Proyecto Fin de Grado
Ingeniería Electrónica, Robótica y Mecatrónica

Métodos de Generación de Trayectorias

Autor: Florentino Prieto Rodríguez
Tutor: Begoña Chiquinquira Arrue Ulles

Dep. Ingeniería de Sistemas y Automática
Escuela Técnica Superior de Ingeniería
Universidad de Sevilla

Sevilla, 2017

Proyecto Fin de Grado
Ingeniería Robótica, Electrónica y Mecatrónica

Métodos de Generación de Trayectorias

Autor:
Florentino Prieto Rodríguez

Tutor:
Begoña Chiquinquira Arrue Ulles
Profesor titular

Dep. de Ingeniería de Sistemas y Automática
Escuela Técnica Superior de Ingeniería
Universidad de Sevilla
Sevilla, 2017

Proyecto Fin de Grado: Métodos de Generación de Trayectorias

Autor: Florentino Prieto Rodríguez

Tutor: Begoña Chiquinquira Arrue Ulles

El tribunal nombrado para juzgar el Proyecto arriba indicado, compuesto por los siguientes miembros:

Presidente:

Vocales:

Secretario:

Acuerdan otorgarle la calificación de:

Sevilla, 2017

El Secretario del Tribunal

A mi familia

A mis maestros

Agradecimientos

Termino un ciclo muy importante en mi vida. Ha sido un periodo nada fácil, ha habido momentos complicados en los que uno no sabía muy bien si podría avanzar o no. Por estos momentos le agradezco a mi familia y amigos por todo su apoyo y ayuda. Gracias a ellos pude avanzar sin miedo y esforzándome.

Aunque haya tenido problemas la verdad que jamás olvidaré estos años, pues han sido los mejores, una experiencia inolvidable y parte de esto es de algunos profesores los cuáles nunca dudaron en echar una mano siempre que podían, que siempre estaban dispuesto a explicar cualquier duda fueran o no de sus asignaturas, algunos incluso te trataban como alguien cercano y eso se agradece mucho, puesto que hacen de este trayecto algo más fácil.

También debo de agradecer a mi tutora del TFG Begoña, la cuál ha sido paciente conmigo y me ha ofrecido toda la ayuda posible y se ha preocupado para que pudiera hacer este trabajo sin problemas.

Por todo esto muchas gracias a todos por estar ahí para mí.

Florentino Prieto Rodríguez

Sevilla, 2017

Resumen

Este proyecto tiene la finalidad de aprender diferentes formas de calcular una ruta hacia un objetivo desde un punto inicial. Podremos verlos en diferentes casos y compararlos entre ellos para determinar cuál es el mejor de todos y para que condiciones.

El objetivo de este trabajo también es el de informar de algoritmos diferentes para su conocimiento y su comprensión.

El trabajo está dividido en dos grandes partes; la primera se encarga de mostrar uno a uno cada algoritmo y su funcionamiento en diferentes casos y la segunda compara estos métodos para verlos en un mismo caso e intentar comprender cuál puede ser mejor en cada caso.

Se explicarán los códigos de cada uno para saber como funcionan.

Abstract

This proyect has the finality of learning differents forms to calculate a route to an objetive with a start point. We can see differents cases and comparate between them for determinate which es the best of all in differents conditions.

The objective of this work is also inform about differents algorithms for your knowledge and your comprenssion.

This work is divided into two major parts; the first is responsable for showing each algorithm one by one and its operation in different cases and the second compares these methods to see them in the same case and try to understand wich may be better en each case.

They will explain the codes of each to know how they work.

Índice

Agradecimientos **ix**

Resumen **xi**

Abstract **xiii**

Índice **xv**

Índice de Tablas **xvii**

Índice de Figuras **xviii**

Notación **xx**

1 Introducción **21**
1.1 Objetivos y motivaciones del Proyecto *21*
1.2 Estructura del Trabajo *21*

2 Métodos de Generación de Trayectoria **2**
2.1 A Estrella *2*
2.1.1 Descripción 2
2.1.2 Características Principales 2
2.1.3 Funcionamiento A* 3
2.1.4 Simulaciones 4
2.3. Descomposición en Celdas *6*
2.2.1 Descripción 6
2.2.2 Funcionamiento 6
2.2.3 Simulaciones 9
*2.4. RRT y RRT** *11*
2.3.1 Rapidly-exploring Random Tree (RTT) 11
2.3.2 Rapidly-exploring Random Tree Star (RRT*) 12
2.3.3 Simulaciones 13
2.3.4 Comparación entre RRT y RRT* 15
2.5. Voronoi y Dijkstra *20*
2.4.1 Diagrama de Voronói 20
2.4.2 Algoritmo Dijkstra 28
2.4.3 Simulaciones 31

3 Comparación entre Métodos **33**
3.1 Simulaciones *33*
3.2 Comparaciones *41*

4 Conclusiones y Desarrollos Futuros **43**
4.1 Conclusiones *43*
4.2 Desarrollos Futuros *43*

5 Bibliografía **44**

ÍNDICE DE TABLAS

Tabla 1 Casos de las Simulaciones de A* 4
Tabla 2 Casos de simulaciones de Descomposición en Celdas 9
Tabla 3 Casos de RRT 13
Tabla 4 Tabla de tiempos para Caso 1 41
Tabla 5 Tabla de tiempos para Caso 2 41
Tabla 6 Tabla de tiempos para ambos caso en RRT 42

ÍNDICE DE FIGURAS

Figura 2-1 Nodo Padre y Nodos Sucesivos 4
Figura 2-2 Simulación de A* 5
Figura 2-3 Caso de de ruta no posible 5
Figura 2-4 Simulación de pasillo 6
Figura 2-2-5 Ejemplo de Descomposción de celdas trapezoidal 7
Figura 2-2-6 Grafo de conectividad de la figura 2-1 8
Figura 2-7 Ejemplo de Creación de Celdas 8
Figura 2-8 Grafo de conectividad 9
Figura 2-9 Simulación 1 de Descomposición en Celdas 10
Figura 2-10 Simulación 2 de Descomposición en Celdas 10
Figura 2-11 Simulación 3 de Descomposición en Celdas 11
Figura 2-12 Simulación de RRT 13
Figura 2-13 Simulación 2 de RRT 14
Figura 2-14 Simulación pasillo 14
Figura 2-2-15 Estado inicial de ambos algoritmos. El nodo inicial empieza en (0,0). 15
Figura 2-2-16 Vemos que para 10 nodos no existen diferencias. 16
Figura 2-2-17 Para 100 nodos siguen siendo similares. 16
Figura 2-2-18 A los 500 nodos empiezan a haber diferencias. 17
Figura 2-2-19 Con 2000 nodos la diferencia se nota bastante. Se observa como el RRT es más caótico que el RRT*. 17
Figura 2-2-20 Con 4000 nodos la diferencia es bastante notable. 18
Figura 2-2-21 Izquierda RRT, derecha RRT* 19
Figura 2-2-22 (Imágenes anteriores ampliadas) Arriba RRT, abajo RRT* 19
Figura 2-2-23 Los cuatros sitios más externos tiene asociados regiones de Voronói abiertas caracterizadas por las semi-aristas 20
Figura 2-2-24 Divide y vencerás 1 23
Figura 2-2-25 Divide y vencerás 2 23
Figura 2-2-26 Divide y vencerás 3 24

Figura 2-2-27 Divide y vencerás 4 24
Figura 2-2-28 Divide y vencerás 5 25
Figura 2-2-29 Fortune (barrido recta) 26
Figura 2-2-30 Fortune Progreso 1 26
Figura 2-2-31 Fortune Progreso 2 27
Figura 2-2-32 Fortune Progreso 3 27
Figura 2-2-33 Fortune Progreso 4 27
Figura 2-2-34 Fortune Progreso 5 28
Figura 2-2-35 Fortune Progreso 6 28
Figura 2-2-36 Ejecución del algoritmo de Dijkstra 30
Figura 2-37 Simulación 1 Voronoi 31
Figura 2-38 Simulación 2 Voronoi 32
Figura 2-39 Simulación 3 Voronoi 32
Figura 3-1 Simulación A* 33
Figura 3-2 Simulación Descomposición en Celdas 34
Figura 3-3 Simulación RRT (distancia 5) 35
Figura 3-4 Simulación RRT (distancia 1) 35
Figura 3-5 Simulación Voronoi (Simulación 1) 36
Figura 3-6 Simulación Voronoi (Simulación 2) 36
Figura 3-7 Simulación A* v2 37
Figura 3-8 Simulación de Descomposición en Celdas v2 38
Figura 3-9 Simulación RRT (distancia 5) v2 39
Figura 3-10 Simulación RRT (distancia 1) v2 39
Figura 3-11 Simulación Voronoi (Simulación 1) v2 40
Figura 3-12 Simulación Voronoi (Simulación 2) v2 40

Notación

RRT	Rapidly-exploring Random Tree
DC	Descomposición en Celdas
e.o.c.	En cualquier otro caso
e	número e
sen	Función seno
tg	Función tangente
arctg	Función arco tangente
sen	Función seno
$\sin^x y$	Función seno de x elevado a y
$\cos^x y$	Función coseno de x elevado a y
rect	Función rectángulo
Sinc	Función sinc
Pr(A)	Probabilidad del suceso A
SNR	Signal-to-noise ratio
$\leq$	Menor o igual
$\geq$	Mayor o igual
$<$	Menor
$>$	Mayor

1 INTRODUCCIÓN

EL avance científico en el campo de la robótica y automática busca dotar de la mayor autonomía posible a los robots y autómatas programables. Su objetivo principal es el de mejorar la calidad de vida de las personas liberando a las personas de funciones, tanto aspiradoras autónomas por ejemplo, hasta transporte de productos.

Este trabajo tiene el objetivo de explicar diferentes algoritmos de búsqueda y métodos de generación de trayectorias. Estos algoritmos serán los llamados A*, descomposición en celdas, RRT y RRT* y por último Voronoi junto con Dijkstra.

1.1 Objetivos y motivaciones del Proyecto

A la hora de hacer que un robot móvil vaya de un punto A a un punto B teniendo que sortear diferentes obstáculos, se puede hacer de diferentes formas que dependen de las limitaciones que se tienen, tanto técnicas como económicas.

La idea principal es estudiar los diferentes algoritmos de búsqueda para poder entenderlos, conocerlos bien a fondo, ver como funcionan en diferentes ámbitos y verlos entre ellos para poder ver las diferencias que existen y así poder elegir uno para las distintas especificaciones que se quieran.

Como idea secundaría esta el de conocer diversos métodos de poder calcular la trayectoria tanto el método de funcionamiento como algo de información sobre ellos.

El trabájo se ha desarrollo mediante el software de Matlab para mostrar distintas simulaciones de los algoritmos y poder entender su funcionamiento mejor. Este tipo de programa es muy útil en estos casos más teóricos donde se pueden ver como podría funcionar nuestro algoritmo antes de hacer una gran inversión en un proyecto sin saber de antemano su funcionamiento.

1.2 Estructura del Trabajo

El desarrollo del proyecto puede ser explicado en las siguientes partes:

- Métodos estudiados: Donde se estudiarán por separado cada uno de los casos elegidos para su estudio en los que se verán tanto una explicación teórica de su funcionamiento, como distintas simulaciones de su funcionamiento en distintos entornos.
- Comparación de los algoritmos: En este apartado juntaremos los anteriores métodos nombrados para hacerlos funcionar en un mismo entorno de trabajo para su estudio. Con los datos y resultados obtenidos se mostrarán las conclusiones llegadas.
- Conclusiones del trabajo: En esta parte se resumen las impresiones generadas en el trabajo, las conclusiones obtenidas y las posibles mejoras del proyecto.

2 MÉTODOS DE GENERACIÓN DE TRAYECTORIA

Existen diferentes formas de conseguir una trayectoria de un punto inicial a uno final a través de diferentes obstáculos. En este apartado veremos unos pocos métodos para conocerlos y saber como funcionan y ver las diferentes formas en las que se obtienen las trayectorias.

2.1 A Estrella

Algoritmo de Búsqueda A*. También conocido como A asterisco o A estrella fue presentado por Peter E. Hart, Nils, J. Nilsson y Bertram Raphael en el año 1968, se clasifica dentro de los algoritmos de búsqueda en grafos. Su función es encontrar siempre y cuando se cumplan determinadas condiciones, el camino de menor costo entre un nodo origen y uno objetivo, es la forma más ampliamente conocida de la búsqueda primero el mejor, siendo la búsqueda A* tanto completa como óptima.

2.1.1 Descripción

Antes de explicar el algoritmo de A* conviene explicar un poco que es un algoritmo voraz. Así pues, un algoritmo voraz 8también conocido como ávido, devorador o goloso) es aquel que, para resolver un determinado problema, sigue una heurística consistente en elegir la opción óptima en cada paso local con la esperanza de llegar a una solución general óptima. Este esquema algorítmico es el que menos dificultades plantea a la hora de diseñar y comprobar su funcionamiento. Normalmente se aplica a los problemas de optimización.

El problema de algunos algoritmos de búsqueda, como puede ser el agoritmo voraz, es que se guían en exclusiva por la función heurística, la cual puede no indicar el camino de coste más bajo, o por el coste real de desplazarse de un nodo a otro (como los algoritmos de escalada), pudiéndose dar el caso de que sea necesario realizar un movimiento de coste mayor para alcanzar la solución. Es por ello bastante intuitivo el hecho de que un buen algoritmo de búsqueda informada debería tener en cuenta ambos factores, el valor heurístico de los nodos y el coste 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 desde el actual, n, hasta el final, y g(n), el coste real del camino recorrido para llegar a dicho nodo, n, desde el nodo inicial. A* mantiene dos estructuras de datos auxiliares, que podemos denominar abiertos, implementado como una cola de prioridad (ordenada por el valor f(n) de todos sus hijos, los inserta en abiertos, y pasa el nodo evaluado a cerrados.

El algoritmo de una combinación entre búsquedas del tipo primero en anchura con primero en profundidad: mientras que h'(n) tiende a primero en profundidad, g(n) tiende a primero en anchura. De este modo, se cambia de camino de búsqueda cada vez que existen nodos más prometedores.

2.1.2 Características Principales

Como todo algoritmo de búsqueda en anchura, A* es un algoritmo completo: en caso de existir una solución, siempre dará con ella.

Si para todo nodo n del grafo se cumple g(n) = 0, nos encontramos ante una búsqueda voraz. Si para todo nodo del grafo se cumple h(n) = 0, A* pasa a ser una búsqueda de coste uniforme no informada.

Para garantizar la optimalidad del algoritmo, la función h(n) debe ser admisible, o sea que no sobrestime el coste real de alcanzar el nodo objetivo.

De no cumplirse dicha condición, el algoritmo pasa a denominarse simplemente A, y a pesar de seguir siendo completo, no se asegura que el resultado obtenido sea el camino de coste mínimo. Asimismo, si garantizamos que h(n) es consistente (o monótona), es decir, que para cualquier nodo n y cualquiera de sus sucesores, el coste estimado de alcanzar el objetivo desde n no es mayor que el de alcanzar el sucesor más el coste de

alcanzar el objetivo desde el sucesor.

La complejidad computacional está relacionada con la calidad de la heurística que se utilice en el problema. En el caso peor, con una heurística de pésima calidad, la complejidad será exponencial, mientras que en el caso mejor, con una buena h'(n), el algoritmo se ejecutará en tiempo lineal. Para que esto último suceda, se debe cumplir que:

$$h'(x) \leq g(y) - g(x) + h'(y)$$

Donde h' es una heurística óptima para el problema, como por ejemplo, el coste real de alcanzar el objetivo.

El espacio requerido por A* para ser ejecutado es su mayor problema. Dado que tiene que almacenar todos los posibles siguientes nodos de cada estado, la cantidad de memoria que requerirá será exponencial con respecto al tamaño del problema. Para solucionar este problema, se han propuesto diversas variaciones de este algoritmo, como pueden ser RTA*, IDA* o SMA*.

El rendimiento de los algoritmos de búsquedas heurísticas depende de la calidad de la función heurística.

2.1.3 Funcionamiento A*

Este algoritmo utiliza una función de evaluación f(n) = g(n) + h'(n), donde h'(n) representa el valor heurístico del nodo a evaluar desde el actual, n, hasta el final, y g(n), el costo real del camino recorrido para llegar a dicho nodo, n. 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 del tipo primero en anchura con primero en profundidad: mientras que h'(n) tiende a primero en profundidad, g(n) tiende a primero en anchura. De este modo, se cambia de camino de búsqueda cada vez que existen nodos más prometedores.

1 Función heurística de A*

f(n) = g(n) + h(n): Coste real del plan (camino) de mínimo coste que pasa por n.

f*(n) = g(n) + h*(n): Estimación de f.

2 Estrategia de A*

Entre las hojas del árbol de búsqueda, elegir el nodo de valor f* mínimo.

3 Interpretación fuerte de A*

Una heurística suele facilitar la resolución de un problema, pero no garantiza que se resuelva.

Una heurística es una regla de tres para un problema.

Busqueda: Optimalidad o incluso completitud no garantizados.

4 Esquematización de A*

Se basa en la búsqueda general.

Almacenar el valor g de cada nodo expandido.

Mantener la estructura abierta ordenada por valores crecientes de f*.

Insertar nuevos nodos en la estructura abierta según sus valores de f*.

En el código empleado para este algoritmo hay una matriz de nombre MAP donde se encuentran los puntos "OPEN" (aquellos por donde el vehículo puede pasar) y "CLOSED" (aquellos por donde no puede pasar). El tamaño de MAP es del tamaño del mapa donde se realiza la simulación.

Primero, antes de empezar el algoritmo, se colocan en la lista de puntos "CLOSED" los obstáculos de la zona y en la de puntos "OPEN" el punto de inicio del vehículo y calcula el coste de la función f(n) entre la posición inical y final.

Quita de la lista "OPEN" el nodo con el coste más pequeño y se pone en la lista "CLOSED", este es el nodo **n**

(si dos nodos tienen exactamente el mismo coste se coge el último nodo en mirar de los dos).

