un camino que pase por estos nodos, que al traducirlos generan un camino que pasa por casilleros ocupados.

Al algoritmo A^* le especificamos una función heurística $h(n)$ para mejorar el desempeño del mismo al momento de buscar un camino que resuelva el grafo. Lo que hace la función $h(n)$ es darle un nivel de preferencia al nodo n que se le ingresa. Para este algoritmo $h(n)$ devuelve la distancia del nodo n al nodo de destino. La misma se calcula de acuerdo a la cantidad de casilleros horizontales más verticales que distancian la celda que representa el nodo n con la celda que representa el nodo de destino. Entonces, h está definida de la siguiente forma:

$$h([i_k, j_k, P]) \quad = \quad \mid i_n - i_k \mid + \mid j_n - j_k \mid$$

donde el nodo a evaluar es $[i_k, j_k, P]$ y el nodo de destino es $[i_n, j_n, Q]$ con P y Q orientaciones cualesquiera.

Finalmente tomamos el camino encontrado por A^* que recorre el grafo y lo convertimos a uno que recorre la matriz. Para esto hacemos lo mismo que lo descripto en la sección 3.2.1. Esto es, usar el hecho de que cada celda de la matriz tiene cuatro nodos que la representan y cada nodo tiene una única celda que lo representa. Entonces lo que resta hacer es tomar por cada nodo del camino encontrado la celda que lo representa, evitando considerar celdas ya encontradas en el caso de los giros.

Con todo lo presentado sobre este algoritmo podemos ver que es parecido al de la sección anterior salvo que cambia la forma en que se eligen los nodos a evaluar. De aquí tenemos que los resultados que obtienen son los mismo que el de la sección anterior, salvo que en muchos casos estos serán obtenidos con menor costo computacional.

3.3.2. Resultados

Al ser A^* una modificación heurística del algoritmo de Dijkstra, los resultados son muy similares a los de la sección anterior. Se puede apreciar una pequeña disminución en los tiempo de ejecución provocadas por la optimización heurística, que evita buscar el camino hacia direcciones que se encuentran lejanas al punto de llegada. En la Figura 3.4 se muestran cinco caminos obtenidos por este algoritmo.

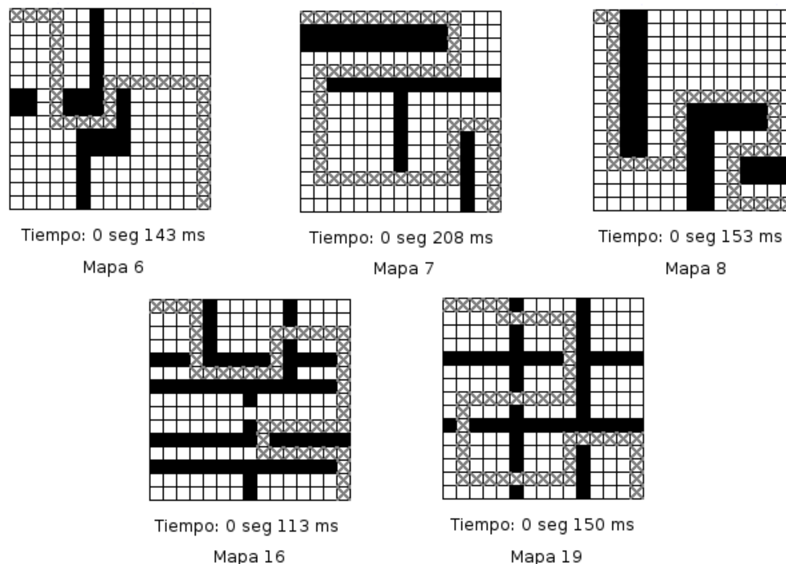


Figura 3.4: Caminos obtenidos por el algoritmo basado en A^* junto con el tiempo de ejecución de cada uno.

Estos caminos son diferentes a los encontrados por el algoritmo basado en Dijkstra mostrados en la Figura 3.3. Esta diferencia está dada por la función heurística h que elije los nodos a visitar según su cercanía al punto de destino. Por ejemplo, en el camino del Mapa 8 generado por A^*, se puede ver que el primer movimiento del robot es desplazarse de la casilla $(1, 1)$ a la $(1, 2)$ acercándose en distancia al punto de llegada; a diferencia del camino del Mapa 8 generado por Dijkstra, en el cual el robot lo primero que hace es girar en la casilla $(1, 1)$ sin acercarse en ese movimiento al punto de llegada.

El algoritmo resolvió los 20 mapas de nuestro set. En todos estos obtuvo caminos óptimos en giros y longitud. Mejora también en forma mínima los tiempos de ejecución. Además al igual que Dijkstra permite poder especificarle la orientación de inicio y fin.

3.4. Comparación de los algoritmos

Podemos comparar los algoritmos de esta sección viendo los tiempos de ejecución de cada algoritmo. Hay una gran diferencia entre el tiempo que consume la ejecución de el algoritmo dinámico y el tiempo de ejecución que consumen tanto la modificación de Dijkstra como la modificación de A^*. La causa más importante de esta diferencia, es que el algoritmo dinámico tiene que ser ejecutado tantas veces como giros tiene el camino mínimo del mapa, por lo tanto, para encontrar un camino que tiene 7 giros, el algorito dinámico debe ser corrido 7 veces mientras que las modificaciones de A^* y Dijkstra son corridas solo una vez. Además, el algoritmo dinámico no termina su ejecución al momento de encontrar la solución. En cambio A^* y Dijkstra terminan una vez que alcanzan el nodo de destino deseado.

No encontramos una gran diferencia entre los tiempos de ejecución de Dijkstra y A^* en nuestro set de mapas. Para hacer más evidente la diferencia existente entre estos algoritmos mostramos las celdas recorridas por cada uno de los algoritmos para un par de ejemplos. En estos solo le pediremos a los algoritmos que encuentren caminos de longitud mínima, para simplificar la visualización de los resultados.

Como primer ejemplo exhibimos dos matrices donde marcamos los casilleros correspondientes a los nodos visitados por los algoritmos. En la Figura 3.5 se observa que se recorren muchas más celdas que en la Figura 3.6 en un mapa libre de obstáculos, dejando en evidencia que la modificación heurística de Dijkstra hace efectivamente lo que deseábamos, que es priorizar los nodos que se encuentran más cerca del punto de destino. En este mapa es muy notable la diferencia entre los dos algoritmos, esto se debe a que no hay obstáculos en el ambiente, en cambio, si el ambiente no se encuentra libre de obstáculos, las diferencias entre las ejecuciones de ambos algoritmos cambia bastante.

Como segundo ejemplo, mostramos la ejecución que producen ambos algoritmos en un mapa donde hay una "trampa". Decimos que en el mapa hay una trampa porque los obstáculos fueron colocados de forma tal que el algoritmo de Dijkstra tenga que evaluar los nodos que se encuentran dentro de la trampa, para luego evaluar los que se encuentran fuera de ella (Figura 3.7). Por su parte A^*, también cae en la trampa, dado que primero intenta encontrar una solución utilizando los nodos que se encuentran dentro de la misma, para luego buscar la solución fuera de la trampa. Pero gracias a la función heurística de A^*, primero son considerados los nodos más cercanos al punto de llegada, lo que provoca que en definitiva sean muchos menos los nodos visitados antes de alcanzar el destino (ver Figura 3.8).

A^* y Dijkstra superan ampliamente al algoritmo Dinámico en cuestión

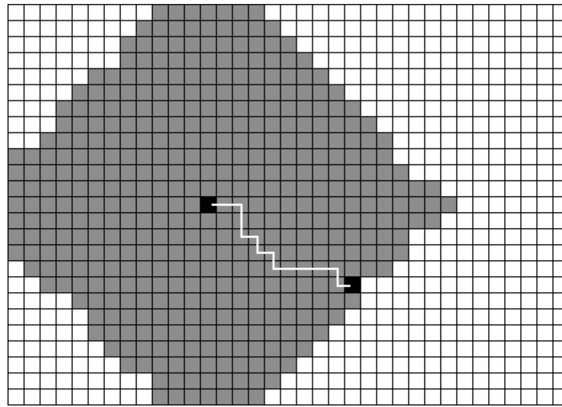


Figura 3.5: Ejemplo de una ejecución de Dijkstra.

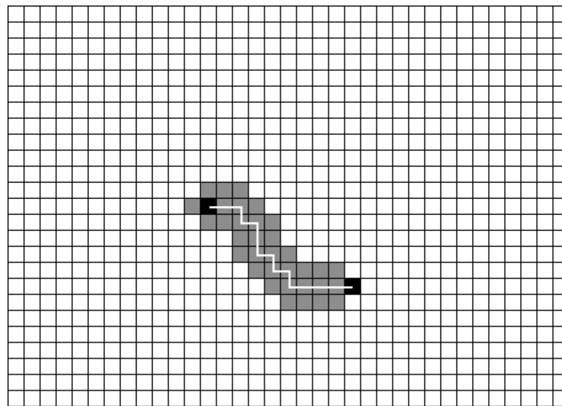


Figura 3.6: Ejemplo de una ejecución de A^*

de tiempos de ejecución. A su vez A^* mejora los tiempos de ejecución de Dijkstra, por esto decidimos que A^* sea el algoritmo elegido para realizar Path Finding determinístico en el sistema de navegación.