Si **n** es el nodo final entonces se usa el algoritmo para obtener la trayectoria, si no, continúa.

Determina todos los nodos siguientes a **n** y calcula el coste de cada uno que no esté en la lista "CLOSED".

Asocia con cada nodo sucesivo que no esté en la lista "OPEN" o "CLOSED" el coste calculado y se pone en la lista de "OPEN", colocando puntero en **n** (**n** es el nodo padre).

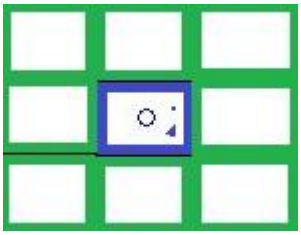

Figura 2-1 Nodo Padre y Nodos Sucesivos

Se asocia con cada sucesor en la lista de "OPEN" el menos valor de los costes calculados y el valor anterior.

Se vuelve al paso de quitar de la lista "OPEN" el nodo con el coste más pequeño.

2.1.4 Simulaciones

Este apartado sirve para ver distintas simulaciones de este algoritmo y así comprender mejor su funcionamiento en diferentes casos.

CASOS	EXPLICACIÓN
Caso 1	Es una simulación simple de como funciona
Caso 2	Caso en el que el vehículo no puede salir
Caso 3	Simulación de un pasillo

Tabla 1 Casos de las Simulaciones de A*

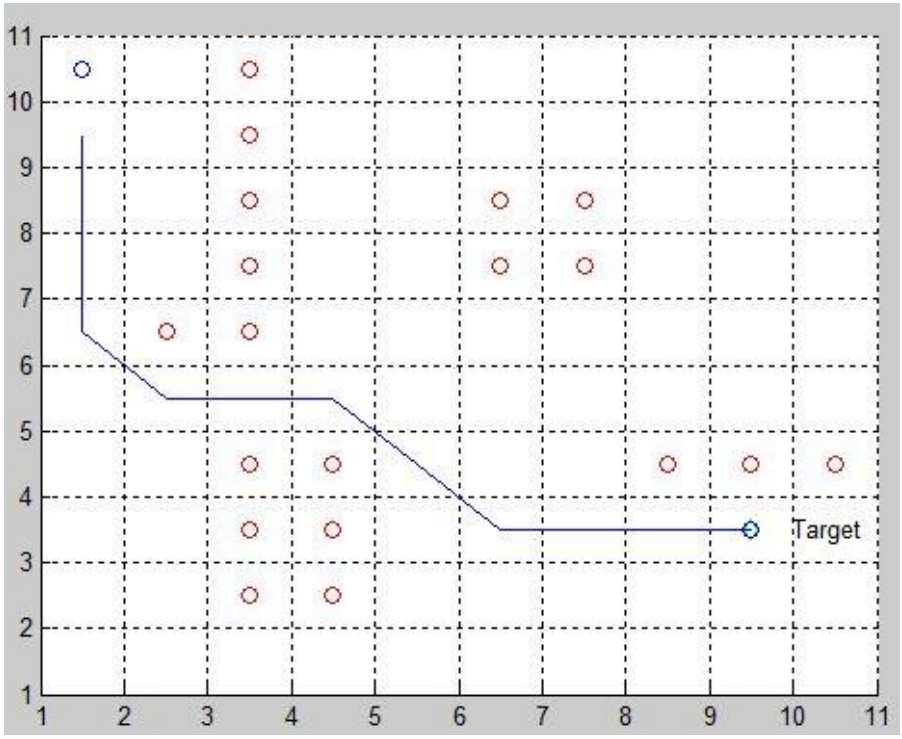

Figura 2-2 Simulación de A*

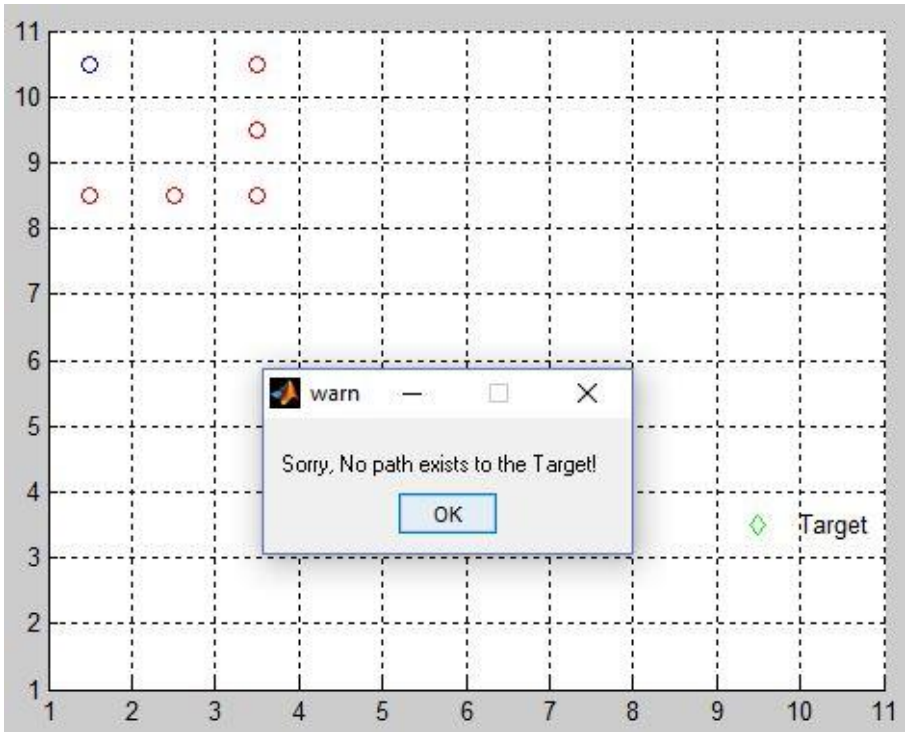

Figura 2-3 Caso de de ruta no posible

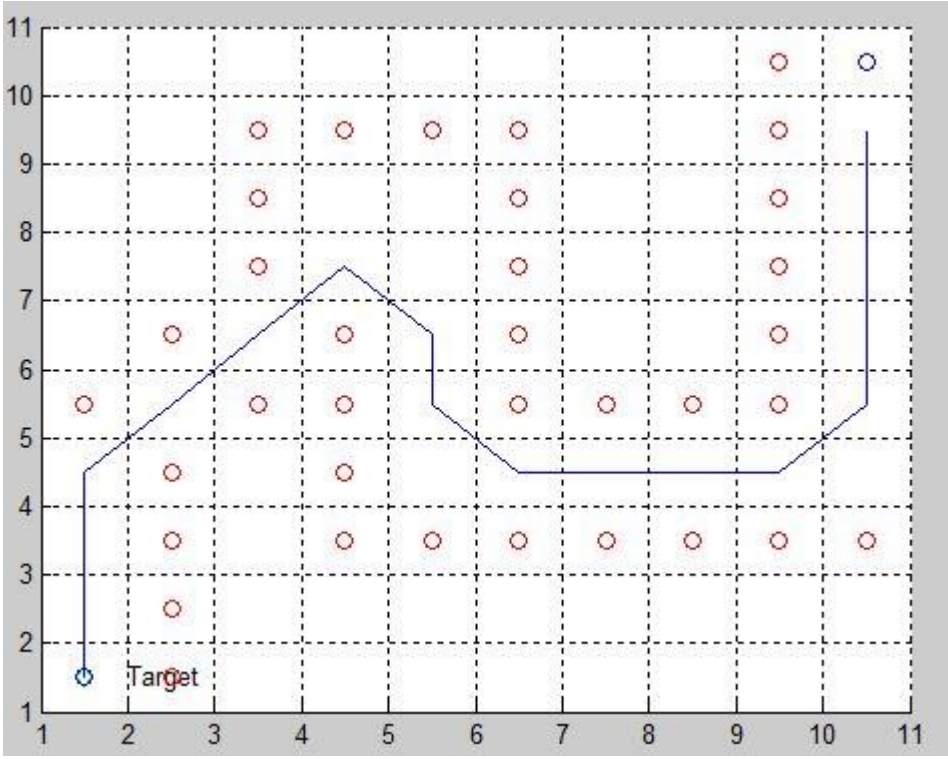

Figura 2-4 Simulación de pasillo

2.3. Descomposición en Celdas

Cómo el nombre del algoritmo indica, se basa en la descomposición de una zona en celdas en las que el vehículo circula. La zona contiene obstáculos (polígonos) que el vehículo debe de rodear en el caso de encontrarse en su trayectoria. Las celdas se forman a partir de los vértices de los objetos mencionados anteriormente.

La descomponsición del espacio de trabajo en celdas permite al robot conocer el espacio libre, el cual corresponde al conjunto de celdas que no contienen partes de algún obstáculo.

2.2.1 Descripción

Con la descomposición de celdas, el planificador de rutas del robot tiene como objetivo encontrar un grafo de adyyacencia que indique cuáles celdas libres de obstáculos comparten una frontera común, con el objetivo de identificar el espacio libre por donde se puede desplazar el robot. Los nodos del grafo corresponden a las celdas y las aristas conectan nodos de celdas adyacentes.

2.2.2 Funcionamiento

La descomposición de celdas en el planificador se realiza siguiendo los siguientes pasos:

1) Se construye el espacio de configuración asociado al espacio de trabajo del robot sobre el cual se identificarán las celdas.

2) Se construyen las celdas de acuerdo con las indicaciones del algoritmo.

3) Se construye el grafo de adyacencias.

4) Se determinan las celdas que contienen el comienzo y el final, respectivamente.

5) Se busca una ruta dentro del grafo de adyacencia.

Las celdas tienen una estructura simple, lo cual permite que sean cubiertas con movimientos sencillos del robot, tales como giros y avances en línea recta, ya que una vez el robot visita cada celda, el cubrimiento puede

ser reducido a encontrar un camino a través del grafo de adyacencia.

Dentro de los algoritmos para la descomposición por celdas, uno de los más conocidos es la descomposición trapezoidal, que está relacionada con la representación poligonal del espacio, y aunque requiere mayor complejidad computacional que otras técnicas, presenta mayor exactitud y completitud. Esta técnica hace referencia, en particular, a las celdas de dos dimensiones que tienen forma de trapezoides. Un ejemplo de este tipo de descomposición se puede ver en la figura 2-1, la cual detalla dos obstáculos y las posiciones de inicio y objetivo del robot.

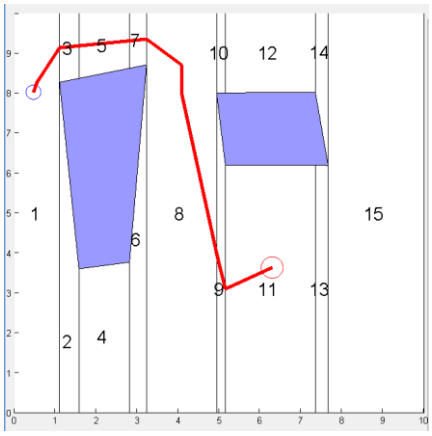

Figura 2-2-5 Ejemplo de Descomposción de celdas trapezoidal

En esta técnica se establece un sistema de coordenadas (x, y) para modelar el escenario en el cual se va a desplazar el robot. Así, el espacio libre estará delimitado por un polígono trapezoidal, y se asume que todos los obstáculos son polígonos.

Para realizar la descomposición en cada vértice *v* se dibujan dos segementos, uno llamado extensión vertical superior, y el otro, extensión vertical inferior. El primero se refiere a incrementar la coordenada *y*, y el segundo, a decrementarla. Las extensiones verticales superior e infreior comienzan en cada vértice y terminan cuando interceptan el primer borde del polígono que se localice inmediatamente arriba o debajo de *v*, respectivamente; o bien, cuando llegan a la frontera del escenario. De esta manera los vértices tendrán una extensión vertical superior, inferior, ambas, o ninguna (en el caso de esquinas cóncavas).

Una vez que las celdas que contienen el punto inical y el final estén definidas, se calcula el grafo de adyacencia para determinar la trayectoria, Sin embargo, el resultado de la búsqueda del grafo es una secuencia de nodos, y no una secuencia de puntos embebidos en el espacio libre. Es importante tener esto en cuenta, pues cada celda tendrá una posición y un área determinadas. Luego, se construye el camino conectando los puntos medios de las extensiones verticales a los centroides de cada trapezoide. Así se obtiene un camino libre de colisiones, a través del espacio libre que se deriva del grafo de adyacencia.

Para obtener la ruta se usa un grafo de conectividad. En el casi de la figura anterior el grafo seria el de la figura 2-2.

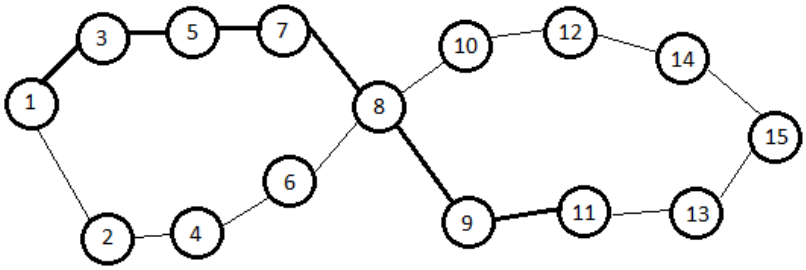

Figura 2-2-6 Grafo de conectividad de la figura 2-1

El código empleado para esta simulación funciona de esta manera; primero se crean las celdas donde los paso a seguir son:

- Desde cada vértice de los polígonos se dibuja una recta vertical que pase por ese punto.
- En el caso de que la recta llegue a coincidir con la arista de otro polígono terminaría aquí la zona (a continuación se muestra una imagen de ejemplo).
- Cada zona entre las rectas es una celda.

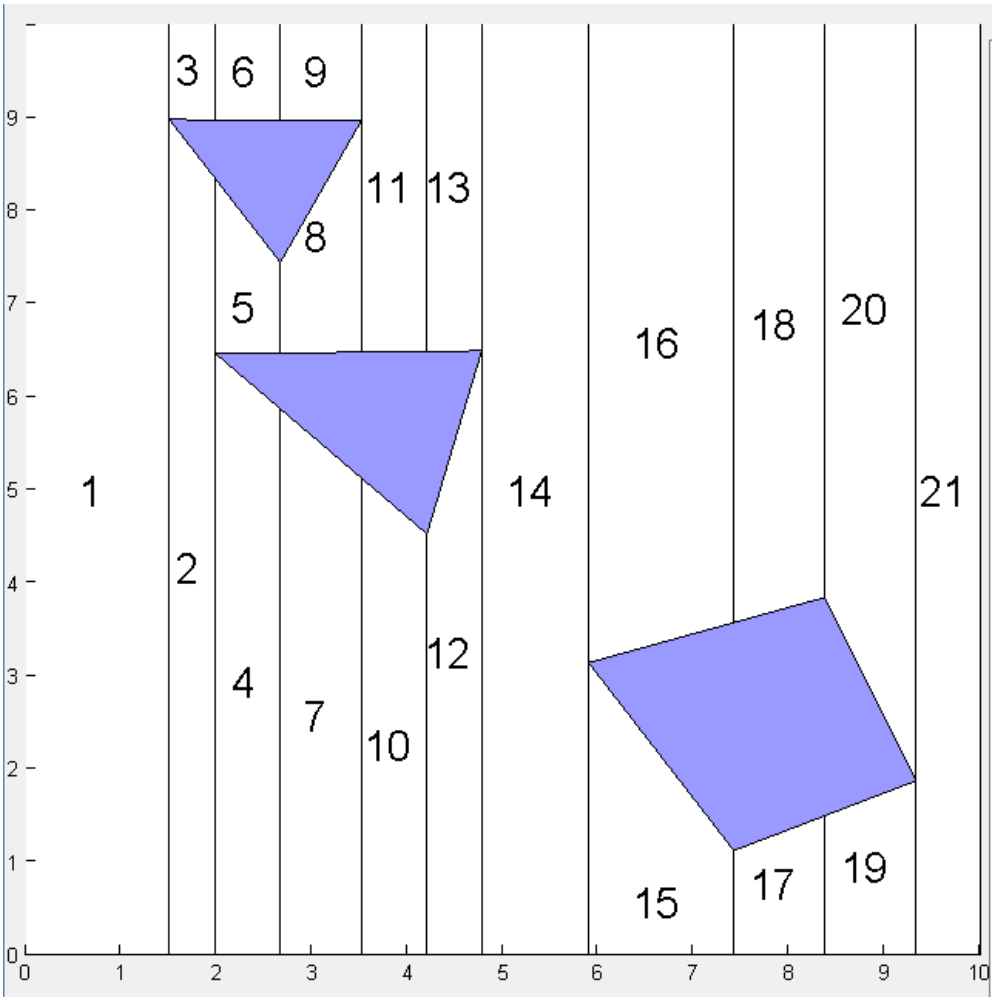

Figura 2-7 Ejemplo de Creación de Celdas

Lo siguiente que hace el código es elegir desde que zona parte el vehículo y donde acaba para luego elegir la ruta a seguir.

Para determinar la ruta se hace de la siguiente forma:

- Se guarda la información de las celdas en las que se encuentran el inicio y el final.
- Se determina las celdas adyacentes y uniendo los puntos medios de cada recta que forman las celdas y se determina la ruta más corta de entre las posibles.
- Para obtener estas rutas se usa un grafo de conectividad como el siguiente:

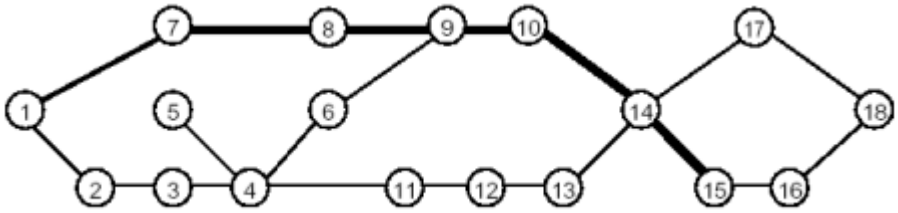

Figura 2-8 Grafo de conectividad

2.2.3 Simulaciones

Como en el caso anterior se mostrarán diferentes casos para su mejor comprensión.

CASOS	EXPLICACIÓN
Caso 1	Simulación simple
Caso 2	Simulación simple 2
Caso 3	Es la simulación con la que se compararan más tarde los 4 algoritmos

Tabla 2 Casos de simulaciones de Descomposición en Celdas

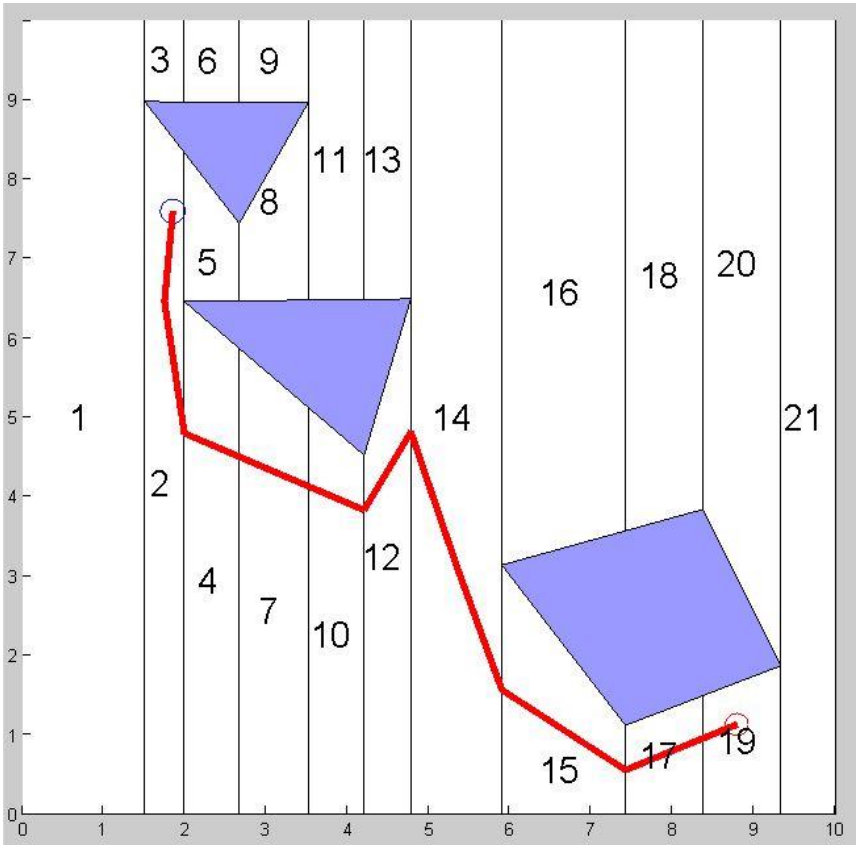

Figura 2-9 Simulación 1 de Descomposición en Celdas

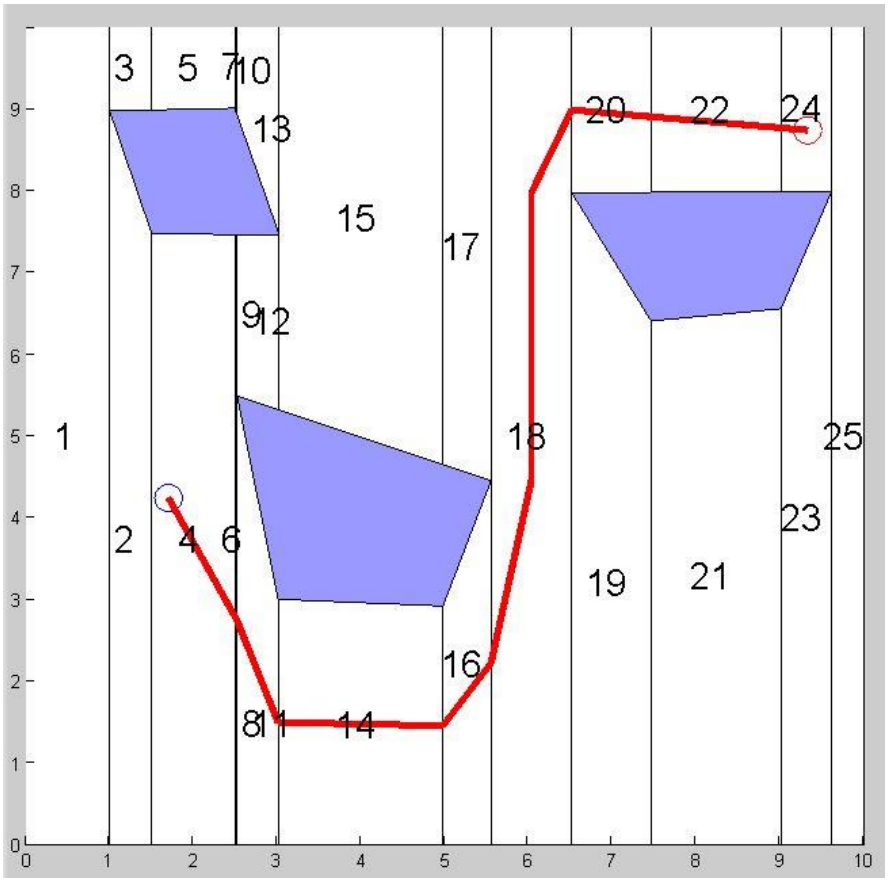

Figura 2-10 Simulación 2 de Descomposición en Celdas

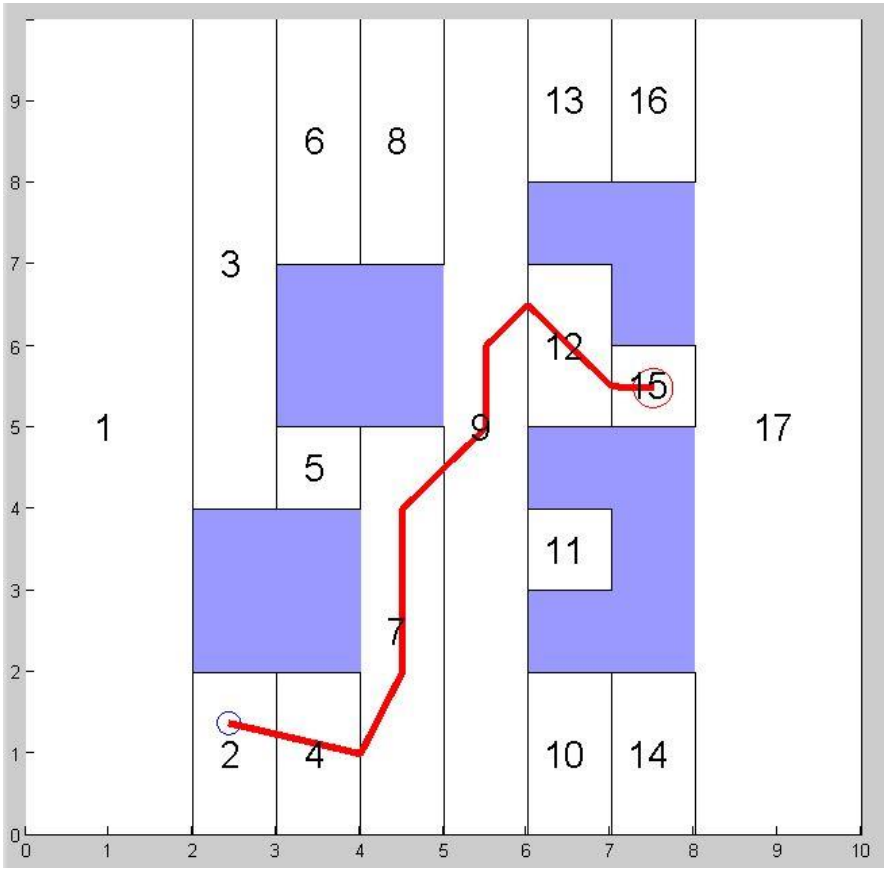

Figura 2-11 Simulación 3 de Descomposición en Celdas

2.4. RRT y RRT*

Un Rapidly-exploring Random Tree (RTT) es un algoritmo diseñado para buscar eficientemente espacios no convexos de alta dimensión mediante la construcción aleatoria de un árbol de relleno de espacio. El árbol se construye de forma incremental a partir de muestras extraídas al azar del espacio de búsqueda y está intrínsicamente sesgada para crecer hacia grandes áreas no buscadas del problema. RRTs fueron desarrollados por Steven M. LaValle y James J. Kuffner Jr. Ellos manejan fácilmente problemas con obstáculos y restricciones diferenciales (no holonómicas y kinodynamic) y han sido ampliamente utilizados en la planificación autónoma de movimiento robótico.

RRTs puede ser visto como una técnica para generar trayectorias de bucle abierto para sistemas no lineales con restricciones de estado. Un RRT también puede ser considerado como un método de Monte-Carlo para sesgar la búsqueda en las regiones más grandes Voronoi de un gráfico en un espacio de configuración. Algunas variaciones pueden incluso considerarse fractales estocásticos.

2.3.1 Rapidly-exploring Random Tree (RTT)

Un RRT crece en forma de árbol arraigado en la configuración inicial usando muestras aleatorias del espacio de búsqueda. A medida que se dibuja cada muestra, se intenta una conexión entre ella y el estado más cercano en el árbol. Si la conexión es factible (pasa completamente a través del espacio libre y obedece cualquier restricción), esto resulta en la adición del nuevo estado al árbol. Con un muestreo uniforme del espacio de búsqueda, la probabilidad de expandir un estado existente es proporcional al tamaño de su región de Voronoi. Como las regiones Voronoi más grandes pertenecen a los estados en la frontera de la búsqueda, esto significa que el árbol se expande preferentemente hacia grandes áreas no buscadas.

La longitud de la conexión entre el árbol y un nuevo estado está frecuentemente limitada por un factor de crecimiento. Si la muestra aleatoria está más alejada de su estado más cercano en el árbol de lo que permite

este límite, se usa un nuevo estado a la distancia máxima del árbol a lo largo de la línea a la muestra aleatoria en lugar de la muestra aleatoria misma. Las muestras aleatorias pueden ser vistas entonces como controlando la dirección del crecimiento del árbol mientras que el factor de crecimiento determina su velocidad. Esto mantiene el sesgo de llenado de espacio del RRT mientras que limita el tamaño del crecimiento incremental.

RRT crecimiento puede ser sesgada por el aumento de la probabilidad de muestreo estados de un área específica. La mayoría de las implementaciones prácticas de RRT hacen uso de esto para guiar la búsqueda hacia las metas del problema de la planificación. Esto se logra introduciendo una pequeña probabilidad de muestrear la meta en el procedimiento de muestreo estatal. Cuanta más alta es esta probabilidad, más avidamente el árbol crece hacia la meta.

El cómo el programa hace esto se explica a continuación:

El código dibuja incialmente el mapa que se le pasa a través de un documento de texto.

Tiene una variable que determina la distancia que se considera "cerca" de la meta por lo que tardará en más iteraciones en encontrar el recorrido o menos.

Siguiente paso es el de calcular la distancia entre el centro del círculo del robot y el del obstáculo.

Inspecciona el mapa y ve si hay posibilidad o no de colisión.

Lo siguiente que hace es calcular las diferentes posibilidades del camino en forma de árbol. Este árbol se hace usando números aleatorios y se calcula distancia entre ese punto y la meta y eso se pone en una variable "rrt" y se va calculando las iteraciones que se hacen.

Calcula el camino más corto posible que el robot puede realizar a través de los caminos ramificados y diseña el más aproximado.

2.3.2 Rapidly-exploring Random Tree Star (RRT*)

RRT * es un algoritmo basado en RRT, que se centra en mejorar la calidad de las soluciones. Mientras RRT simplemente conecta un aleatorio con el nodo más cercano en el árbol, RRT * busca los nodos en una esfera de volumen con un radio específico para encontrar el nodo que hace el menor costo para llegar a la muestra aleatoria.

Es un algoritmo probabilístico para resolver problemas de planificación de movimientos, es parecido al RRT pero al contrario que éste proporciona mayor rango de convergencia para una solución factible y es probabilísticamente óptimo.

Ventajas del RRT*:

1) La expansión de un RRT está fuertemente sesgada hacia las partes inexploradas del espacio de estados.
2) La distribución de los vértices en un RRT se aproxima a las distribuciones de muestreo, lo que lleva a un comportamiento coherente
3) Es probabilísticamente completo bajo condiciones muy generales.
4) El algoritmo RRT es relativamente simple , lo que facilita el análisis de rendimiento (esto también es una característica de las hojas de ruta probabilísticos prefiere)
5) Un RRT siempre permanece conectado, a pesar de que el número de aristas es mínima.
6) Un RRT puede considerarse como un módulo de planificación de la trayectoria , que se puede adaptar y se incorpora en una amplia variedad de sistemas de planificación
7) La totalidad de los algoritmos de planificación de ruta se pueden construir sin requerir la capacidad de dirigir el sistema entre dos estados prescritos, lo que amplía en gran medida la aplicabilidad de los RRT.

2.3.3 Simulaciones

Con este algoritmo vamos a ver varias simulaciones del mismo caso porque el RRT es diferente cada vez.

CASOS	EXPLICACIÓN
Caso 1	Simulaciones simples
Caso 2	Simulaciones simples 2
Caso 3	Son simulaciones en un pasillo