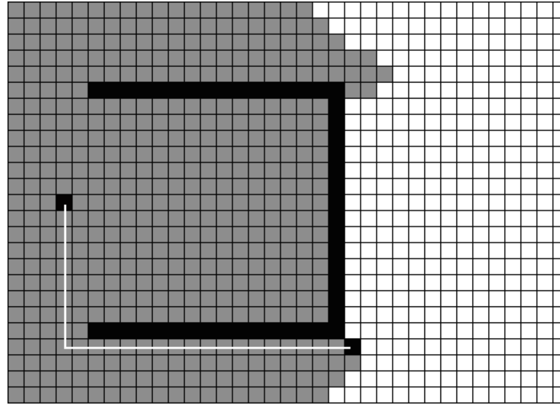


Figura 3.7: Ejemplo de una ejecución de Dijkstra en un mapa con obstáculos

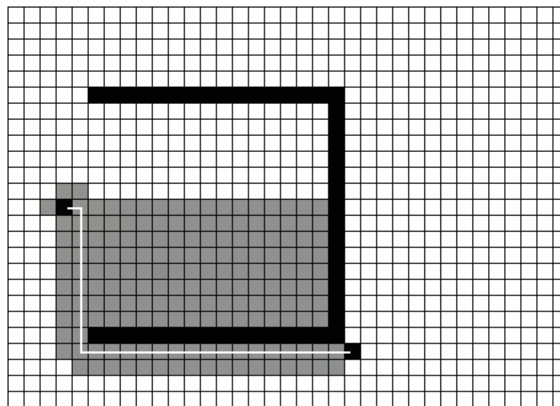


Figura 3.8: Ejemplo de una ejecución de A^* en un mapa con obstáculos.

Capítulo 4

Path Executing

En este capítulo presentamos los problemas del uso de la plataforma durante la navegación, como así también las soluciones que proponemos a estos problemas. A las estrategias tomadas para el uso y solución de problemas las llamamos Path Executing.

Path Executing es un conjunto de operaciones que trabajan con los instrumentos de movimiento y ubicación del robot. La tarea de Path Executing es evitar que el robot se pierda, y asegurar que la plataforma siempre se encuentra en el lugar del ambiente donde el robot (a través de una representación interna del ambiente) cree que se encuentra.

Para esta parte del sistema utilizamos un robot en particular (ver Figura 4.1). Este tiene determinados tipos de elementos mecánicos y electrónicos, como así también sensores con determinadas características. Los problemas que tratamos y las soluciones que proponemos están basadas en este robot (Omega). Mas allá de las particularidades especificas del hardware podemos afirmar que los problemas y estrategias propuestas pueden ser generalizadas.

Inicialmente en este capítulo, presentamos el robot utilizado en los experimentos, y detallamos los componentes de hardware que posee. Luego, describimos los problemas que presenta el uso de dichos componentes. A continuación, exponemos y explicamos las soluciones provistas por Path Executing para el manejo del hardware del robot. Finalmente analizamos las ventajas y desventajas de cada operación de Path Executing.

4.1. Robot utilizado

Para los experimentos y la implementación de esta parte del sistema utilizamos el robot Omega (Figura 4.1). Este es una plataforma móvil de tres ruedas, dos delanteras que proveen tracción y una tercer rueda "loca" o libre,

ubicada en la parte posterior (Figura 4.2). En el interior del robot se encuentra una PC estándar que corre el sistema operativo Linux, sobre el cual se ejecuta el sistema de navegación.

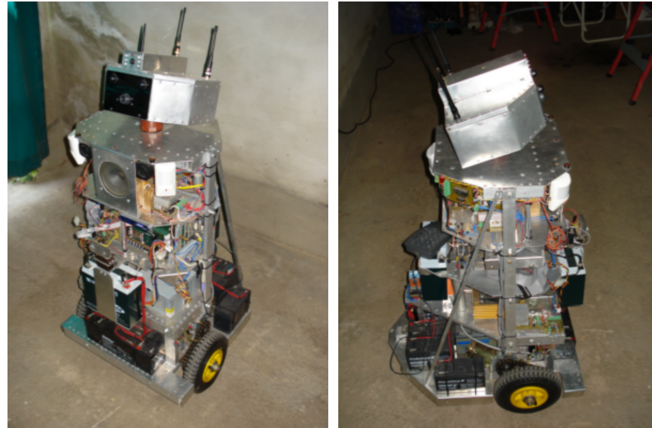


Figura 4.1: Fotos del robot Omega usado para la implementación del sistema de navegación.

Omega cuenta con dos ruedas impulsadas independientemente por dos motores. Los motores, a pesar de ser iguales, funcionan con diferentes potencias a causa de errores propios del hardware utilizado para administrarles tensión. La velocidad y dirección de rotación de cada motor puede ser establecida independientemente. Se pueden establecer 6 tipos diferentes de velocidades (de 0 a 5). Así podemos tener un motor girando en un sentido a velocidad 5 mientras el otro se encuentra girando en sentido inverso a una velocidad 2.

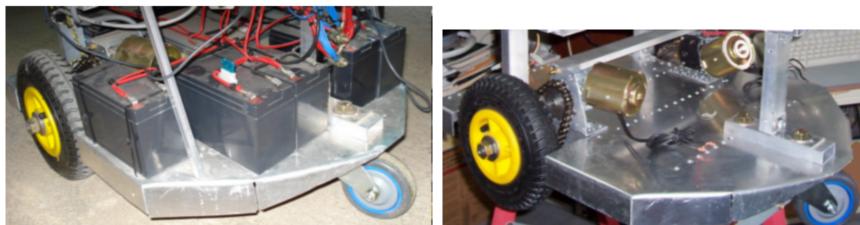


Figura 4.2: Foto de los motores de tracción y la rueda "loca" trasera. A la derecha se puede apreciar las diferencia de ubicación de los motores

El robot también cuenta con dos odómetros, que se encuentran ubicados al costado de las dos ruedas de tracción. Un odómetro es un dispositivo

que indica la distancia recorrida en viaje por un vehículo, en nuestro caso cada odómetro indica la distancia recorrida por la rueda en que se encuentra ubicado. Cada uno de estos artefactos puede determinar por separado la distancia que se desplazo cada rueda.

En el frente del robot se encuentra un sensor de ultrasonido, comúnmente denominado sónar. Es un sistema de localización similar a los radares, utiliza ondas de ultrasonido para determinar la distancia a un objeto. Tanto el emisor de ultrasonido como el receptor de ultrasonido están montados sobre un servo. Un servo (también llamado servomotor), es un dispositivo capaz de ubicarse en cualquier posición de su rango de operación y mantenerse estable en ese lugar. Combinando el servo con el sónar podemos detectar objetos en un rango de al rededor de 180 grados.

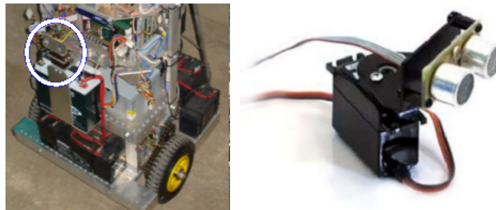


Figura 4.3: sónar de Omega. A la izquierda se puede apreciar la ubicación del sónar en el robot. A la derecha, los sensores de ultrasonido montados sobre el servomotor.

4.2. Problemas

Por más que el sistema pueda planear un camino realizable en un ambiente, existen muchos problemas al momento de recorrerlo. Para poder desplazar la plataforma por un camino, es necesario que la representación interna de la ubicación y orientación del robot coincidan con la realidad en cada paso del camino. Una forma muy sensilla de lograr esto es averiguar la posición del robot en cada paso del camino y si ocurre alguna desviación en la trayectoria, reubicar la plataformar.

Debemos tener en cuenta que en ambientes cerrados convencionales, como una casa, edificio u oficina, no es posible utilizar un sistema externo que nos indique donde estamos ubicados precisamente, sin adaptar artefactos ajenos a las habitaciones. En ambientes abiertos existe el sistema de posicionamiento global (G.P.S), el cual permite determinar en todo el mundo la posición de un objeto, una persona, un vehículo o una nave. El sistema G.P.S. utiliza señales

satelitales para determinar la ubicación, pero estas no pueden penetrar en ambientes cerrados, imposibilitando esto el uso de dicho sistema en nuestro trabajo.

Como no es posible establecer la ubicación del robot en cada paso de un camino, el sistema debe poder ejecutar movimientos confiables que le aseguren que la ubicación del robot es la esperada por el sistema. Es decir, si el robot se encuentra inicialmente en una ubicación y una orientación conocidas, y se le indica que avance hacia delante un metro, es necesario que esta orden se cumpla de la forma más precisa posible, debido a que el sistema supondrá que la nueva ubicación del robot se encuentra desplazada un metro de la ubicación anterior en la dirección en que se encontraba orientado el robot.

El robot para poder realizar diferentes tipos de movimientos se vale de los motores para desplazarse, de los odómetros para determinar la distancia que se desplazo y de el sónar para determinar la distancia a los objetos. Estos artefactos poseen imperfecciones que provocan errores al momento de utilizarlos, causando desfasajes entre la representación interna de la ubicación del robot y la realidad.

Podemos diferenciar tres grandes problemas al hacer uso de la plataforma:

Potencia de los motores: el robot posee dos motores de tracción, los cuales tienen diferentes potencias. Esto provoca que por más que a ambos motores se les ordene funcionar a una misma velocidad, uno gira más rápido que otro. Esta diferencia de velocidades en los motores de tracción provoca que el robot no realice un movimiento rectilíneo, sino que se desvía hacia el lado del motor más lento, realizando este una trayectoria curva.