Tabla 3 Casos de RRT

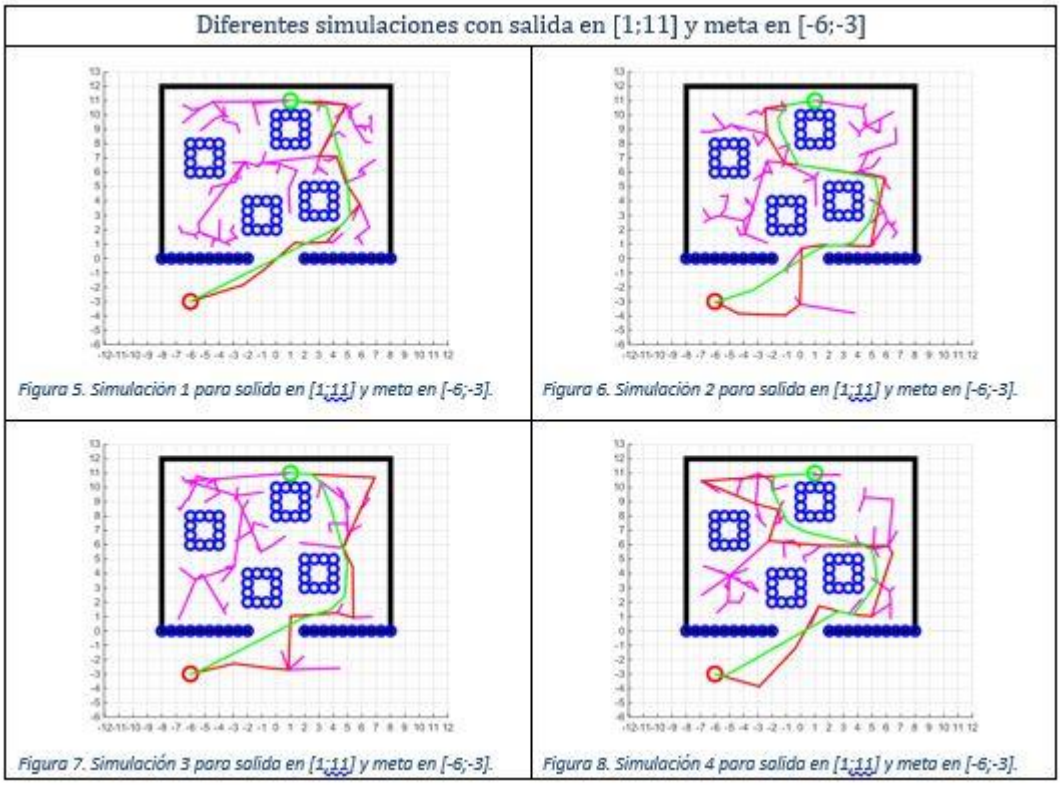

Figura 2-12 Simulación de RRT

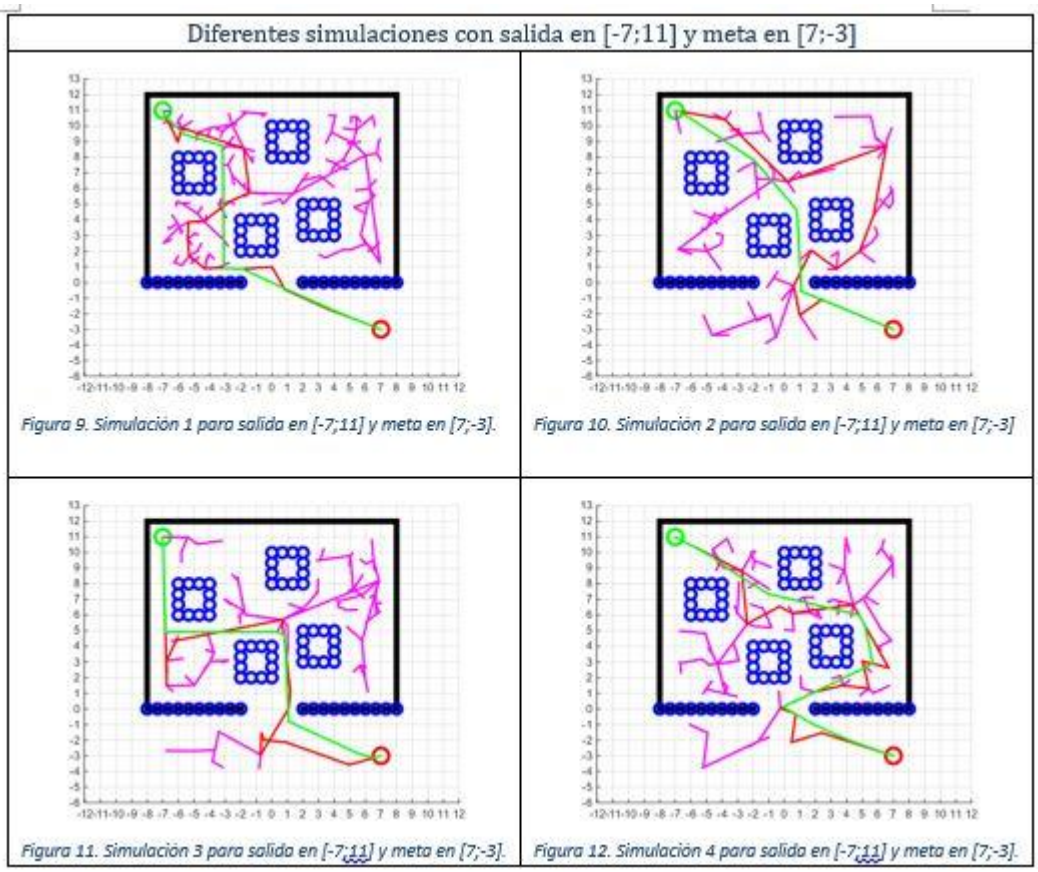

Figura 2-13 Simulación 2 de RRT

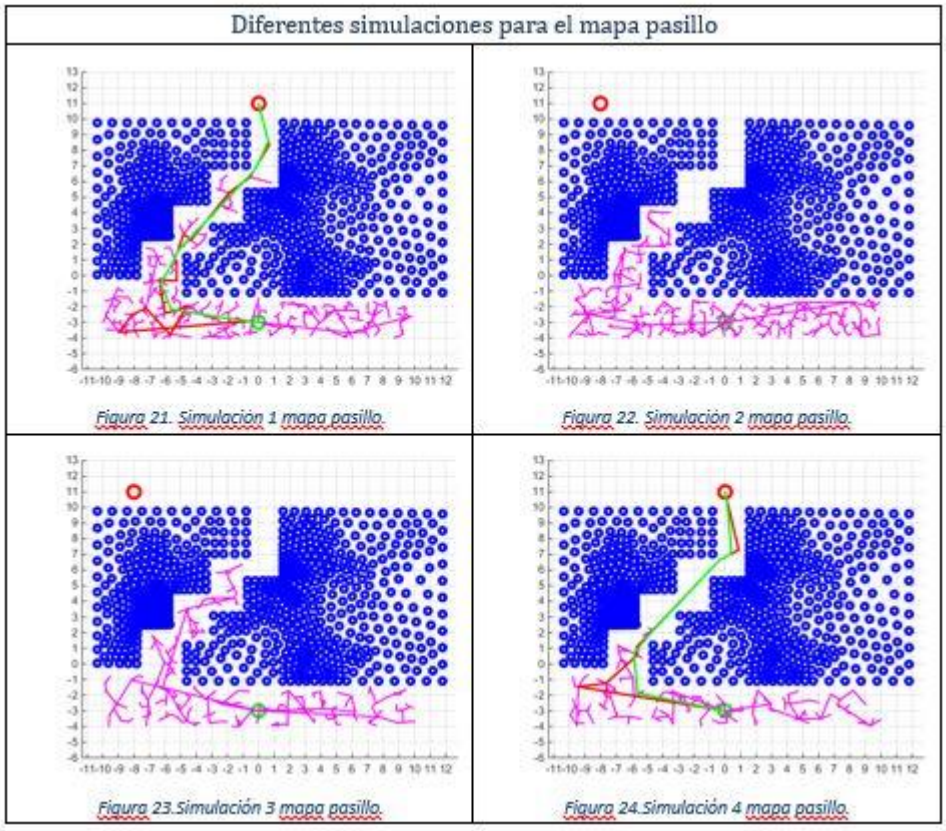

Figura 2-14 Simulación pasillo

2.3.4 Comparación entre RRT y RRT*

El algoritmo Rapidly-exploring Random Tree (RRT) se esfuerza por buscar el espacio para un camino factible mediante el "crecimiento" de un árbol de relleno de espacio.

Este árbol está arraigado en el punto inicial; en cada expansión incremental del crecimiento del árbol, se genera una semilla aleatoria, se inspeccionan nodos del árbol ya existente; basándose en ciertas reglas, se seleccionará un nodo como el siguiente que se va a expandir (nodo padre). El árbol comienza el crecimiento (navegación del vehículo) desde el padre, hasta que por ciertas reglas detienen el crecimiento del árbol, y este punto de parada (nodo hijo) se agrega al árbol.

Una ventaja de RRT es que, en cada expansión del árbol, las restricciones no holonómicas (aquellas que dependen de la velocidad, que sean integrable y no se puedan obtener derivando una restricción holónoma) podrían ser tomadas en cuenta en la navegación, por lo tanto, todos los nodos agregados en el árbol están disponibles para la dinámica del vehículo. La trayectoria devuelta de RRT es una innata y factible.

La desventaja, sin embargo, es que debido a la aleatoriedad en la generación de semillas, la trayectoria de RRT podría ser un zigzag. Para superar este problema, se puede sesgar la generación de las semillas, o un ajuste del RRT original denominado como RRT*.

Aparte de conectar al hijo con el nodo padre, RRT* también inspecciona cada uno de los nodos que se encuentran dentro de un vecindario del niño recién agregado. Si un nodo puede rastrear hasta la raíz del árbol a través del hijo a una distancia más corta que la conexión actual, el padre de este nodo se cambiará a hijo.

El RRT* básicamente intenta suavizar las ramas de los árboles en cada paso, por lo tanto el zigzag desaparecerá. Matemáticamente se ha demostrado que con el número de nodos alcanzado el infinito, la trayectoria devuelta desde RRT* será la más corta.

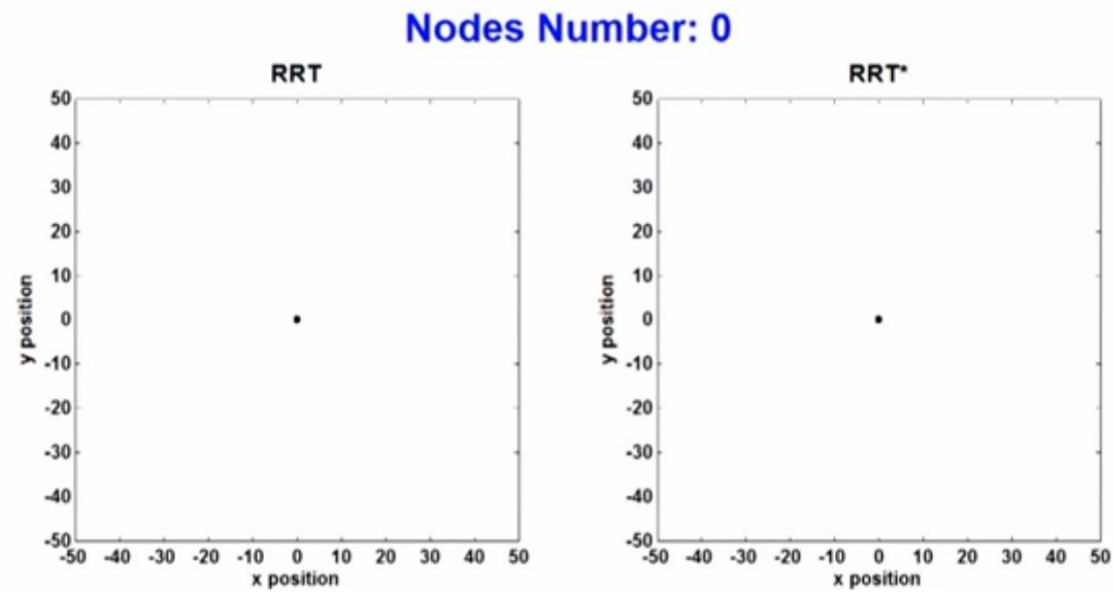

Figura 2-2-15 Estado inicial de ambos algoritmos. El nodo inicial empieza en (0,0).

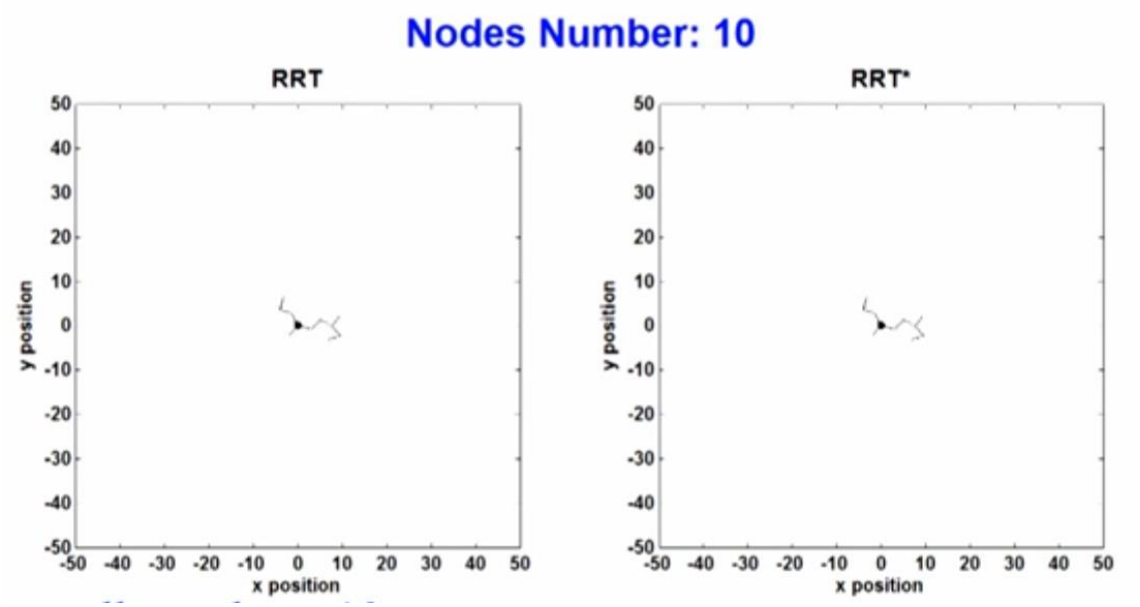

Figura 2-2-16 Vemos que para 10 nodos no existen diferencias.

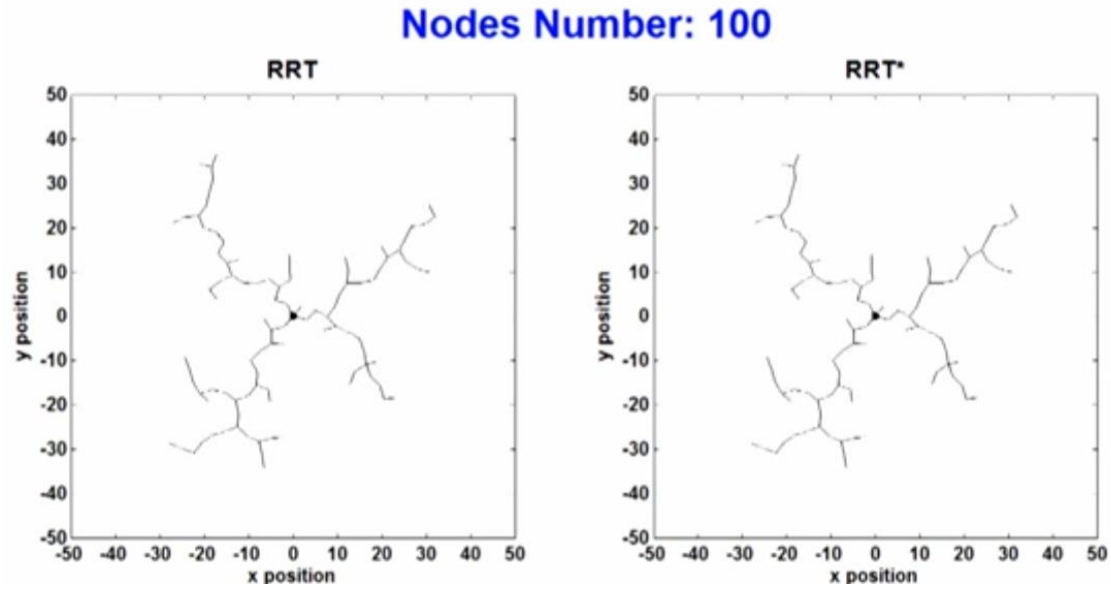

Figura 2-2-17 Para 100 nodos siguen siendo similares.

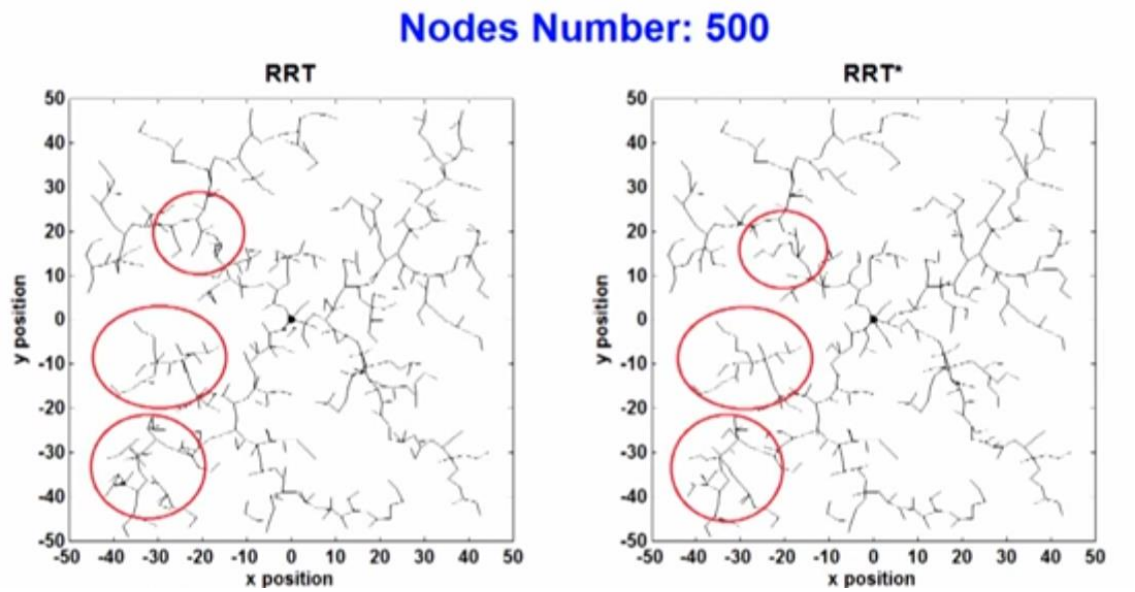

Figura 2-2-18 A los 500 nodos empiezan a haber diferencias.

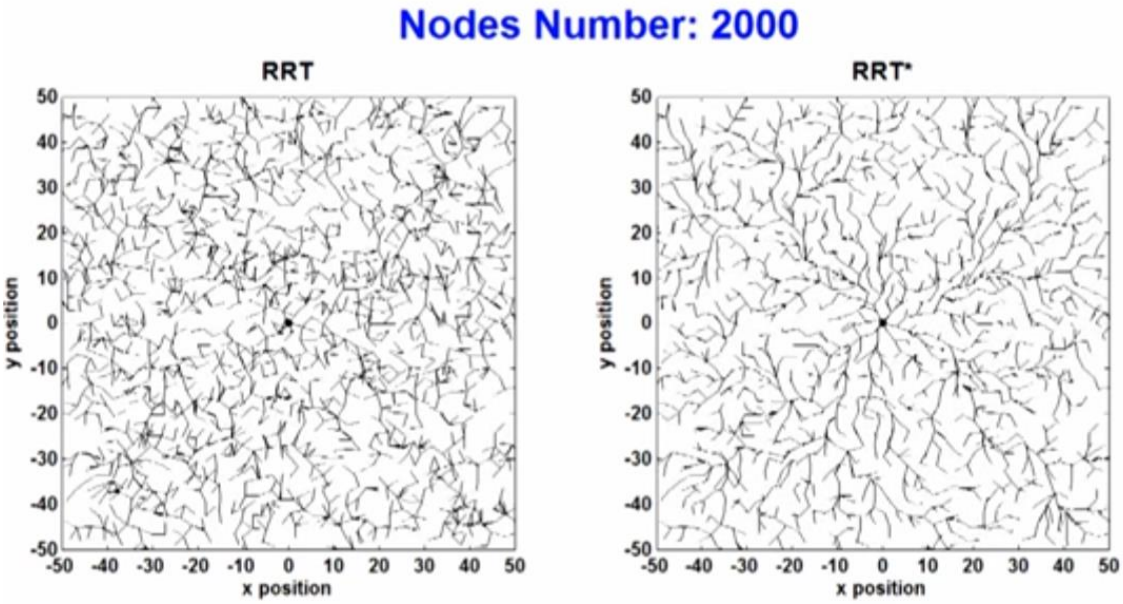

Figura 2-2-19 Con 2000 nodos la diferencia se nota bastante. Se observa como el RRT es más caótico que el RRT*.

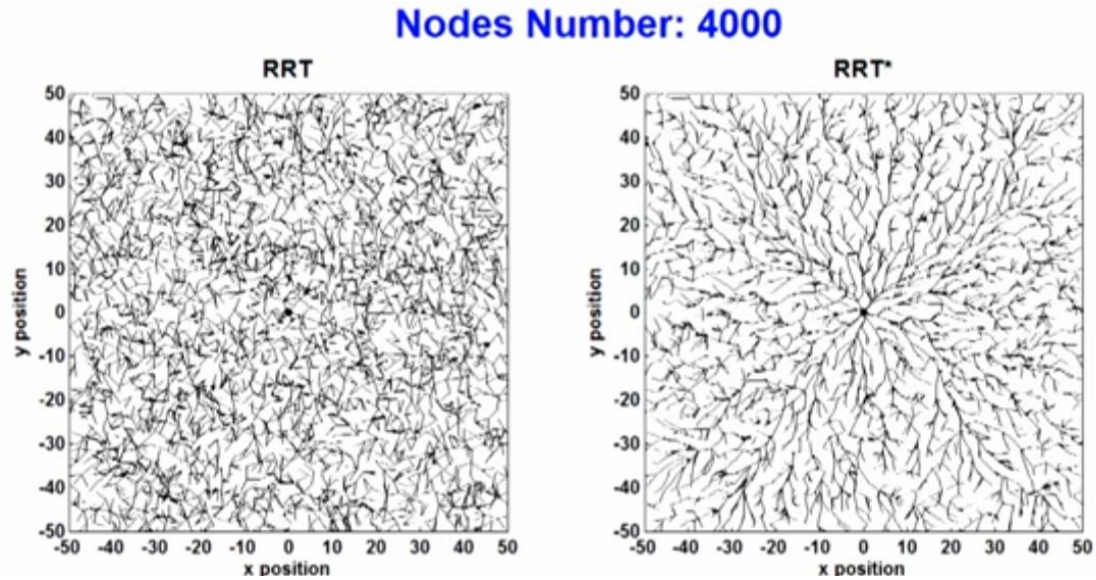

Figura 2-2-20 Con 4000 nodos la diferencia es bastante notable.

El árbol que crece a partir de RTT* está más organizado (menos caótico) que el de RRT. El camino generado por RRT* viene con menos zigzag, y por lo tanto podría ser encontrado un camino más corto.

Parece que, teóricamente, RRT* tiene cada pedazo de la vanguardia sobre el RRT. Pero para propósitos de la aplicación, ¿es esta la verdad?

Antes de responder hay que tener en cuenta que:

a) El ajuste básico de RRT* sobre RRT es la reconexión de los nodos, y el objetivo general de RRT* es correr por un camino más corto. Pero en aplicaciones reales, ¿siempre es necesario exigir un camino "más corto"?
b) La eficacia del RRT* se basa en el hecho de que se deben ahorrar algunos recursos computacionales para la inspección y reconexión de los nodos. ¿Vale la pena pagar estos preciados recursos a la ruta "más corta" no garantizada, sobre todo porque con los cuales podríamos utilizarlos para la detección del ambiente/evaluación de la amenaza,etc?
c) En b) por "no garantizado" quiero decir que sólo cuando el número de nodos se acerca al infinito, el camino retornado de RRT* será el más corto. Pero ¿y si sólo necesitamos 100 nodos? Como en anteriores parcelas árboles RRT y RRT* no son significativamente diferentes. ¿Es un poco un desperdicio invertir ese tiempo para el RRT*?
d) En a) planteamos la cuestión de la "optimalidad". Incluso de que necesitamos un camino óptimo, RRT fue diseñado para la exploración del espacio y el propósito de búsqueda de camino. La ventaja clave de RRT sobre otros algoritmos (A*, Artifical Potential Field, etc.) es que podría impartir las restricciones no-holonómicas a las nodos de los árboles; y puesto que para cada paso necesitamos navegar solamente un nodo, es relativamente más rápido que otros algoritmos. Por lo tanto, realmente es una sabia opción correr el riesgo de posiblemente perder estas dos ventajas, e ir a una optimalidad no garantizada, que incluso si hacemos la demanda, podría ser bien logrado mediante el empleo de RRT para obtener un camino factible primero, y utilizando algunos enfoques de optimización entonces.

Con todo lo dicho podríamos decir que en general el RRT* renta más usarlo que RRT en casos en los que tengamos un número elevados de nodos y para casos en los que los números de nodos son inferiores el uso de RRT es totalmente viable.

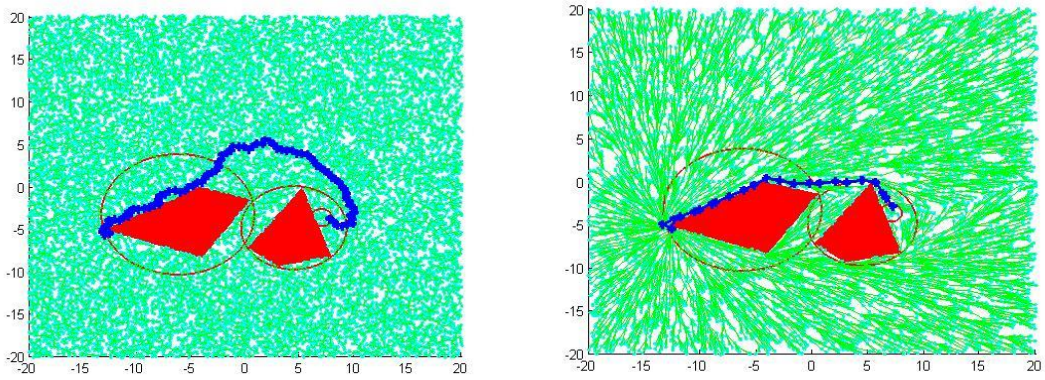

Figura 2-2-21 Izquierda RRT, derecha RRT*

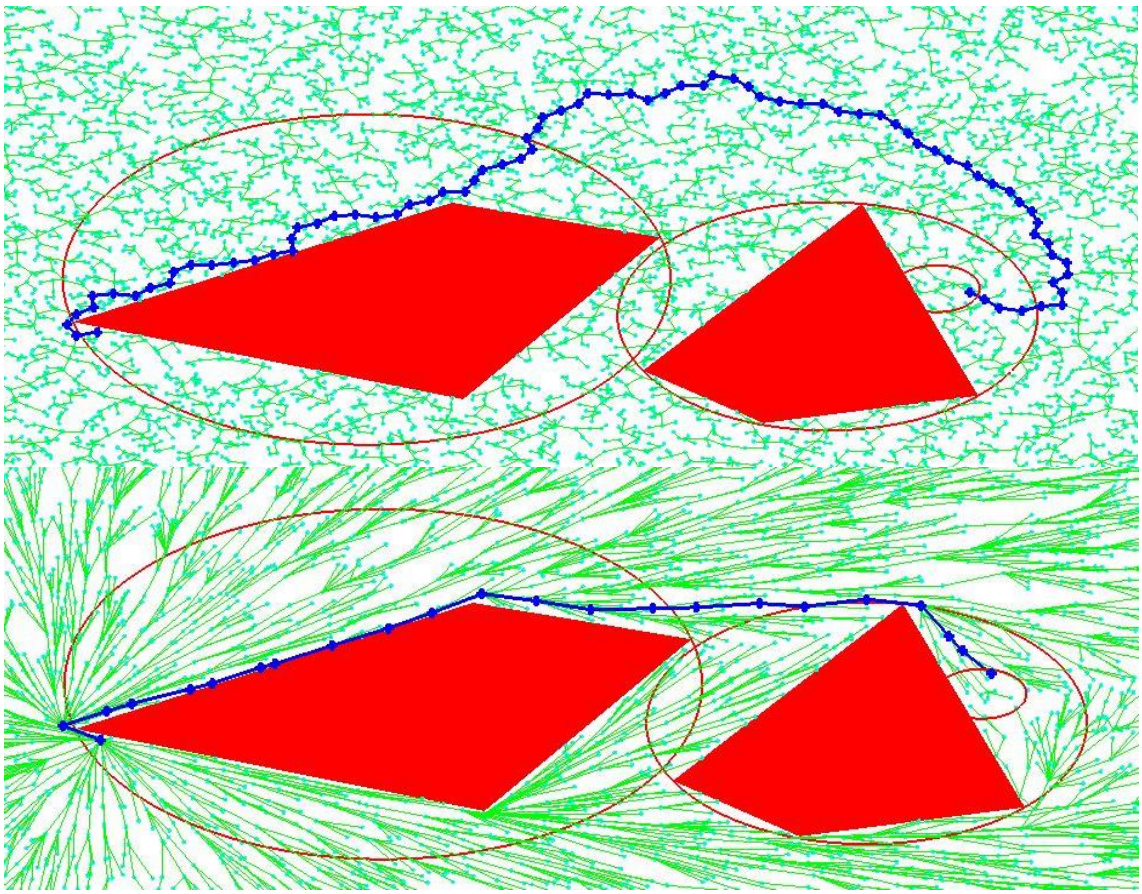

Figura 2-2-22 (Imágenes anteriores ampliadas) Arriba RRT, abajo RRT*

2.5. Voronoi y Dijkstra

2.4.1 Diagrama de Voronói

2.4.1.1 Descripción

Los Diagramas de Voronoi son uno de los métodos de interpolación más simples, basados en la distancia euclidiana, especialmente apropiada cuando los datos son cualitativos. Se crean al unir los puntos entre sí, trazando las mediatrices de los segmentos de unión. Las intersecciones de estas mediatrices determinan una serie de polígonos en un espacio bidimensional alrededor de un conjunto de puntos de control, de manera que el perímetro de los polígonos generados sea equidistante a los puntos vecinos y designan su área de influencia.

La definición formal de los Diagramas de Voronoi es la siguiente:

Denotemos a la distancia euclidiana entre dos puntos p y q por ||p-q||. En el plano entonces se tiene:

$$\|p\text{-}q\| = = \sqrt{(px - qx)^2 + (py - qy)^2}$$

Sea $\boldsymbol{P = p_1, p_2, \ldots, p_n}$ el conjunto de n puntos distintos en el plano que son denominados como sitios. Se define el diagrama de Voronói de $\boldsymbol{P}$ como la subdivisión del plano en n regiones, una para cada $\boldsymbol{p_i \in P}$, cumpliendo la propiedad de proximidad en la que un punto $\boldsymbol{q}$ pertenece a la región de un sitio $\boldsymbol{p_i}$ si y sólo si $\boldsymbol{\|q\text{-}p_i\| < \|q\text{-}p_j\|}$ para cada $\boldsymbol{p_j \in P, j \neq i}$. Se denotará al diagrama de Voronoi de $\boldsymbol{P}$ mediante $\boldsymbol{Vor(P)}$. Cada región que corresponde a un sitio $\boldsymbol{p_i}$ se denotará como $\boldsymbol{V(p_i)}$ y será llamada región de voronoi de $\boldsymbol{p_i}$.

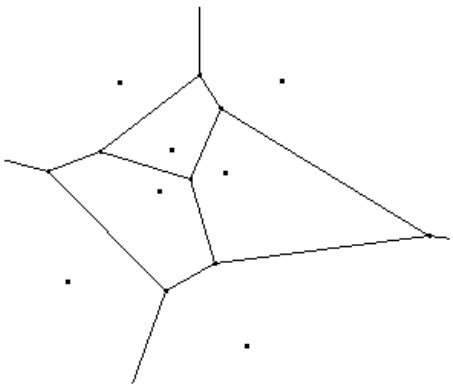

Figura 2-2-23 Los cuatros sitios más externos tiene asociados regiones de Voronói abiertas caracterizadas por las semi-aristas

La región de Voronoi para un sitio $\boldsymbol{p_i}$ está construida a partir de las intersecciones de los semiplanos formados al trazar los bisectores de $\boldsymbol{p_i}$ hacia los sitios $\boldsymbol{p_j, j \neq i}$.

$\boldsymbol{V(p_i)}$ está compuesta por la intersección de **n-1** semiplanos que conforman una región poligonal convexa que puede ser abierta o cerrada. La región está acotada por a lo más **n-1** vértices y a lo más **n-1** aristas. Se tienen estas cantidades de vértices y aristas debido a que los sitios más lejanos suele tener asociados regiones de Voronoi no acotadas conformadas por aristas y semi-aristas (aristas que tienen un vértice de inicio pero no uno final).

2.4.1.2 Propiedades

A continuación se enunciarán un conjunto de propiedades del Diagrama de Voronói donde se asume que no puede haber cuatro puntos del conjunto original ***P*** en posición cocircular. En caso de que esta situación no sea contemplada, entonces se deben considerar una gran cantidad de detalles que deben ser agregados a las diferentes propiedades.

- **Teorema 1.** Sea ***P*** un conjunto de **n** sitios en el plano. Si todos los sitios son colineales, entonces **Vor(P)** consiste en **n-1** líneas paralelas. De otra forma, **Vor(p)** es conexo y sus aristas podrían ser segmentos o semi-líneas.
- **Teorema 2.** Sea $\mathbf{n \geq 3}$, el número de vértices en el diagrama de Voronói de un conjunto de **n** sitios en el plano es a lo más **2n-5** y el número de aristas es a lo más **3n-6.**

La siguiente propiedad ayuda a caracterizar a los vértices y aristas que componen el diagrama de Voronói. Sin embargo, es necesario definir qué signifca el círculo vacio más grande de **q**, denotado como **Cp(q)**. Para un punto **q, Cp(q)** se define como el círculo más grande que tiene a **q** como su centro y no contiene ningún otro sitio de **P** en su interior.

- **Teorema 3.** Para un diagrama de Voronói **Vor(P)** de un conjunto de sitios **P** lo siguiente se cumple:
  - Un punto **q** es un vértice de **Vor(P)** si i sólo si su más grande círculo vacío **Cp(q)** contiene tres o más sitios en su contorno.
  - El bisector entre los sitios $\mathbf{p_i}$ y $\mathbf{p_j}$ define una arista de **Vor(P)** si y sólo si hay un punto **q** sobre el bisector tal que **Cp(q)** contiene tanto a $\mathbf{p_i}$ y a $\mathbf{p_j}$ en su contorno pero no otro sitio.
- **Teorema 4.** Cada vecino más cercano del sitio $\mathbf{p_i}$ en ***P*** define una arista de $\mathbf{V(p_i)}$.
- **Teorema 5.** La región de Voronói $\mathbf{V(p_i)}$ es abierta y sólo si $\mathbf{p_i}$ es un punto en la envolvente convexa del conjunto ***P***.
- **Teorema 6.** La gráfica dual del diagrama de Voronói define la triangulación de Delaunay.

2.4.1.3 Algoritmos para la construcción del diagrama de Voronói

2.4.1.3.1 Algoritmo por fuerza bruta

Una primera aproximación para la construcción del diagrama de Voronoi consiste en explotar la geometría de cada región de Voronoi. Por cada sitio $\mathbf{p_i \, \varepsilon \, P}$ se construirá su región de Voronoi mediante el cálculo explícito de los **n-1** semiplanos originados debido a los bisectores trazados con respecto a los demás sitios. A continuación, se computará la intersección de estos n-1 semiplanos para dar origen a **γ(pi)**.

Este algoritmo tiene muchas desventajas de entre las cuales se tienen las que a continuación se describen. En primera instancia, el cálculo explícito de los semi-planos y su intersección puede provocar problemas de precisión en la computadora generados, evidentemente, por una versión incorrecta de **Vor(P)**. El segundo inconveniente involucra que no se produce información inmediata que se pueda aprovechar acerca del vecindario de cada sitio. Finalmente, dado que se trata de un algoritmo ineficiente, no resulta extraño descubrir que su complejidad computacional sea alta. El algoritmo está en el orden de:

$$(n \ 2 \log n)$$

2.4.1.3.2 Algoritmo divide y vencerás

Este método primero divide un problema en dos subproblemas más pequeños de modo que cada subproblema sea idéntico al problema original, excepto porque su tamaño o dimensión es menor. Luego, ambos subproblemas se resuelven y las subsoluciones se fusionan en la solución final.

Obviamente, también estos dos subproblemas pueden resolverse aplicando la estrategia divide-y-vencerás, es decir, se pueden resolver de manera recurrente o recursiva.

La resolución de un problema mediante esta técnica consta fundamentalmente de los siguientes pasos:

1. En primer lugar ha de plantearse el problema de forma que pueda ser descompuesto en **k** subproblemas del mismo tipo, pero de menor tamaño. Es decir, si el tamaño de la entrada es **n**, hemos de conseguir dividir el problema en k subproblemas (donde $\mathbf{1 \leq k \leq n}$), cada uno con una entrada de tamaño **n/k** y donde $\mathbf{0 \leq n/k < n}$. A esta tarea se le conoce como división.
2. En segundo lugar han de resolverse independientemente todos los subproblemas, bien directamente si son elementales o bien de forma recursiva. El hecho de que el tamaño de los subproblemas sea estrictamente menor que el tamaño original del problema nos garantiza la convergencia hacia los casos elementales, también denominados casos base.
3. Por último, combinar las soluciones obtenidas en el paso anterior para construir la solución del problema original.

Dado el problema de construir el diagrama de Voronoi para el conjunto **P** de sitios, ahora se dividirá a éste último en dos subconjuntos **P1** y **P2**, con aproximadamente el mismo tamaño, de los que se debe encontrar su diagrama de Voronoi independientemente. Finalmente, **Vor(P1)** y **Vor(P2)** deben ser unidos para poder obtener **Vor(P).**

Dada una petición **P1 P2** de **P**, sea **σ(P1,P2)** el conjunto de aristas de Voronoi que son compartidas por pares de regiones de Voronoi **γ(pi ε P1)** y **γ(pi ε P2).** La colección de aristas **σ(P1,P2)** es el conjunto de aristas de una subgráfica de Vor(P) con las siguientes propiedades:

- **σ(P1,P2)** consta de ciclos y cadenas de aristas disjuntas. Si una cadena tiene una sola arista, ésta es una línea recta; de otra forma sus dos aristas extremas son rayos semi-infinitos.
- Si **P1** y **P2** son linealmente separados (si más de un punto pertenece a la línea de separación, todos estos puntos son asignados a un mismo conjunto de la partición), **σ(P1,P2)** consiste en una sola cadena monotónica.

Con el fin de separar a **P** en dos subconjuntos se le deberá ordenar con respecto a las abscisas y tomar la recta **m** que pase por la mediana, de tal forma que se tengan dos subconjuntos de aproximadamente el mismo tamaño. Adicionalmente, dada esta elección de recta de separación, se puede decir que σ parte al plano en una porción izquierda **πL** y una porción derecha **πR**. Con base en esto, se tiene la siguiente propiedad:

Si **P1** y **P2** son linealmente separados por una línea vertical con **P1** a la izquierda y **P2** a la derecha, entonces el diagrama de Voronoi **Vor(P)** es la unión de **Vor(P1) ∩ πL** y **Vor(P2) ∩ πR**.