Precisión de los odómetros: los odómetros tienen una unidad mínima de medida de $2,4cm$, causando un margen de error de aproximadamente $\pm 2,3cm$. Entonces puede suceder que al usar los odómetros para desplazar el robot a $50cm$ de distancia en un movimiento, en realidad lo hayamos desplazado unos $48cm$. Esto puede parecer poco significativo, pero debemos tener en cuenta que esta clase de error es acumulativa. Es decir que si en vez de avanzar el robot directamente $50cm$ lo que hacemos es avanzarlo primero $25cm$ y luego $25cm$, en realidad podríamos haber desplazado el robot sólo $46cm$.

Precisión del Sónar: el sónar tiene establecida la unidad mínima para diferenciar distancias en $1cm$, por lo que los datos obtenidos pueden contener un error de $\pm 0,9cm$. Además, el sónar no puede diferenciar obstáculos que se encuentren a más de $255cm$ o a menos de $28cm$.

Otros problemas adicionales que deben ser tenidos en cuenta al utilizar el sensor de ultrasonido:

- El eco que se recibe como respuesta a la reflexión del sonido indica la presencia del objeto más cercano que se encuentra dentro del cono acústico y no especifica en ningún momento la localización angular del mismo (ver Figura.4.4 (a)).
- La cantidad de energía acústica reflejada por el obstáculo depende en gran medida de la estructura de su superficie.
- la densidad del aire depende de la temperatura, influyendo este factor sobre la velocidad de propagación de la onda de ultrasonido.
- Las ondas de ultrasonido obedecen a las leyes de reflexión de las ondas, por lo que una onda de ultrasonido tiene el mismo ángulo de incidencia y reflexión respecto a la normal de la superficie (ver Figura.4.4 (b)).

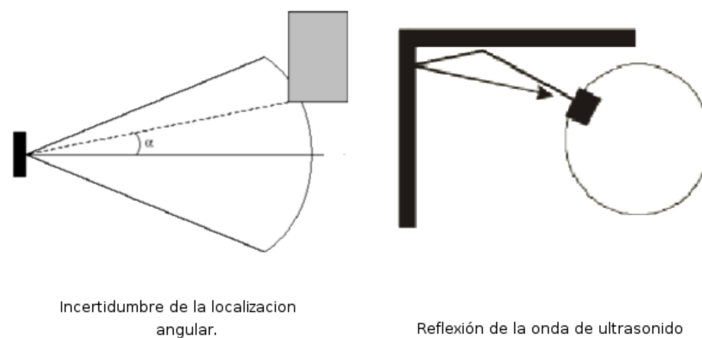


Figura 4.4: Problemas en el uso del sónar.

4.3. Soluciones

Para solucionar los tres problemas de hardware mencionados, limitamos el uso de la plataforma a cuatro directivas o acciones llamadas Basic Forward, Wall Stop, Wall Following y Turn. El sistema solo se vale de las mismas para desplazar el robot. Las acciones o directivas permiten desplazar el robot hacia delante o hacerlo girar, las cuales le dan una movilidad suficiente al robot para poder recorrer cualquier camino planeado por el sistema.

A continuación asociamos los problemas de hardware a las acciones que intentan resolverlos. Para luego describir en detalle el funcionamiento de las mismas:

Potencia de los motores: este problema se evidencia cuando intentamos hacer avanzar el robot en línea recta. Para solucionarlo Path Executing provee al sistema de las directivas de Basic Forward y Wall Following, las cuales permiten desplazar el robot hacia delante en línea recta una distancia determinada, tratando de evitar los desvíos causados por las diferencias de potencias en los motores.

Precisión de los odómetros: los odómetros son usados para medir la distancia recorrida, y los errores causados por los mismos se ven reflejados al momento de decidir si ya fue recorrida una cierta distancia. Para lidiar con este problema Path Executing provee al sistema de la directiva Wall Stop. La cual permite desplazar el robot hacia delante una determinada distancia, haciendo uso de objetos presentes en el ambiente para determinar la distancia donde debe finalizar el movimiento.

Precisión del sónar: los errores en el uso del sónar dependen en gran medida de la forma en que es usado. Si el sónar es utilizado para medir la distancia a un objeto plano, y no se interponen obstáculos u otros objetos entre los sensores y el objetivo, obtenemos resultados con errores menores al centímetro, esta precisión es suficiente para la utilización que pretendemos hacer del sónar. Por lo tanto, Path Executing evita los problemas del uso del sónar, solicitando al sistema que solo utilice las operaciones que hacen uso del sónar cuando es seguro hacerlo. Es decir, cuando se tiene la seguridad de que no hay objetos cercanos que puedan alterar la mediciones del sónar. La operaciones que hacen uso del sónar son Wall Stop y Wall Following.

Entre las acciones que provee Path Executing al sistema se encuentra la directiva Turn, la cual permite girar la plataforma. Esta acción no pretende solucionar ningún error específico de hardware, sino que solo intenta proveer al sistema de una operación que le permita girar de la forma más precisa conservando la ubicación del robot.

A continuación presentamos cada una de las cuatro acciones de las que se vale el sistema para desplazar la plataforma.

4.3.1. Basic Forward

Esta acción intenta proveer al sistema un movimiento rectilíneo hacia delante. A la misma se le especifica la distancia que se desea recorrer, y con

esta información Basic Forward hará desplazar la plataforma hacia delante lo más recto posible hasta haber recorrido la distancia indicada.

Como ya mencionamos anteriormente, los motores de Omega tienen diferentes potencias. Esto causa, que al indicarle a ambos que se pongan en avance a la misma velocidad, obtengamos que uno gire más rápido que otro. Describiendo el robot en el avance un movimiento curvo.

Para evitar la curvatura en el desplazamiento, Basic Forward se vale de los odómetros que le indican el avance individual de cada rueda. La estrategia consiste en no permitir que una rueda avance más que la otra. Para esto, durante el desplazamiento se van leyendo los odómetros para corroborar que los dos se incrementen de la misma manera. En el caso en que un odómetro indique una distancia recorrida por una rueda mayor que la otra, Basic Forward desacelerara dicha rueda y dejará la otra funcionando a velocidad normal, hasta que ambos odómetros vuelvan a marcar la misma distancia. En ese momento se le volverá a indicarle a ambos motores que vuelvan a funcionar a la misma velocidad. Finalmente, Basic Forward detiene los motores cuando alguno de los dos odómetros alcanza la distancia indicada.

4.3.2. Wall Stop

Esta acción intenta proveer al sistema de un movimiento rectilíneo hacia delante que deja ubicado al robot cerca de un objeto detectable por el sónar (como es el caso de una pared). A Wall Stop se le especifica la distancia a la pared a la que se desea dejar ubicado el robot. Con esta información y con la suposición de que en frente del robot se encuentra una pared, Wall Stop desplaza la plataforma hasta alcanzar distancia indicada.

Para la correcta ejecución de esta acción se requiere que en frente del robot, a menos de $255cm$ y más de $28cm$ se encuentre un objeto detectable por el sónar. Además se pide que no haya algún objeto extra que pueda alterar el uso del sónar. Esta tarea es designada a la parte del sistema que haga uso de la directiva Wall Stop. Así se evitan los problemas del uso del sónar.

Los odómetros utilizados en Omega, al tener una unidad de medida mínima grande, provocan discrepancias entre la representación interna del robot y la realidad. El desfasaje de la representación se puede evidenciar al momento de querer aproximar la plataforma a una pared. Si este desfasaje no es tenido en cuenta al momento de acercar al robot a la pared se puede provocar una colisión.

Para evitar esta colisión y para reparar posibles desfasajes en la representación, Wall Stop se vale del sónar y de la acción de Basic Forward para desplazar la plataforma. La estrategia de esta directiva consiste en arrimarse

a la pared con un movimiento generado por Basic Forward (recto), pero a diferencia de la primitiva expuesta anteriormente, usa el sónar para no colisionar, e intenta dejar el robot lo más cerca posible de la distancia deseada.

Para lograr esto, lo que inicialmente hace Wall Stop es identificar la distancia a la pared (u objeto detectable) para así determinar cuanto recorrido puede realizar de forma segura usando Basic Forward. Una vez hecho esto, ejecuta la directiva de Basic Forward con la distancia estableciada. El uso de esta acción garantiza que la desviación al desplazar el robot será mínima y se obtendrá un movimiento recto. Este proceso es repetido mientras el chequeo de que la distancia que existe entre el robot y el objetivo es segura para realizar Basic Forward. Decimos que la distancia es segura para realizar Basic Forward si hay más de $10cm$ a la distancia deseada con el objetivo. Finalmente cuando se determina que ya no es seguro avanzar con Basic Forward (el robot se encuentra a menos de $10cm$ de estar ubicado correctamente), se comienza a realizar un movimiento "continuo", el cual consiste en poner los motores en funcionamiento e ir leyendo el sónar permanentemente mientras se avanza hasta llegar a la distancia establecida. En este punto se detiene la marcha y se verifica la distancia a la pared, siempre utilizando el sónar como instrumento de medición, en el caso en que la plataforma se haya excedido de la distancia deseada, el robot retrocede hasta quedar en posición.

4.3.3. Wall Following

Wall Following es una acción que provee al sistema la capacidad de hacer que el robot "siga una pared", o en otras palabras, que el robot se desplace de forma paralela a un objeto detectable por el sónar. Para esto la acción requiere que el sistema le indique a que lado del robot se encuentra la pared, la separación que debe mantener el robot con la pared durante el desplazamiento, y la distancia que se pretende avanzar. Luego, Wall Following hace uso de esa información para desplazar la plataforma en forma paralela a la pared manteniendo la separación indicada.