A partir del último teorema se puede responder la pregunta inicial acerca de cómo se vinculan dos diagramas de Voronoi de dos particiones para generar el diagrama de Voronoi de todo el conjunto de sitios. El siguiente algoritmo establece la forma de calcular el diagrama de Voronoi mediante la técnica divide y vencerás:

1. Paso 1: Se divide el conjunto en dos mitades de, aproximadamente, el mismo tamaño

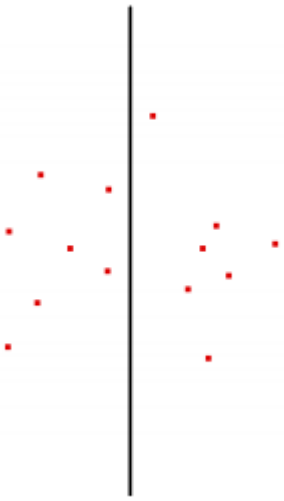

Figura 2-2-24 Divide y vencerás 1

2. Paso 2: Se calcula el diagrama de Voronoi de la izquierda

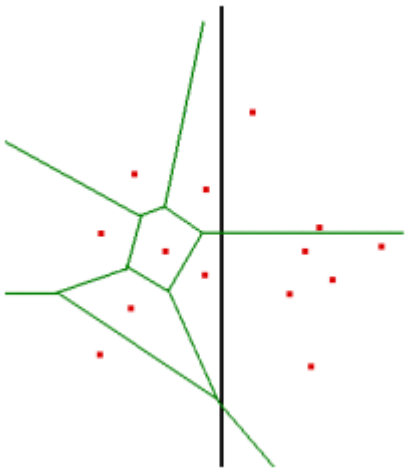

Figura 2-2-25 Divide y vencerás 2

3. Paso 3: Se calcula el diagrama de Voronoi de la derecha

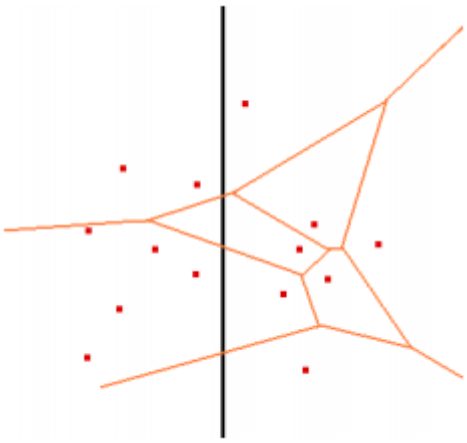

Figura 2-2-26 Divide y vencerás 3

4. Paso 4: Se calcula la cadena divisoria

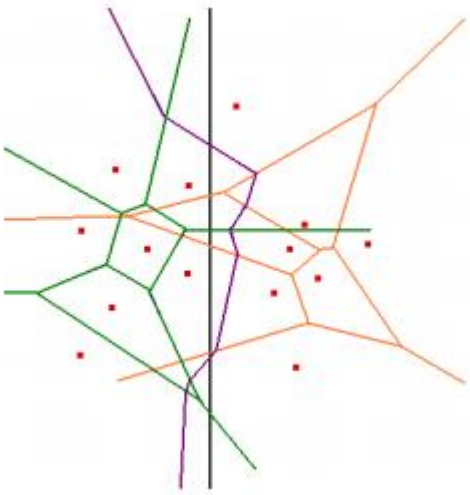

Figura 2-2-27 Divide y vencerás 4

5. Paso 5: Se eliminan todas las líneas de cada diagraam que no caen a su lado de la cadena divisoria

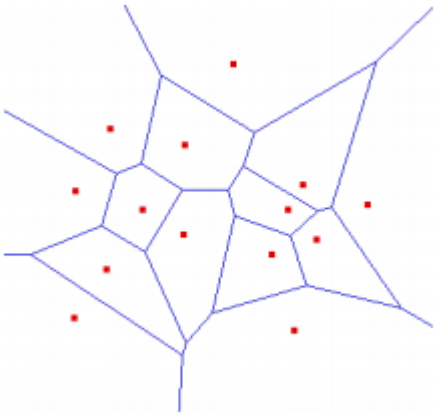

Figura 2-2-28 Divide y vencerás 5

2.4.1.3.3 Algoritmo divide y vencerás

El algoritmo de fuerza bruta previamente revisado tiene una complejidad O(n2 log n) , sin embargo, teniendo en cuenta la propiedad siguiente:

Para n ≥ 3, el número de vértices en el diagrama de Voronoi de un conjunto de n sitios en el plano es a lo más 2n - 5 y el número de aristas es a lo más 3n - 6.

Se puede suponer por tanto que hay una forma mucho más eficiente de encontrar el diagrama de Voronoi pues sus elementos constituyentes tienen complejidad O(n). En efecto, existe este algoritmo y es llamado Algoritmo de Fortune en honor a Steven Fortune quien lo inventó en 1986 y cuya complejidad es:

$$(n \log n)$$

El algoritmo de Fortune está basado en una de las técnicas clave dentro de la geometría computacional denominada barrido de recta. La esencia de esta técnica yace en suponer que existe una recta **L** que recorre el plano de arriba hacia abajo (o de izquierda a derecha, incluso en direcciones opuestas) y que a lo largo de su recorrido se interseca con las estructuras que deseamos procesar. Cuando se da esta intersección, se guarda cierta información de tal forma que ayude en los cálculos.

El problema con el diagrama de Voronoi es el de predecir cuándo y dónde se producirán los próximos eventos. Imagine que detrás de la línea de barrido ya se ha construido el diagrama de Voronoi basándose en los sitios que se han encontrado hasta ahora en el barrido. La dificultad es que un sitio que queda por delante de la línea de barrido podría generar un vértice de Voronoi que se encuentre detrás de la línea de barrido. Son precisamente estos eventos no anticipados los que complican el diseño de un algoritmo de barrido del plano.

Fortune hizo la inteligente observación de que podía calcularse el diagrama de Voronoi mediante barrido del plano construyendo una versión “distorsionada” de éste pero que es topológicamente equivalente. Esta versión distorsionada del diagrama se basa en una transformación que modifica la forma en que las distancias son medidas en el plano. El diagrama resultante tiene la misma estructura topológica que el diagrama de Voronoi, pero sus aristas son arcos parabólicos, en vez de segmentos de línea recta.

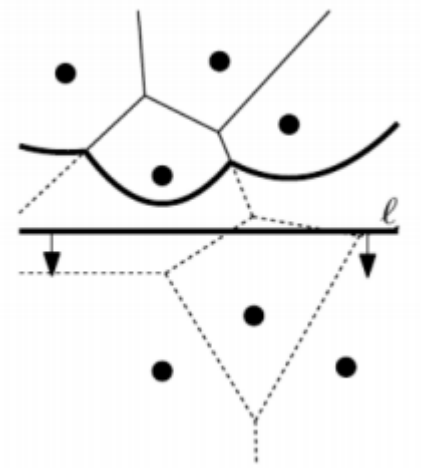

Figura 2-2-29 Fortune (barrido recta)

Es necesario que la información que se haya obtenido en regiones ya visitadas por la recta sea invariante. Es muy común que está técnica utilice dos tipos de estructuras de datos: cola de prioridades donde se guardan eventos que no son más que puntos donde la recta debe detenerse y un árbol binario de búsqueda donde se almacenan los elementos geométricos que se han intersecado con la recta y se necesita recordar para el procesamiento futuro.

Cabe resaltar que debido a que en la computadora no se puede emular tal cual el movimiento continuo de la recta de barrido, se requiere idear una forma de discretización del movimiento de la recta que sea procesable en la computadora, de ahí que los eventos sean tal discretización.

En la siguiente secuencia de imágenes se puede ver la evolución de la recta de barrido durante el proceso:

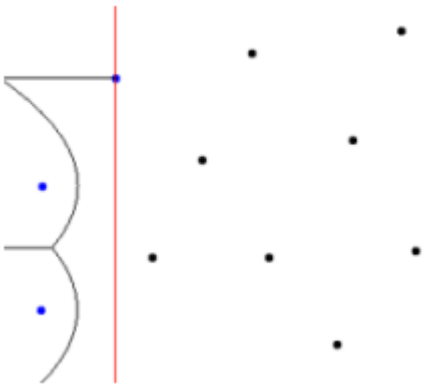

Figura 2-2-30 Fortune Progreso 1

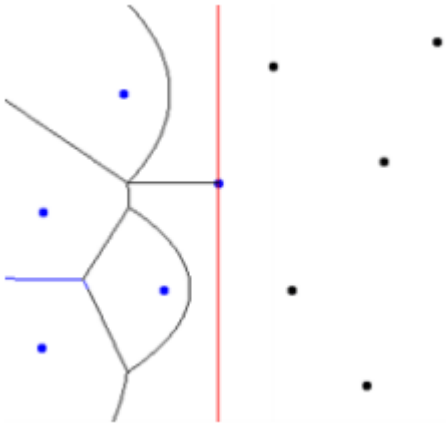

Figura 2-2-31 Fortune Progreso 2

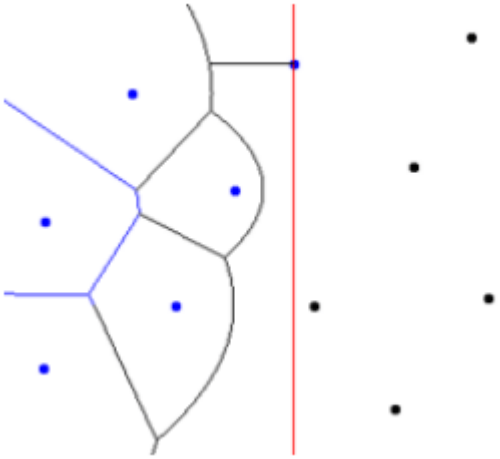

Figura 2-2-32 Fortune Progreso 3

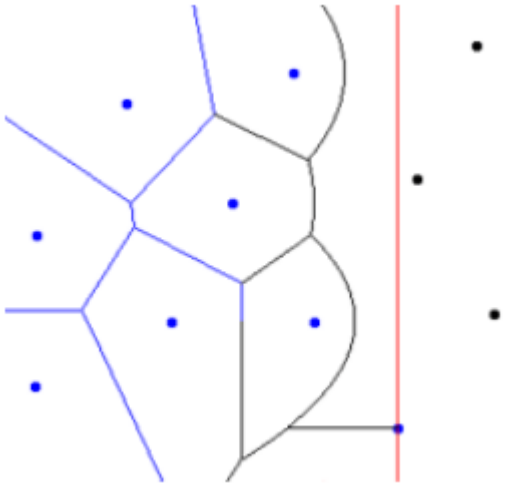

Figura 2-2-33 Fortune Progreso 4

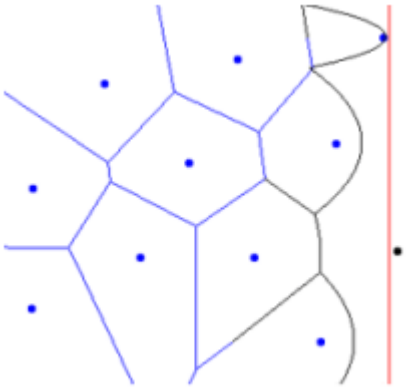

Figura 2-2-34 Fortune Progreso 5

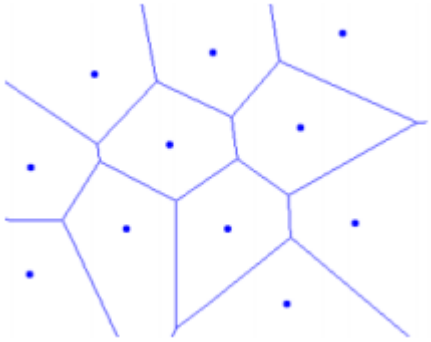

Figura 2-2-35 Fortune Progreso 6

Este algoritmo tendrá un coste computacional igual al algoritmo de Divide y Vencerás, sin embargo su implementación es mucho más sencilla, por lo que actualmente es el método más extendido para generar el diagrama.

2.4.2 Algoritmo Dijkstra

El algoritmo de Dijkstra, también llamado algoritmo de caminos mínimos, es un algoritmo para la determinación del camino más corto dado un vértice origen al resto de los vértices en un grafo con pesos en cada arista. Su nombre se refiere a Edsger Dijkstra, quién 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 unfirome, y como tal, no funciona en grafos con aristas de coste negativo (al elegir siempre el nodo con distancia menor, pueden quedar excluidos de la búsqueda nodos que en próximas iteraciones bajarían el costo general del camino al pasar por una arista con costo negativo).

2.4.2.1 Funcionamiento del algoritmo

Teniendo un grafo dirigido ponderado de N nodos no aislados, sea x el nodo inicial, un vector D de tamaño N guardará al final del algoritmo las distancias desde x al resto de los nodos.

1. Inicializar todas las distancias en D con un valor infinito relativo ya que son desconocidas al principio, exceptuando la de x que se debe colocar en 0 debido a que la distancia de x a x sería 0.
2. Sea a = x (tomamos a como nodo actual).

3. Recorremos todos los nodos adyacentes de a, excepto los nodos marcados, llamaremos a estos nodos no marcados v_i.
4. Para el nodo actual, calculamos la distancia tentativa desde dicho nodo a sus vecinos con la siguiente fórmula:

   $$Dt(v_i) = D_a + d(a,v_i).$$

   Es decir, la distancia tentativa del nodo 'v_i' es la distancia que actualmente tiene el nodo en el vector D más la distancia desde dicho el nodo 'a' (el actual) al nodo v_i. Si la distancia tentativa es menor que la distancia almacenada en el vector, actualizamos el vector con esta distancia tentativa. Es decir: Si $dt(v_i) < D_{vi} \rightarrow D_{vi} = dt(v_i)$
5. Marcamos como completo el nodo a.
6. Tomamos como próximo nodo actual el de menor valor en D (puede hacerse almacenando los valores en una cola de prioridad) y volvemos al paso 3 mientras existan nodos no marcados.

Una vez terminado el algoritmo, D estará completamente lleno.

Matlab tiene una función que hace Voronoi pasándole una serie de puntos. Estos puntos se obtienen mediante la función "rand" de matlab también. Cuando se obtiene voronoi se pasa a unas funciones vx y vy.

Luego se dibujan los obstáculos y se borra el interior de los obstáculos y se guardan en variable vx y vy nuevas.

Una vez hecho esto se agregan el punto inicial y final y se van calculando la distancia entre los puntos.

Una vez esto se utiliza el algoritmo de Dijkstra para encontrar el camino más corto mirando entre las posibilidades.

Una vez hecho todo esto se muetra el resultado.

Un ejemplo gráfico de como funciona sería:

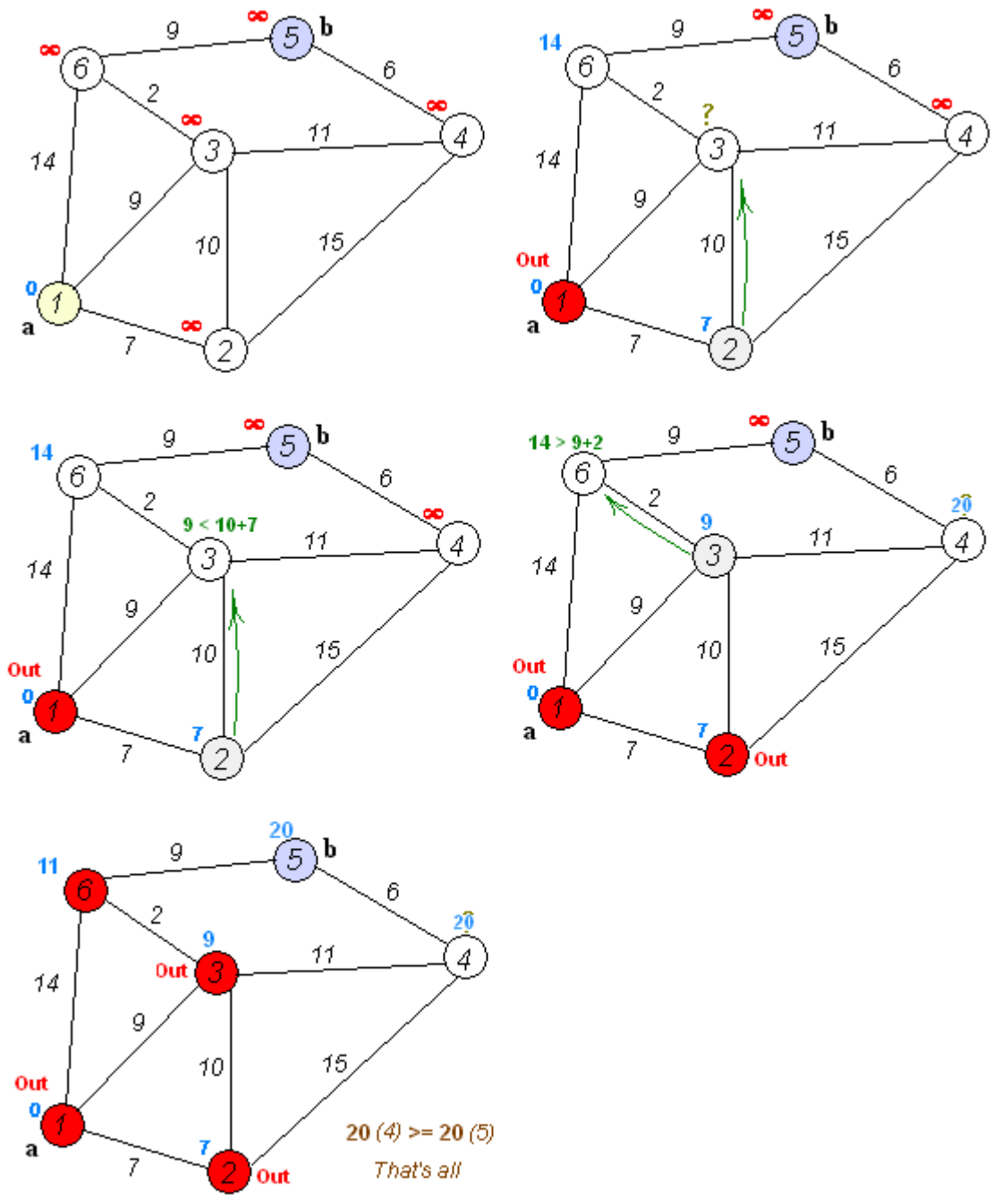

Figura 2-2-36 Ejecución del algoritmo de Dijkstra

2.4.3 Simulaciones

Se mostrarán tres casos distintos en el cual dos de ellas se observará como para el mismo caso la ruta es muy distinta.