Wall Following supone que efectivamente hay un objeto detectable del lado del robot indicado, por lo que es crucial la presencia del mismo para una ejecución correcta de esta directiva. Nuevamente Path Executing delega esta responsabilidad a la parte del sistema que ejecuta la acción de Wall Following.

La diferencia de potencia en los motores de Omega pueden causar un desvio en la trayectoria del robot. A causa de esto, al momento de desplazar el robot de manera paralela a una pared o a un objeto, el sistema corre el riesgo de colisionar con ellos o de terminar más alejado de lo planeado de los mismos.

También puede suceder que el robot no este lo suficientemente paraleo a la pared al momento de iniciar un movimiento recto y paralelo a la misma. Esto podría llegar a causar una colisión o un desfasaje de la ubicación real del robot con la representación interna.

Para evitar esta clase de problemas Wall Following va verificando la distancia que separa al robot de la pared durante todo el desplazamiento. La estrategia consiste en utilizar dicha distancia para saber como modificar las velocidades de los motores, de forma de mantener al robot separado de la pared la distancia especificada por el sistema.

Al comenzar la ejecución de la directiva Wall Following, primero se ubica el sónar apuntando al lado de la plataforma donde se encuentra la pared. Luego, se comienza el desplazamiento hacia delante y a su vez se comienza a tomar medidas con el sónar. Durante el avance, Wall Following va comparando la distancia a la pared que va obteniendo con la que le especificó mantener el sistema. En el caso en que la distancia obtenida sea mayor a la separación especificada, lo que la directiva hace es disminuir la velocidad del motor del lado de la pared para desviar el movimiento hacia la pared. En el caso contrario (en que la distancia obtenida es mayor a la especificada) la estrategia aplicada es la inversa, es decir, se disminuye la velocidad de la rueda opuesta a la pared para desviar la dirección del robot, alejandolo así de la pared y acercandolo a la separación especificada. Por ultimo, cuando alguno de los dos odómetros indica que se alcanzo la distancia que desea avanzar el sistema, Wall Following detiene los motores y finaliza la acción.

En la Figura 4.5 ilustramos el funcionamiento de la directiva Wall Following. En la imagen aprecia una vista esquemática superior del robot Omega junto el rastro del movimiento que se encuentra realizando.

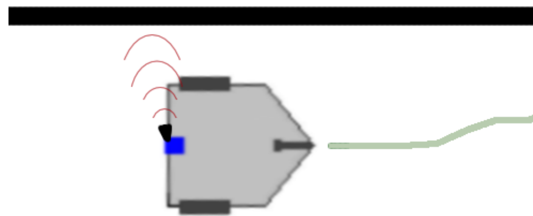


Figura 4.5: En la Imagen se aprecia una vista esquemática superior de Omega, y se ilustra el funcionamiento de Wall Following.

4.3.4. Turn

Esta directiva, intenta proveer al sistema de una acción que le permita cambiar la orientación de la plataforma en 90 grados sin modificar su ubicación. A Turn solo se le indica hacia que lado se desea realizar el giro (derecha o izquierda). Con esta información se rota la plataforma hacia la derecha o izquierda hasta lograr un giro lo más parecido a uno de 90 grados, sin perder la ubicación que tenia el robot antes de usar la directiva.

El problema de esta acción se basa en que Omega es una plataforma móvil que tiene dos motores de tracción y una rueda "loca" trasera, la cual esta libre. Esto causa que al hacer girar los motores de Omega en sentidos opuestos obtengamos que el robot gire sobre el centro del eje que une las ruedas delanteras. Si consideramos que el centro del robot para nosotros se encuentra en el centro de la plataforma esto nos provoca que al girar la plataforma no conserve su ubicación. En la figura 4.6 se muestra la ubicación de ambos centros de giro de Omega.

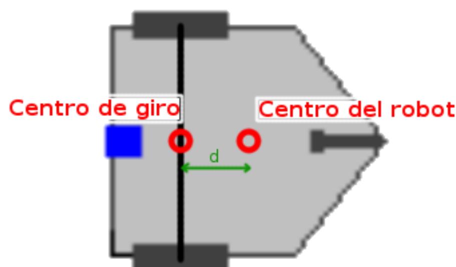


Figura 4.6: En la figura se aprecia una vista esquemática superior de la plataforma, donde se individualizan los centros de giro del Omega.

Para evitar el problema de no perder la ubicación original del robot al girar, la directiva Turn se vale de operaciones extras para lograr un cambio de orientación. La estrategia consiste en mover el robot hasta hacer coincidir el centro de giro de la plataforma con la ubicación original del centro del robot. Luego girando a 90 grados hacia la dirección indicada, y finalmente haciendo avanzar el robot la misma distancia que se retrocedió en el desplazamiento inicial.

Para que el centro de giro y el centro de la plataforma coincidan, haciendo uso de los odómetros la operación debe retroceder la plataforma la distancia que separa a ambos centros (indicado como d en la Figura 4.6). Luego, debe rotar la plataforma en el sentido indicado. En el caso en que se le haya indicado girar a la derecha, se pone el motor izquierdo en avance y el derecho

en retroceso, ambos a la misma velocidad. En caso de pretender girar a la izquierda se realiza el proceso inverso. Para girar la plataforma a 90 grados, se colocan los odómetros en cero y se inicia el giro. Los motores se detienen cuando alguno de los odómetros consigue una distancia g, la cual fue obtenida mediante experimentación y asegura que el robot ha girado aproximadamente 90 grados. A continuación, Turn debe avanzar la plataforma la distancia que retrocedió en el paso inicial (d).

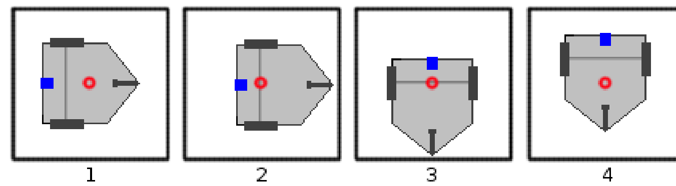


Figura 4.7: Breve esquema de las acciones que realiza la directiva Turn para rotar el robot cuando se encuentra orientado al Oeste y se desea que gire a su derecha, quedando orientado al Norte.

4.4. Resultados

En esta sección mostramos las ventajas y desventajas encontradas durante las pruebas de desempeño que realizamos sobre cada una de las cuatro directivas provistas por Path Executing.

Basic Forward Según los experimentos realizados sobre Omega, Basic Forward tiene la ventaja de introducir menos errores que si se usaría directamente el hardware de lo motores. En la Figura 4.8 se muestra dos imágenes que comparan el movimiento generado por Basic Forward y el movimiento generado al poner los motores en velocidad constante hacia delante. En la figura se puede apreciar la mejora que realiza Basic Forward.

Como desventaja tenemos que Basic Forward utiliza los odómetros para corregir errores, los cuales a su vez poseen errores de medición. Esto provoca que Basic Forward pueda generar un desfasaje con respecto a la representación interna de la ubicación. Por otro lado, esta directiva no es capaz de eliminar errores acumulados en los movimientos previos dado que no se vale de elementos del ambiente para determinar la ubicación final del robot.

Wall Stop Según los resultados obtenidos mediante experimentación, Wall Stop nos asegura que al finalizar su ejecución el robot se encontrará a la dis-

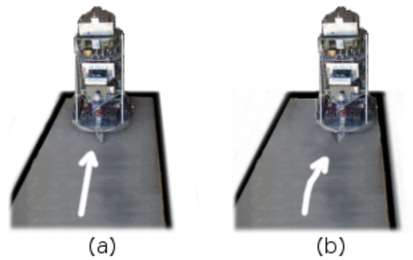


Figura 4.8: (a) Movimiento obtenido por Basic Forward. (b) Movimiento obtenido por el uso directo del hardware.

tancia esperada a la pared. A diferencia de Basic Forward esta acción puede corregir errores acumulados en la representación interna de la ubicación, dado que utiliza a objetos del medio para establecer la ubicación final del robot.

Como desventaja tenemos que este uso de objetos del ambiente limita al sistema a utilizar Wall Stop solo en el caso que este requiere aproximarse a una pared u objeto. Además esta acción no es capaz de corregir errores en la orientación del robot.

Wall Following Como ventaja, al igual que Wall Stop, esta acción demostró mediante experimentación poder corregir errores en la ubicación del robot, dado que se vale de objetos del medio para determinar donde reubicar el robot. Además si esta acción es usada en forma correcta, permite volver a orientar correctamente el robot, eliminando así errores de orientación acumulados.

Wall Following al hacer uso de objetos ubicados en el ambiente limita al sistema a utilizar este movimiento solo para desplazamientos paralelos a paredes u objetos. Por otra parte, al intentar mantener el robot a una distancia dada a la pared, esta acción puede causar un zigzagueo producido por las aceleraciones y desaceleraciones de los motores, lo cual puede causar que la distancia del punto inicial al final del movimiento sea menor a que la marcada por los odómetros, resultando esto en un desfasaje en la representación de la ubicación.

Turn Esta operación tiene como ventaja el hecho de que permite girar el robot sin alterar su ubicación. Mediante experimentación pudimos determinar que es lo suficientemente buena como para suponer que después de girar el robot se encuentra aproximadamente en la misma ubicación.

Como desventaja, Turn, puede agregar errores asociados al uso de los

odómetros. Además, los problemas generados por la discrepancia de potencias en los motores pueden desviar las trayectorias estimadas al retroceder y avanzar para completar el giro.

Capítulo 5

Path Planning

Path Planning es la parte del sistema que se encarga de hacer uso de Path Finding y Path Executing para así llevar a cabo la navegación del robot. Path Planning por un lado utiliza a Path Finding para planear la ruta a seguir y por otro lado hace uso de Path Executing para mover la plataforma a lo largo de la ruta.

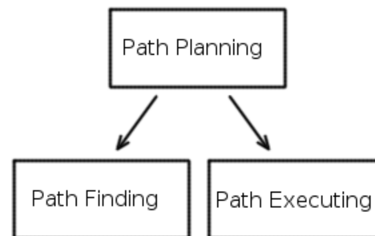


Figura 5.1: Esquema de uso entre Path Planning, Path Executing y Path Finding

Existen dos clases de Path Planning: Global Path Planning y Local Path Planning. Global Path Planning es un algoritmo que resuelve el problema de encontrar un camino libre de colisiones en un terreno estático, donde todos los obstáculos del ambiente son conocidos. En Local Path Planning, el planeamiento es realizado mientras el robot se encuentra en movimiento, en otras palabras, el algoritmo es capaz de producir un nuevo camino como respuesta a cambios en el ambiente. El Path Planning utilizado en este sistema es local Path Planning.

En el capitulo se muestra nuestra propuesta de Path Planning. Para llegar al resultado final de Path Planning, pasamos por dos etapas. La primer etapa es realizar el Path Planning simulando el ambiente, y simulando el uso de

una plataforma. La segunda etapa consiste en realizar Path Planning en el ambiente real con el robot. Para finalizar, mostramos los resultados obtenidos con nuestro Path Planning.

5.1. Path Planning propuesto

Nuestro Path Planning hace uso de Path Finding y Path Executing presentados previamente en el trabajo. Path Planning utiliza Path Finding para encontrar un camino y emplea Path Executing para recorrerlo.

Para realizar el Path Planning es indispensable una representación del ambiente ó mapa, un punto de origen, un punto de destino y el tamaño de grillado del mapa. Tanto el mapa como el punto de origen y el punto de destino se describen en la Sección 2 de este trabajo. Las grillas se suponen cuadradas, por lo tanto solo es necesaria la dimensión de uno de sus lados para determinar el tamaño de las grillas del mapa. Path Planning también supone que pueden existir obstáculos desconocidos en el ambiente y el mismo se mantiene estático durante la ejecución de Path Planning.

Path Planning utiliza la información de la representación para determinar la estrategia a seguir. Lo primero que Path Planning debe realizar es obtener un camino desde el punto de origen al punto de destino evitando los obstáculos existentes en la representación del ambiente, además, el camino debe optimizar giros y minimizar la distancia recorrida entre estos dos puntos. Para esto hace uso de algún algoritmo de Path Finding. Utilizamos dos algoritmos de Path Finding para realizar el Path Planning, un algoritmo determinístico y uno no determinístico. El algoritmo no determinístico utilizado es nuestro GA, mientras que el algoritmo determinístico usado es la modificación de A^*.

Como Path Planning debe evitar colisionar durante todo el planeamiento, no es suficiente evitar chocar con los obstáculos conocidos de la habitación, sino que también debe evitar colisiones con los obstáculos desconocidos del lugar. Para cumplir con esto, Path Planning recorre el camino encontrado a través de Path Finding, si se topa con un obstáculo no identificado en la representación detiene el movimiento de la plataforma. Como el obstáculo se encuentra en el camino, el mismo ya no es útil para seguir con el planeamiento, por lo tanto, Path Planning debe encontrar un nuevo camino que comience en la posición donde se encuentra detenido el robot y llegue hasta el punto final deseado. Para lograr esto, reconoce el obstáculo en la representación del ambiente y Path Finding utiliza este nuevo mapa para encontrar un nuevo camino. Path Planning repite este proceso hasta que eventualmente llega al punto deseado, o hasta que determina que no existen caminos que

eviten colisiones.

En la Figura 5.2 se puede observar una ejecución de Path Planning utilizando A^* como algoritmo de Path Finding. Además, en la figura, se puede apreciar un circulo, el cual simula un obstáculo que no esta presente en la representación del ambiente, pero si en el ambiente. Las cruces negras son obstáculos no identificados en la representación con los que se encontró Path Planning. En la figura hay dos cruces negras, que nos indican que Path Planning se encontró con obstáculos desconocidos dos veces mientras recorría el camino encontrado.

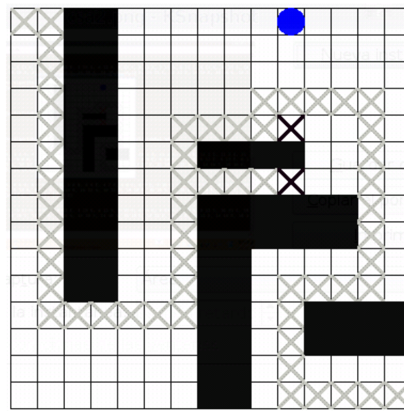


Figura 5.2: Ejemplo de una ejecución de Path Planning, evitando colisiones con todos los obstáculos del ambiente.

Para recorrer el camino encontrado a través del mapa, es indispensable que se mantenga correlación entre la posición real del robot y la posición en la representación del ambiente en la que se encuentra. Si esta correlación no es conservada, el robot puede no terminar en el lugar de destino deseado. También es necesario que la posición y dirección inicial del robot sean las mismas que en la representación. La precisión de los movimientos de cada plataforma depende tanto de los elementos mecánicos, como de los elementos de ubicación presentes en la misma. Nuestro Path Planning utiliza las primitivas de Path Executing para realizar correcciones de errores de desplazamiento y para conseguir movimientos con menos errores. Algunas de las primitivas brindadas por Path Executing tienen la virtud de aprovechar los elementos presentes del ambiente para realizar correcciones, y es Path Planning quien debe reconocer que método de Path Executing le conviene utilizar en cada momento de la ejecución.

5.2. Simulación de Path Planning

Para realizar la simulación de Path Planning es necesario encontrar un camino y simular tanto la plataforma como los obstáculos desconocidos del ambiente. La plataforma es simulada suponiendo que realiza los movimientos de manera perfecta, traduciendolos en primitivas de Path Executing pero sin utilizarlas. Los obstáculos desconocidos del ambiente son marcados en la representación pero al principio no son tenidos en cuenta como tales para la busqueda de caminos. Si mientras se simula el recorrido del camino la plataforma se topa con un obstáculo desconocido, se detiene la ejecución, y desde ese momento en adelante el obstáculo es tenido en cuenta como tal en la busqueda de nuevos caminos. Una vez que se completa el Path Planning, ya sea de manera exitosa (llegando al punto de destino) o no exitosa (no llegando al punto de destino), el mapa es vuelto a su condición original perdiendo noción de aquellos obstáculos encontrados durante este recorrido. Procedemos de esta forma porque no podemos asegurar que los obstáculos en un principio desconocidos del ambiente se mantendrán siempre en el lugar en el que fueron encontrados. Por ejemplo, si la plataforma se encuentra con una silla durante su recorrido, sería un error suponer que la silla siempre va a estar ubicada en ese lugar. Solo los obstáculos que tienen pocas chanceas de cambiar de sitio son reconocidos como obstáculos del ambiente, y se supone que todos ellos han sido marcados en un principio en el mapa, por ejemplo paredes, muebles de gran tamaño o grandes maquinarias.

Uso de Path Finding

Como ya hemos mencionado antes en este trabajo, hemos evaluado el comportamiento de Path Planning utilizando dos tipos distintos de algoritmos de Path Finding. Utilizamos un algoritmo no determinístico y uno determinístico, que son nuestro GA y la modificación de A^* respectivamente. Elegimos nuestro GA como representante de los algoritmos no determinísticos de Path Finding, por haber sido el algoritmo genético con el que obtuvimos mejores resultados en nuestro conjunto de mapas de prueba. Dado que todos los algoritmos determinísticos resuelven correctamente los mapas encontrando caminos óptimos en todas sus ejecuciones, optamos por A^* como su representante por ser el algoritmo más rápido.

Path Planning con nuestro GA Path Planning intenta obtener un camino con nuestro GA, recordemos que existe la posibilidad que nuestro GA no encuentre soluciones sin colisiones en un mapa en el que existen caminos que evitan colisiones. Supongamos que con nuestro GA obtenemos un camino

que no colisiona con los obstáculos conocidos del ambiente, supongamos también que durante la ejecución del camino nos encontramos con un obstáculo desconocido del ambiente, como ya dijimos antes, reconocemos ese obstáculo y planificamos un nuevo camino utilizando nuestro GA.
El algoritmo genético necesita utilizar todo el mapa para encontrar un nuevo camino, el camino obtenido une el punto de origen con el punto de destino deseado, el nuevo camino puede no pasar por el punto en el que se detuvo el movimiento de la plataforma, por lo tanto, para llegar al nuevo camino retrocedemos por el camino antiguo hasta alcanzar la primer intersección con el nuevo camino. Dicha intersección siempre existe, ya que ambos caminos al menos coincidirán en el punto de origen. Recordemos que la codificación del GA solo permite mapas de dimensiones cuadradas, y el resultado del mismo son caminos que van desde el punto de origen $(0, 0)$ al punto de destino $(n - 1, n - 1)$.

Algo para tener en cuenta es que Path Planning aprovecha que el resultado del algoritmo genético no es solo un camino, sino que es un conjunto de caminos. Utiliza este conjunto para buscar un nuevo camino, primero evalúa el fitness de cada uno de los caminos en la representación que contiene el nuevo obstáculo, si encuentra una o más soluciones que eviten colisiones se queda con la de mayor fitness. Si no encuentra una solución que evite colisionar las utiliza como población inicial para una nueva ejecución del algoritmo genético. Como estas soluciones tienen muchos tramos de camino buenos no es necesario evolucionarlas mucho para encontrar otros caminos que eviten (de ser posible) los obstáculos del ambiente.