CASOS	EXPLICACIÓN
Caso 1	Simulación simple
Caso 2	Es el mismo caso que el anterior pero aquí al no haber punto entre los obstáculos no se puede calcular trayectoria por ahí. Se demuestra como cambia en cada simulación.
Caso 3	Otra simulación cambiando el punto de origen y final

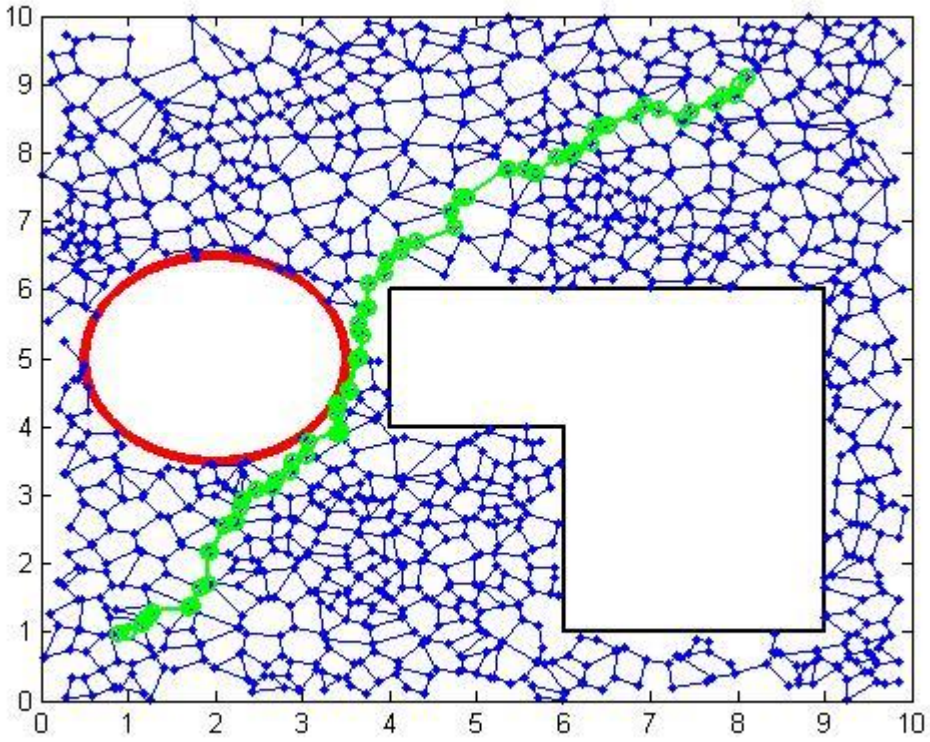

Figura 2-37 Simulación 1 Voronoi

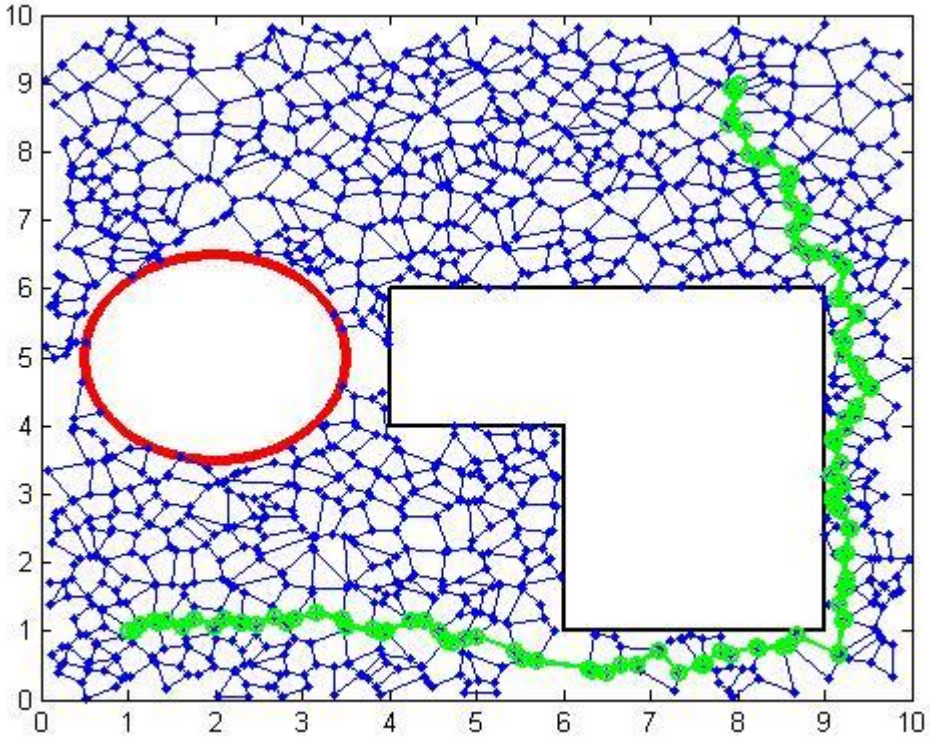

Figura 2-38 Simulación 2 Voronoi

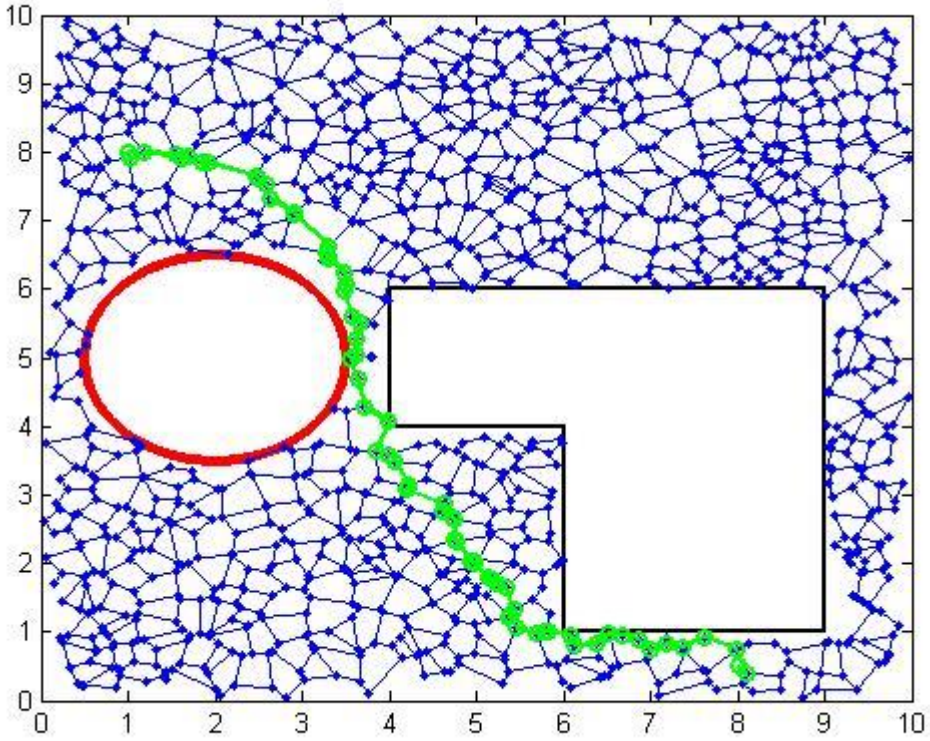

Figura 2-39 Simulación 3 Voronoi

3 COMPARACIÓN ENTRE MÉTODOS

En este apartado veremos como funciona cada uno de los algoritmos para un caso más complejo y así estudiarlos juntos y poder ver cuál método es mejor o peor y en que condiciones es mejor usar uno u otro algoritmo.

3.1 Simulaciones

A continuación se muestran las simulaciones hechas para un mismo mapa con los mismos obstáculos. Primero se pondrán las imágenes y se explicarán luego. Las simulaicones son las siguientes:

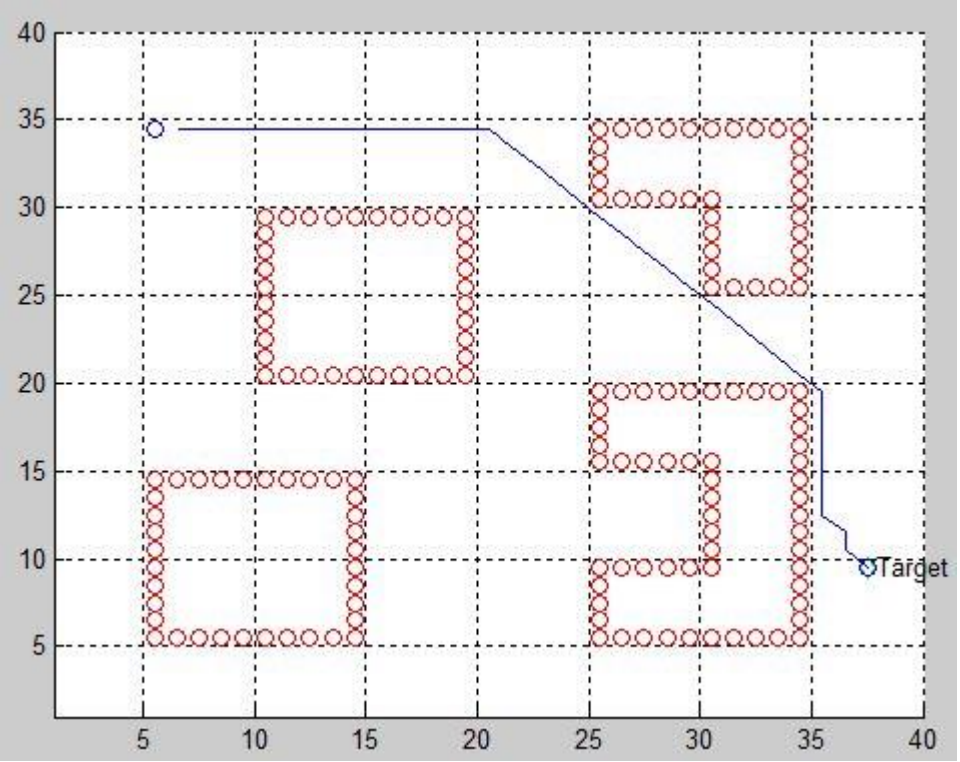

Figura 3-1 Simulación A*

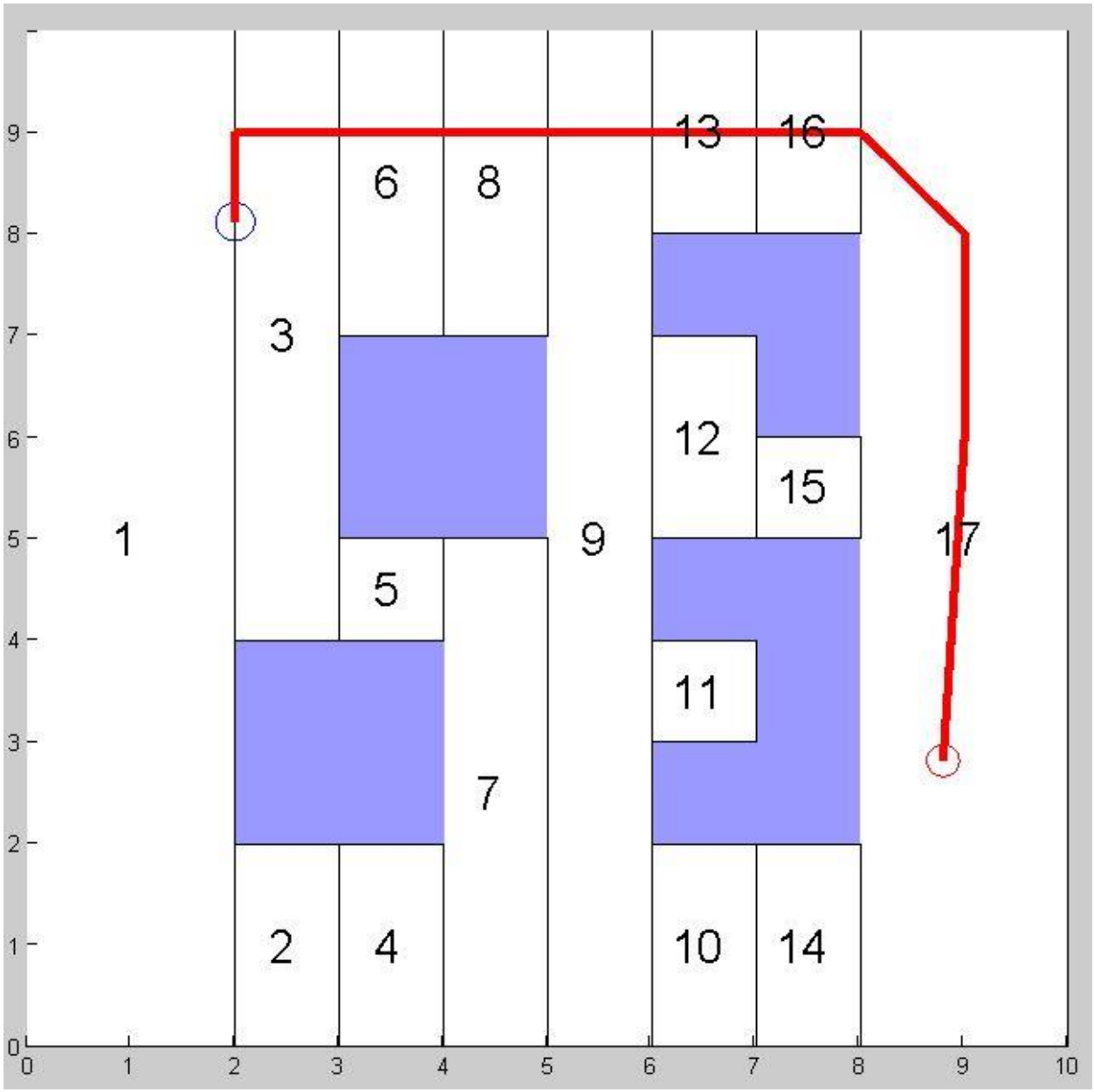

Figura 3-2 Simulación Descomposición en Celdas

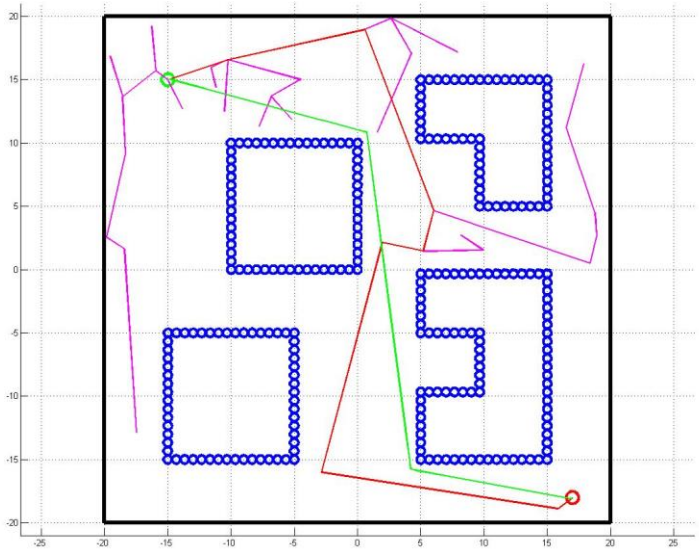

Figura 3-3 Simulación RRT (distancia 5)

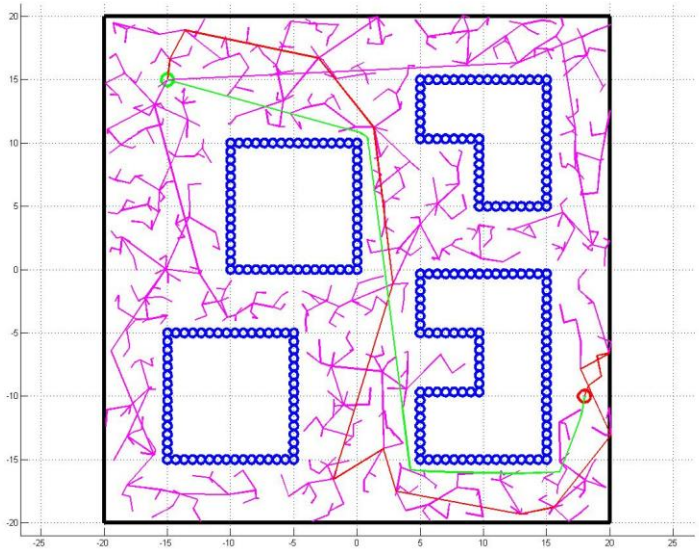

Figura 3-4 Simulación RRT (distancia 1)