En la Figura 5.3 se muestra una ejecución de Path Planning utilizando a nuestro GA como algoritmo de Path Finding. en la imagen se puede observar que Path Planning se encontró con dos obstáculos desconocidos del ambiente y que tuvo que volver a ejecutar el algoritmo genético. Además, en la figura, se puede observar la forma de unir el nuevo camino con el lugar donde se detuvo la ejecución con el nuevo camino, es claro que Path Planning volvió por sobre sus pasos hasta que llego a la primer intersección con el nuevo camino para empezar a ejecutarlo desde ese lugar.

Path Planning con A^* De existir varios caminos que evitan los obstáculos del ambiente, el algoritmo A^* encuentra uno que minimiza la cantidad de giros y la longitud del camino. Si durante el recorrido del camino la plataforma se encuentra con un obstáculo desconocido, lo reconoce y busca un camino entre el lugar donde se detuvo el robot y la meta deseada en el nuevo mapa. Este nuevo camino puede ser encontrado gracias a que A^* puede utilizar cualquier punto de inicio y fin del mapa.

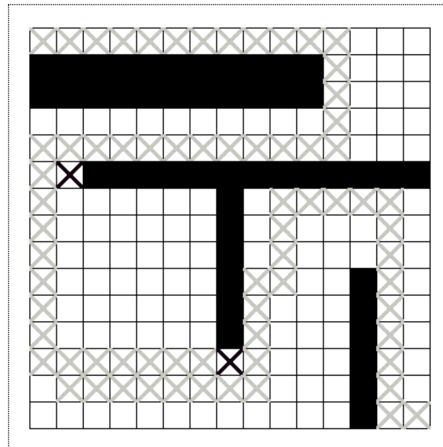


Figura 5.3: Ejemplo de una ejecución de Path Planning utilizando como algoritmo de Path Finding a nuestro GA.

En la Figura 5.4 se muestra una ejecución de Path Planning usando el algoritmo de Path Finding A^*, se puede observar que durante la ejecución el robot se encuentra con dos obstáculos desconocidos y resuelve correctamente el mapa. A diferencia del Path Planning con nuestro GA no se debe volver por el camino original hasta la intersección con el nuevo camino, porque como ya hemos dicho, el nuevo camino obtenido con A^* comienza en el lugar donde se detuvo la ejecución.

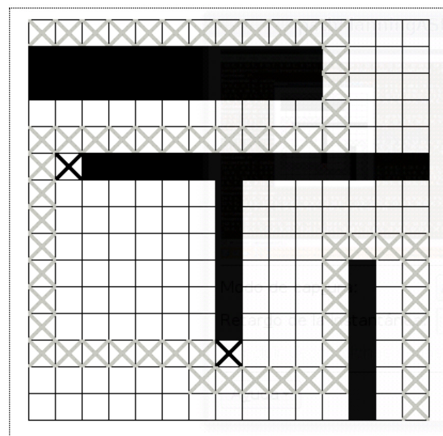


Figura 5.4: Ejemplo de una ejecución de Path Planning utilizando como algoritmo de Path Finding el algoritmo A^*.

Uso de Path Executing

Path Planning utiliza la información presente en la representación del ambiente para hacer uso de las sentencias de Path Executing. Como la operación con la que obtuvimos mejores resultados al momento de realizar un movimiento rectilíneo fue Wall Following, intentaremos utilizarla cada vez que sea posible. Recordemos que Wall Following, además, puede corregir la dirección del robot. Luego priorizaremos Wall Stop respecto a Basic Forward por el echo de corregir errores previos del desplazamiento y por haber obtenido mejores resultados en el movimiento en línea recta. Turn será usada cuando el robot necesite doblar. Esta última la descompondremos en dos acciones, Turn Rigth y Turn Left según se necesite doblar a la derecha o izquierda respectivamente.

Si bien describimos que sentencias de Path Executing se utilizaran, como estamos en etapa de simulación y no realizamos movimientos reales con la plataforma, supondremos que las sentencias son realizadas de forma correcta y el robot siempre termina en el lugar indicado de la representación.

Para recorrer el camino Path Planning hace una traducción del mismo a sentencias de Path Executing, por lo que transforma una sucesión de posiciones y direcciones en el mapa, a una sucesión de sentencias de Path Executing. Como algunas de las sentencias son parametrizadas por la distancia por la que deben ser ejecutadas, Path Planning utiliza la información del tamaño de grilla para asegurarse que siempre va a dejar ubicado el robot en el medio de una grilla luego de realizar un movimiento. Las únicas sentencias donde no importa el tamaño de la grilla son ambos giros, dado que ellas aseguran que luego de girar la plataforma se encontrara ubicada exactamente en el mismo lugar pero con el robot orientado a otra dirección.

Path Planning define nuevas sentencias para usar Wall Following de una forma más especifica. Primero define Wall Following a derecha o izquierda, y luego divide esto en dos nuevas categorías, en Wall Following cercano (near) o Wall Following lejano (far). Se considera Wall Following cercano cuando se puede realizar Wall Following usando una pared o un obstáculo que se encuentra paralelo al movimiento del robot y que esta ubicado en las celdas vecinas por donde transitará el robot. Wall Following lejano es considerado cuando el movimiento del robot se realiza paralelo a alguna pared que se encuentra a celda libre de por medio de la posición del robot, al costado del mismo, y no hay paredes paralelas al movimiento que se encuentren más cerca. Por lo tanto tenemos cuatro sentencias de Wall Following, que son: Near Rigth Wall Following, Near Left Wall Following, Far Left Wall Following, Far Rigth Wall Following.

Para traducir un camino, Path Planning intenta utilizar las primitivas de

Path Executing en el orden mostrado a continuación, hasta que se completa el recorrido del camino.

1. Si hay un giro en el camino lo traduce en las primitivas Turn Left o Turn Rigth según corresponda.
2. Si perpendicular a la dirección de ejecución del camino hay una pared que se encuentra celda libre de por medio de la posición actual del robot, Path Planning realiza lo siguiente:

    a) De poder ejecutarse alguno de los dos Near Wall Following, se realiza Wall Following hasta 2/3 del tamaño de la celda y en el tercio restante realiza Wall Stop terminando a una determinada distancia de la pared. La distancia a la que termina el robot de la pared, es el espacio que debe existir entre el frente del robot y el final de la celda para que el robot se encuentre centrado en la misma. Ver Figura 5.5 (a).

    b) De poder realizarse Far Wall Following, se hace durante 2/3 de la grilla y a partir de ese momento se realiza Wall Stop. Ver Figura 5.5 (b).

    c) De no poderse realizar ninguno de los anteriores se realiza Wall Stop. Ver Figura 5.5 (c).

3. De poder realizarse Near Wall Following, dicha acción es ejecutada.Ver Figura 5.5 (d).
4. Si las condiciones están dadas se ejecuta Far Wall Following. Ver Figura 5.5 (e).
5. Se ejecuta Basic Forward. Ver Figura 5.5 (f).

En la Figura 5.6 se muestra una ejecución de Path Planning. La grilla se obtuvo dividiendo el ambiente en 6 filas y 6 columnas, y cada celda mide 1 metro de lado. Las filas y las columnas se encuentran numeradas entre 0 y 5. La tripla (I, J, P) describe la ubicación y orientación del robot, por ejemplo, $(1, 2, S)$ corresponde a la ubicación del robot en la fila 1 y la columna 2, orientado al Sur. A la ejecución de Path Planning se le pidió que resuelva el mapa comenzando en la posición $(0, 0, E)$ y que termine en la posición $(5, 5, S)$. El camino mostrado en la figura es traducido a sentencias de Path Executing de la siguiente manera:

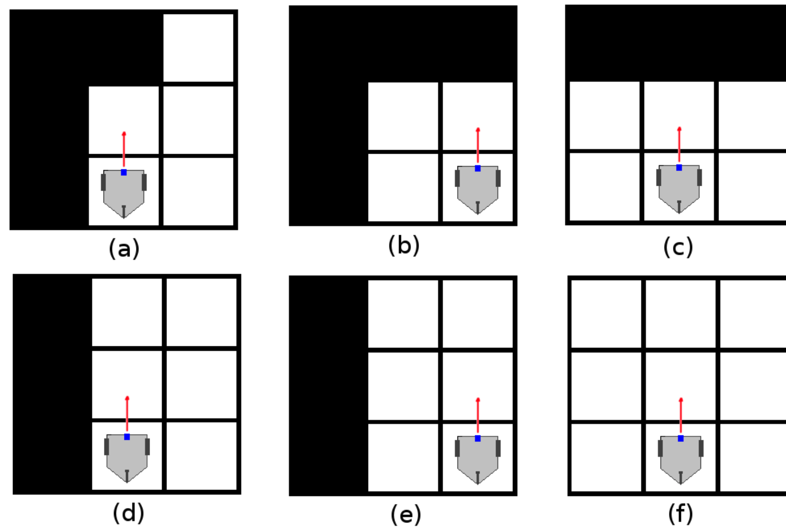


Figura 5.5: En la figura se muestra el orden de prioridad que utiliza Path Planning para seleccionar las primitivas al ejecutar un camino. (a) Near Wall Following, luego Wall Stop. (b) Far Wall Following, luego Wall Stop. (c) Wall Stop. (d) Near Wall Following. (e) Far Wall Following. (f) Basic Forward.

Posición Inicial	Posición Final	Sentencia de Path Executing
(0,0,E)	(0,0,S)	Turn Rigth
(0,0,S)	(1,0,S)	Wall Following (Near): 1 metro
(1,0,S)	(2,0,S)	Basic Forward: 1m
(2,0,S)	(2,0,E)	Turn Left
(2,0,E)	(2,1,E)	Wall Following (Near): 1 metro
(2,1,E)	(2,2,E)	Wall Following (Far): 0,66 metros + Wall Stop 0,33 metros
(2,2,E)	(2,2,N)	Turn Left
(2,2,N)	(1,2,N)	Wall Following (Near): 0,66 metros + Wall Stop 0,33 metros
(1,2,N)	(1,2,E)	Turn Rigth
(1,2,E)	(1,3,E)	Wall Following (Near): 1 metro
(1,3,E)	(1,4,E)	Wall Following (Near): 0,66 metros + Wall Stop 0,33 metros
(1,4,E)	(1,4,S)	Turn Rigth
(1,4,S)	(4,4,S)	Wall Following (Near): 3 metros
(4,4,S)	(5,4,S)	Basic Forward: 1m
(5,4,S)	(5,4,E)	Turn Left
(5,4,E)	(5,5,E)	Wall Following (Far): 1 metro
(5,5,E)	(5,4,S)	Turn rigth

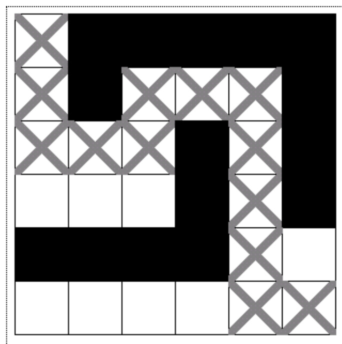


Figura 5.6: Ejecución de Path Planning con A^*.

5.3. Ejecución de Path Planning

La ejecución consiste en realizar el Path Planning en el ambiente real, utilizando la plataforma para realizar los movimientos a través de las sentencias de Path Executing, y usando sensores para detectar obstáculos desconocidos en la representación.

Si bien en el trabajo recorremos el camino obtenido por Path Finding con la plataforma utilizando las primitivas de Path Executing, no detectamos obstáculos que se encuentran ausentes en la representación del ambiente. Por lo tanto, nuestra ejecución del Path Planning se resume a encontrar un camino y recorrerlo. No se evitan obstáculos desconocidos en la representación porque la plataforma no posee aún dispositivos que sean suficientes para detectar obstáculos. En consecuencia, para que Path Planning funcione correctamente en cualquier ambiente, necesitamos que todos los obstáculos sean descriptos en el mapa. Una vez que dispongamos en la plataforma elementos suficientes como para determinar que en frente del robot hay un obstáculo, por los resultados obtenidos a través de simulación, extender lo realizado a una versión de Path Planning que evite los obstáculos desconocidos del ambiente no supone una gran complejidad. Por los problemas de la precisión del sónar mostrados en la Sección 4.2, notamos que este dispositivo no sirve para la detección de obstáculos no reconocidos en la representación.

Los experimentos realizados con la plataforma nos muestran que las operaciones de Path Executing corrigen muchos de los errores que surgen de la ejecución total del camino. Aunque la corrección de errores es buena, no puede evitar que los errores existan, sobre todo en habitaciones en las que hay pocos elementos para reubicarse. Ambientes con pocas paredes perjudican notablemente la corrección porque no nos permite ejecutar Wall Following

y Wall Stop que son las sentencias de Path Executing útiles para corregir errores.

5.4. Resultados

En la simulación del Path Planning, la experiencia de utilizar dos tipos diferentes de Path Finding arrojo buenos resultados, tanto con el algoritmo genético como con la modificación de A^*. En el primer algoritmo nos encontramos con la grata sorpresa de que algunos de los caminos encontrados no eran soluciones óptimas, pero evitaban obstáculos desconocidos del ambiente que no eran evitados por algunas de las soluciones óptimas. Estas soluciones a pesar de no ser óptimas se encontraban muy cerca de serlo. En las figuras 5.7 y 5.8 se muestran ejemplo de que el algoritmo genético puede encontrar soluciones que no necesariamente son óptimas, pero evitan colisionar con obstáculos desconocidos por la representación.

Recordemos que si durante la ejecución de un camino se encuentra un obstáculo, un nuevo camino debe ser buscado. Aquí aparece otra de las situaciones favorables del algoritmo genético. Como el resultado de un GA es una población (un conjunto de caminos) y no solo un camino, la población puede ser utilizada en la busqueda de un nuevo camino. Si no existe un camino en la población que resuelva el nuevo mapa evitando colisionar, se utiliza dicha población como población inicial para una nueva ejecución del algoritmo genético por tener partes de caminos que son buenos. En consecuencia el GA no debe evolucionar demasiado para encontrar una nueva solución. Esto provoca que el tiempo de ejecución del algoritmo genético disminuya notablemente.

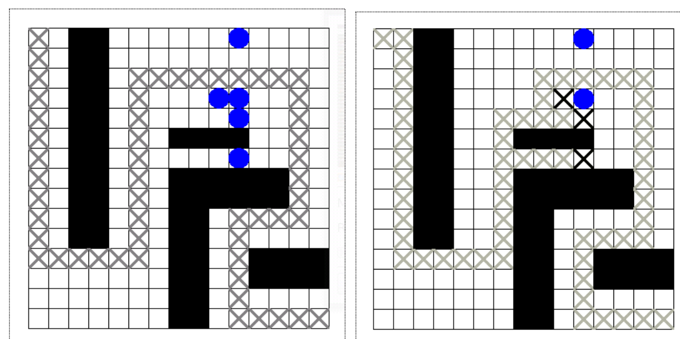


Figura 5.7: La primer imagen corresponde a una ejecución de Path Planning utilizando nuestro GA y la segunda usando A^*.

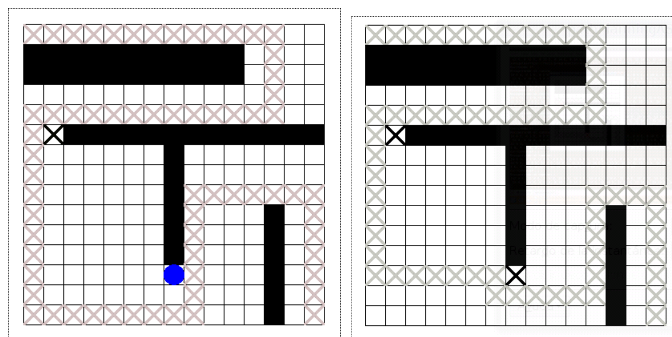


Figura 5.8: La primer imagen es una ejecución de Path Planning utilizando nuestro GA y la segunda corresponde a una ejecución de Path Planning con A*.

A pesar de todas estas condiciones favorables que tiene el algoritmo genético, no son suficientes para mejorar lo realizado por A^* el cual es mucho más rápido, y lo más importante, si existe una solución posible para el mapa, A^* la encuentra, algo que no es asegurado por el algoritmo genético. Si bien los algoritmos genéticos pueden encontrar soluciones que evitan obstáculos desconocidos para la representación, a veces pueden generar caminos no óptimos que se encuentren con estos obstáculos a pesar de que existen caminos óptimos que no lo hacen.

El resultado del uso de Path Executing es bueno, dado que se aprovecha la información presente en cada mapa utilizando la operación de Path Executing que utilice mejor la información disponible en el ambiente en cada momento. A pesar de que se observa a través de experimentos que las operaciones corrigen muchos de los errores de desplazamiento, no todos los errores pueden ser corregidos, pudiendo terminar en una ejecución incorrecta de Path Planning. Mucho se debe a que las operaciones dependen de los obstáculos presentes del ambiente para realizar las correcciones, curiosamente ambientes con pocos obstáculos complican la ejecución de Path Planning por tener pocas referencias. A pesar de esto, los resultados nos demuestran un gran acoplamiento entre Path Finding, Path Executing y Path Planning, aprovechando la información presente tanto en la representación como el ambiente.

Capítulo 6

Conclusión

En el trabajo realizado se presentó un sistema de navegación de interiores para plataformas autonomas móviles. El sistema fue dividido en tres partes: Path Finding, Path Planning y Path Executing.

Path Finding se encarga de encontrar un camino libre de colisiones que una dos puntos del ambiente, además el camino debe ser mínimo en giros y en longitud.

Path Executing provee al sistema un conjunto de operaciones útiles para realizar movimientos con la plataforma. Algunas de las operaciones utilizan la información presente del ambiente para realizar correcciones de errores acumulados en movimientos anteriores.

Path Planning se encarga de recorrer un camino en un determinado ambiente, evitando colisionar con los obstáculos presentes en el mismo.

Se presentaron posibles soluciones para los siguientes problemas:

Camino Óptimo: Encontrar un camino libre de colisiones que sea mínimo en giros y longitud entre dos puntos del ambiente.

Manejo de Hardware: El hardware de la plataforma presenta ciertas imperfecciones. El uso de estos elementos puede derivar en errores durante el desplazamiento de la plataforma en el ambiente.

Ambientes Variables: El ambiente no se mantiene estático, es decir que pueden aparecer obstáculos desconocidos para la plataforma. Por lo tanto mientras se desplace la plataforma también se debe evitar colisionar con estos obstáculos.

Cada uno de estos problemas precisó distintas soluciones, las cuales se detallan a continuación:

Camino Óptimo: Path Finding soluciono el problema de encontrar un camino que minimiza la distancia y los giros que la plataforma debe efectuar para llegar al destino. Para realizar esto se propusieron algoritmos de diversos tipos que solucionan el problema de Path Finding, que pueden ser clasificados en determinísticos y no determinísticos. El grupo de los algoritmos de Path Finding no determinísticos contiene dos Algoritmos Genéticos (GA). El primer algoritmo es una réplica del propuesto en el paper [5, kamran H. Sedighi y otros]. El segundo es nuestra propuesta de algoritmo genético para Path Finding. El grupo de los algoritmos de Path Finding determinísticos se encuentra compuesto por un algoritmo dinámico, una modificación de Dijkstra y una modificación del algoritmo A^*. Tanto los algoritmos no determinísticos como los tres algoritmos determinísticos solucionaron el problema de Path Finding.

Manejo de Hardware: Como el sistema fue probado en un robot real, para evitar que el uso del hardware introdujera errores, se crearon operaciones utilizadas por el sistema de navegación y que hacen uso del hardware. Las siguientes primitivas fueron diseñadas para corregir errores de hardware.

Basic Forward: propuso una solución a los errores generados por la diferencia de potencia de los motores para lograr realizar un movimiento rectilíneo.

Wall Stop: es una operación que realiza un movimiento rectilíneo hacia adelante, deteniendo al robot a cierta distancia de una pared que se encuentra ubicada perpendicular al movimiento del robot, para esto utiliza el sensor de ultrasonido. Esta operación logró corregir los errores generados por los odómetros.

Wall Following: también hace uso del sensor de ultrasonido, para mantener cierta distancia con un obstáculo que se encuentra paralelo a la dirección de movimiento del robot. La operación solucionó los problemas generados por el uso de los odómetros de la plataforma para realizar giros.

Turn: no resuelve ninguno de los problemas de los dispositivos presentes en la plataforma, solo garantiza que luego de ejecutarla el robot se encuentra ubicado en el mismo sitio pero girado 90 grados.

Ambientes Variables: Path Planning combina los caminos encontrados por Path Finding y el uso de las primitivas de Path Executing a fin de

solucionar el problema de encontrar un camino libre de colisiones en el ambiente y ejecutarlo evitando colisionar con los obstáculos desconocidos del mismo. Path Planning fue realizado en dos etapas, simulación y ejecución. En la primera se realizo Path Planning simulando el ambiente y la ejecución del mismo. En la otra etapa se realizo en el ambiente real, utilizando la plataforma para la ejecución. En la etapa de simulación se utilizaron nuestro GA y la modificación de A^* como algoritmos de Path Finding. En la etapa de ejecución se opto por A^* para la busqueda de caminos. En esta etapa, a diferencia con la simulación, no se resolvio el problema de detectar obstáculos desconocidos del ambiente por no tener dispositivos en la plataforma capaces de realizarlo.

Para evaluar los caminos conseguidos con los algoritmos de Path Finding, generamos manualmente una colección de 20 mapas.

En el trabajo mostramos que no siempre se obtuvieron caminos sin colisiones con la réplica del algoritmo propuesto en [5]. La experimentación realizada con la colección de mapas nos mostró que el índice de caminos exitosos que obtuvimos con nuestra réplica es mayor al índice expuesto por el paper.

Utilizamos nuevamente la colección de mapas para comparar nuestro GA con nuestra réplica del GA propuesto en el paper. La tasa éxitos con nuestro GA fue superior a la obtenida por la réplica realizada del algoritmo en todos los mapas del conjunto. En uno de los mapas del conjunto la diferencia radica en que nuestra propuesta de algoritmo genético logró solucionarlo exitosamente en algunas de las ejecuciones, mientras que la réplica nunca logró resolverlo evitando colisionar con los obstáculos del ambiente. La diferencia a favor de nuestro GA no solo es visible en el porcentaje de ejecuciones resueltas de cada mapa, sino que también se observa en la cantidad de soluciones óptimas de ambos algoritmos, nuestro algoritmo genético encuentra mas soluciones óptimas en 10 mapas del conjunto de pruebas, mientras que la réplica encuentra mas soluciones óptimas que nuestro algoritmo genético solo en 6 mapas.

Con estos resultados pudimos ver que nuestras modificaciones en la representación del GA del [5] lograron darle más expresividad a los caminos pudiendo así solucionar una variedad de mapas más grande. Además, como la cantidad de variables de nuestro GA es menor, resolvió los mapas precisando menos tiempo de ejecución.

Los algoritmos de Path Finding determinísticos se compararon ejecutando el set de 20 mapas. Los tres algoritmos (El algoritmo Dinámico, Dijkstra y A^*) obtuvieron caminos óptimos en todos los mapas. Solo encontramos diferencias en los tiempos de ejecución. De las pruebas obtuvimos que tanto

la modificación de Dijkstra como la modificación de A^* superan ampliamente a nuestro algoritmo dinámico. Como era de esperar, la modificación de A^* obtuvo mejores tiempos de ejecución que la modificación de Dijkstra.

Los resultados de Path Executing se obtienen analizando cada primitiva por separado. Según los experimentos realizados sobre Omega, Basic Forward tiene la ventaja de introducir menos errores que si se usaría directamente el hardware de los motores. Como desventaja tenemos que Basic Forward utiliza los odómetros para corregir errores, los cuales a su vez poseen errores de medición. Esto provoca que Basic Forward pueda generar un desfasaje con respecto a la representación interna de la ubicación. Por otro lado, esta directiva no es capaz de eliminar errores acumulados en los movimientos previos dado que no se vale de elementos del ambiente para determinar la ubicación final del robot.

Según los resultados obtenidos mediante experimentación, Wall Stop nos asegura que al finalizar su ejecución el robot se encontrará a la distancia esperada a la pared. A diferencia de Basic Forward esta acción puede corregir errores acumulados en la representación interna de la ubicación, dado que utiliza objetos del medio para establecer la ubicación final del robot. La desventaja de este movimiento es que el uso de objetos del ambiente limita al sistema a utilizar Wall Stop solo en el caso que este requiere aproximarse a una pared u objeto. Además esta acción no es capaz de corregir errores en la orientación del robot.

Como ventaja, Wall Following demostró mediante experimentación poder corregir errores en la ubicación del robot. Además si esta acción es usada en forma correcta, permite volver a orientar correctamente el robot, eliminando así errores de orientación. Wall Following limita al sistema a utilizar este movimiento solo para desplazamientos paralelos a paredes u objetos. Al intentar mantener el robot a una distancia dada a la pared puede causar un zigzagueo recorriendo menos distancia que la marcada por los odómetros, y generando un desfasaje en la representación de la ubicación.

A través de experimentos pudimos determinar que Turn es lo suficientemente buena como para suponer que despues de girar el robot se encuentra aproximadamente en la misma ubicación. En contraste, la primitiva puede agregar errores asociados al uso de los odómetros. Además, los problemas generados por la discrepancia de potencias en los motores pueden desviar las trayectorias estimadas al retroceder y avanzar para completar el giro.

En la simulación del Path Planning, la experiencia de usar nuestro GA y A^* como algoritmos de Path Finding arrojo buenos resultados. En el primer algoritmo nos encontramos con la grata sorpresa de que algunos de los caminos encontrados no eran soluciones óptimas, pero evitaban obstáculos desconocidos del ambiente que no eran evitados por algunas de las solucio-

nes óptimas. Estas soluciones a pesar de no ser óptimas se encontraban muy cerca de serlo. Como Path Planning también evita los obstáculos desconocidos en la representación del ambiente, si durante la ejecución de un camino se encuentra un obstáculo, un nuevo camino debe ser buscado. Como el resultado de un GA es una población (un conjunto de caminos) y no solo un camino, la población puede ser utilizada en la busqueda de un nuevo camino, y esto provoca que el tiempo de ejecución del algoritmo genético disminuya notablemente.

A pesar de todas estas condiciones favorables que tiene el algoritmo genético, las mismas no son suficientes para mejorar lo realizado por A^*, el cual es mucho mas rápido, y lo más importante, si existe una solución posible para el mapa, A^* la encuentra, algo que no es asegurado por el algoritmo genético.

El resultado del uso de Path Executing fue bueno, dado que en las pruebas realizadas aprovechó la información presente en cada mapa utilizando la operación de Path Executing que utilice mejor la información disponible en el ambiente en cada momento. A pesar de que se observó que las operaciones corrigieron muchos de los errores de desplazamiento, no todos los errores pudieron ser corregidos, terminando a veces en una ejecución incorrecta de Path Planning. Mucho se debió a que las operaciones dependen de los obstáculos presentes del ambiente para realizar las correcciones, curiosamente ambientes con pocos obstáculos complican la ejecución de Path Planning por tener pocas referencias. Sin embargo los resultados nos demuestran un gran acoplamiento entre Path Finding, Path Executing y Path Planning, aprovechando la información presente tanto en la representación como el ambiente.

Bibliografía

[1] Lance Chambers. Practical handbook of genetic algorithms.

[2] David E. Goldberg. Genetic algorithms in search, optimization & machine learning, 1998.

[3] Sean Luke, Liviu Panait, Gabriel Balan, Sean Paus, Zbigniew Skolicki, Joseph Harrison, Jeff Bassett, Robert Hubley, and Alexander Chircop. An evolutionary computation and genetic programming system.

[4] Tom M. Mitchell. Machine learning.

[5] Kamran H. Sedighi, Kaveh Ashenayi, Theodore W. Manikas, Roger L. Wainwright, and Heng ming Tai. Autonomous local path planning for a mobile robot using a genetic algorithm.