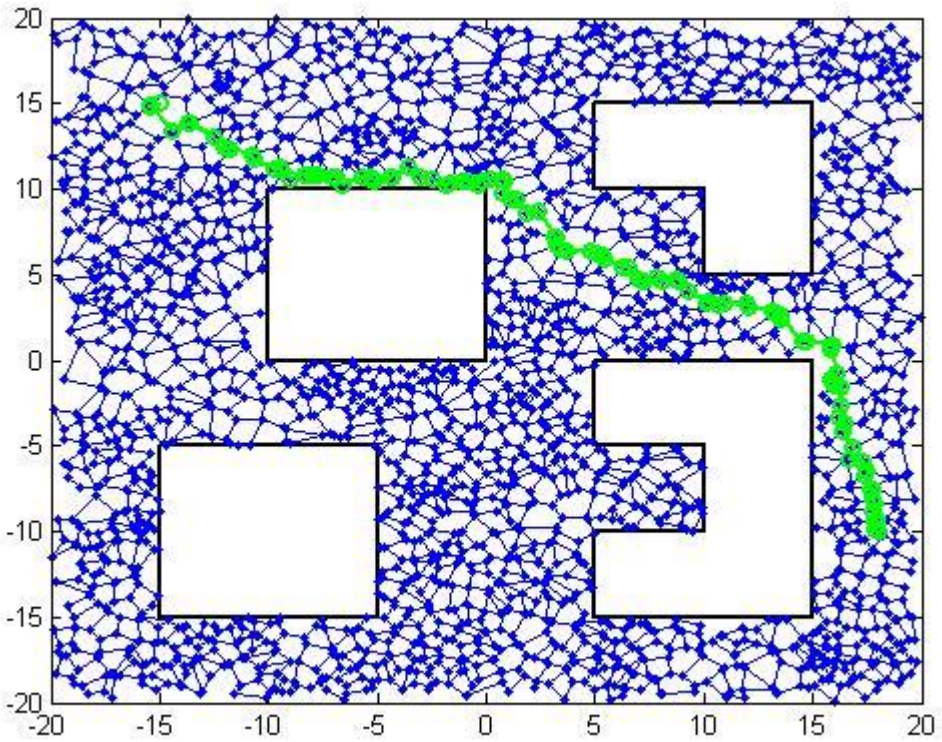

Figura 3-5 Simulación Voronoi (Simulación 1)

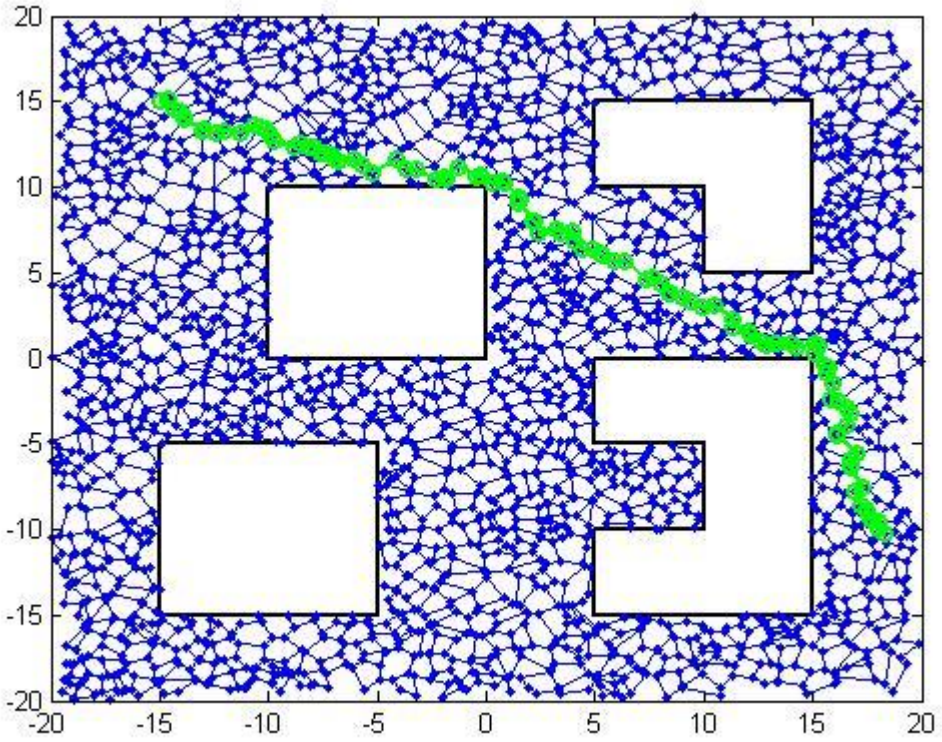

Figura 3-6 Simulación Voronoi (Simulación 2)

Como podemos observar tenemos un mismo mapa en el que se encuentran 4 obstáculos que el vehículo debe soltear.

El primer caso es el del algoritmo de A*, el segundo es el de Descomposición en Celdas (DC), los siguientes son de RRT y los últimos de Voronoi.

En el caso de RRT hay dos imágenes, la primera imagen hay menos ramas puesto que en el código la distancia que tomaba el algoritmo para saber si el objetivo estaba cerca o no era 5 y en el segundo caso era 1, por eso tarda más en buscar el objetivo.

Con Voronoi hay dos imágenes para demostrar que para los mismos valores al hacer dos simulaciones distintas el recorrido es distinto puesto que Voronoi no usa un valor fijo para hacer las regiones.

Como podemos observar los únicos recorridos parecidos han sido los de A* y Voronoi, en cambio los otros dos son diferentes.

Ahora veremos simulaciones en los que el punto de partida y el objetivo cambian.

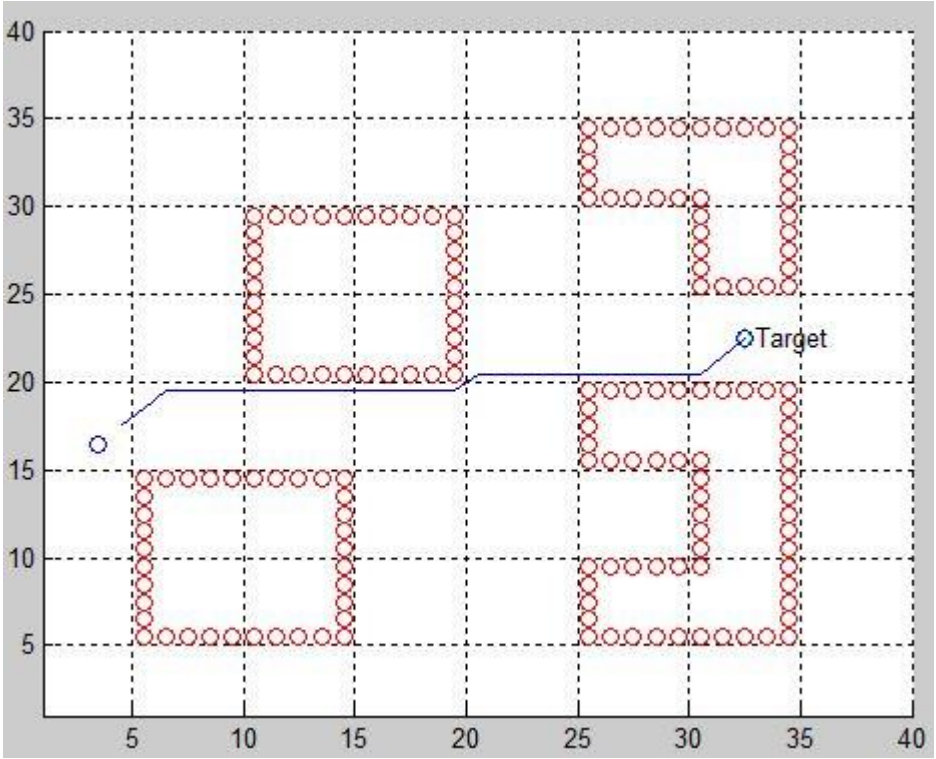

Figura 3-7 Simulación A* v2

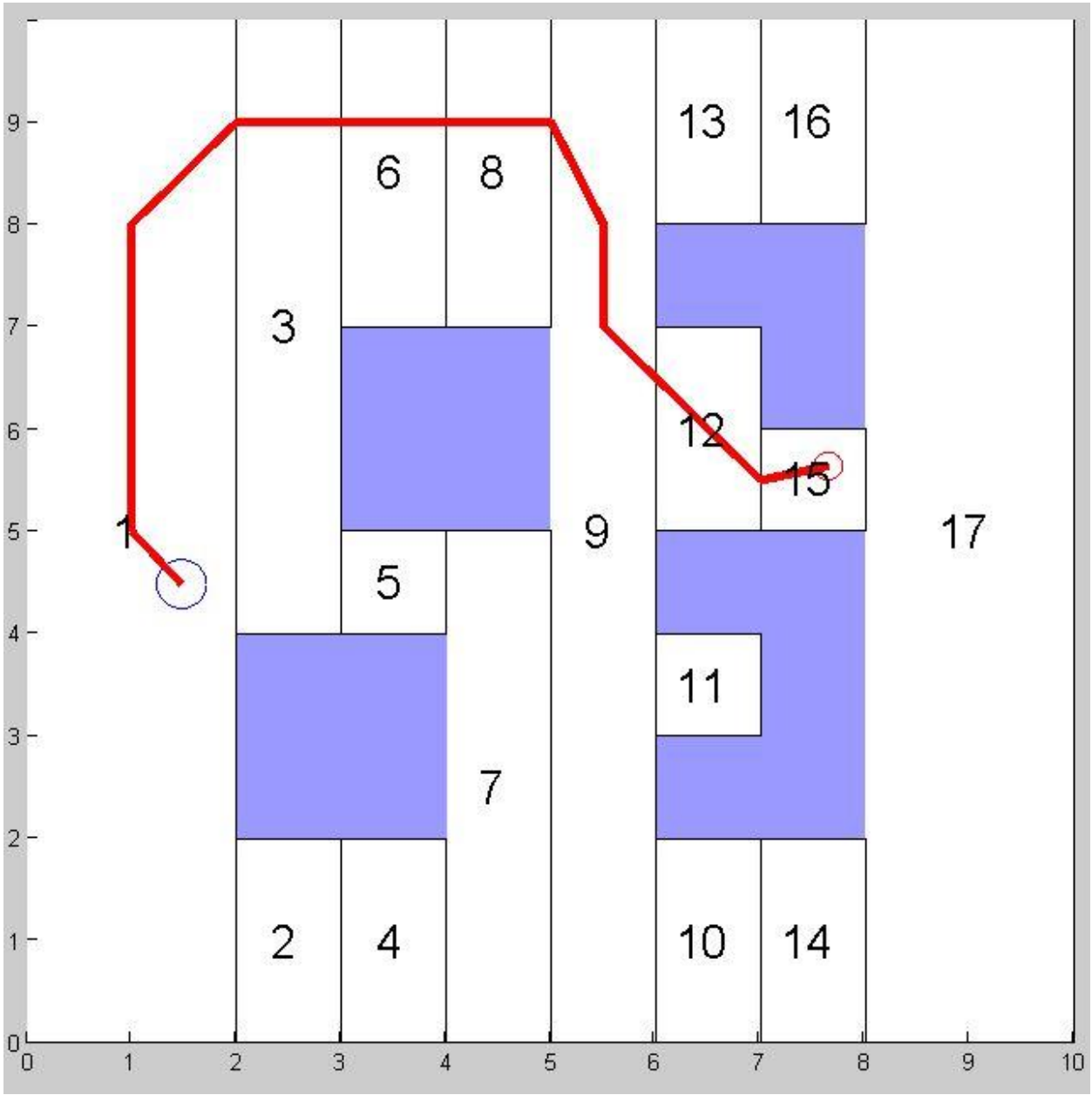

Figura 3-8 Simulación de Descomposición en Celdas v2

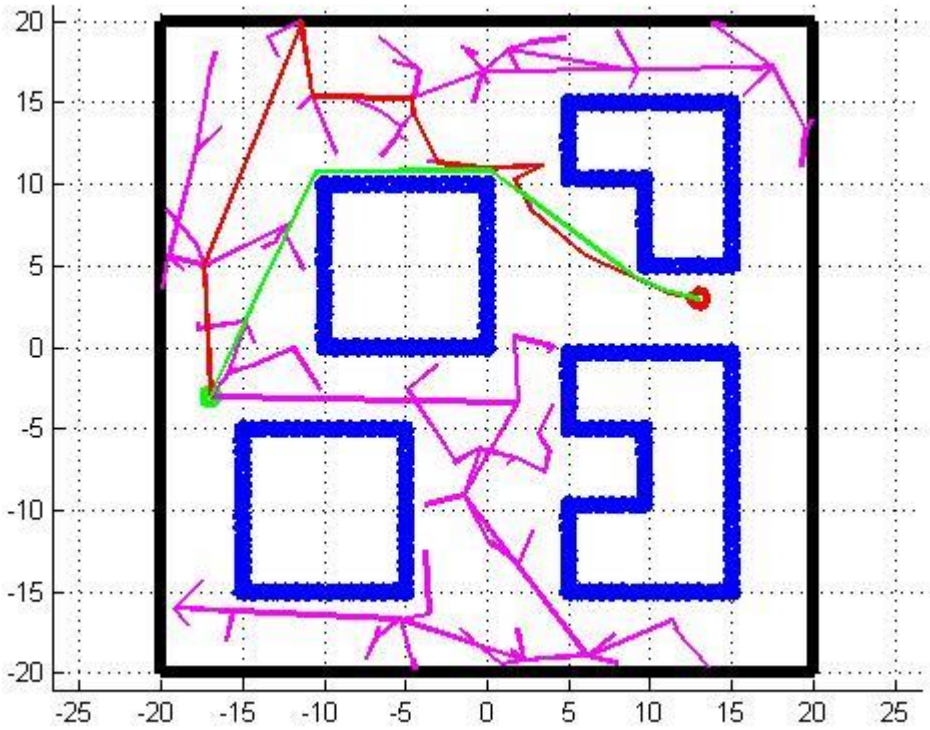

Figura 3-9 Simulación RRT (distancia 5) v2

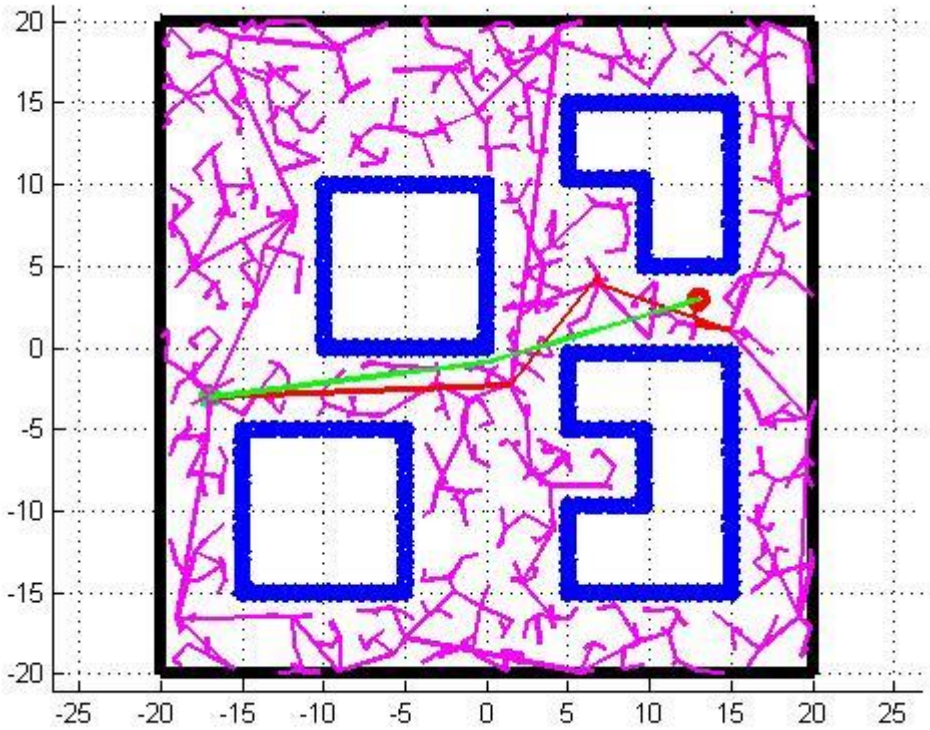

Figura 3-10 Simulación RRT (distancia 1) v2

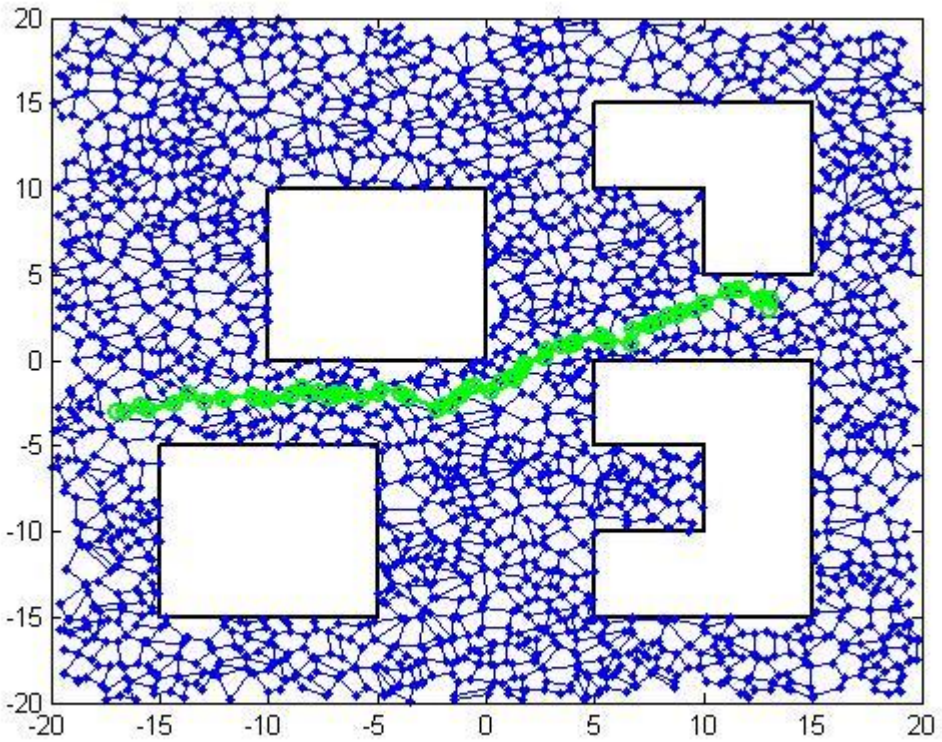

Figura 3-11 Simulación Voronoi (Simulación 1) v2

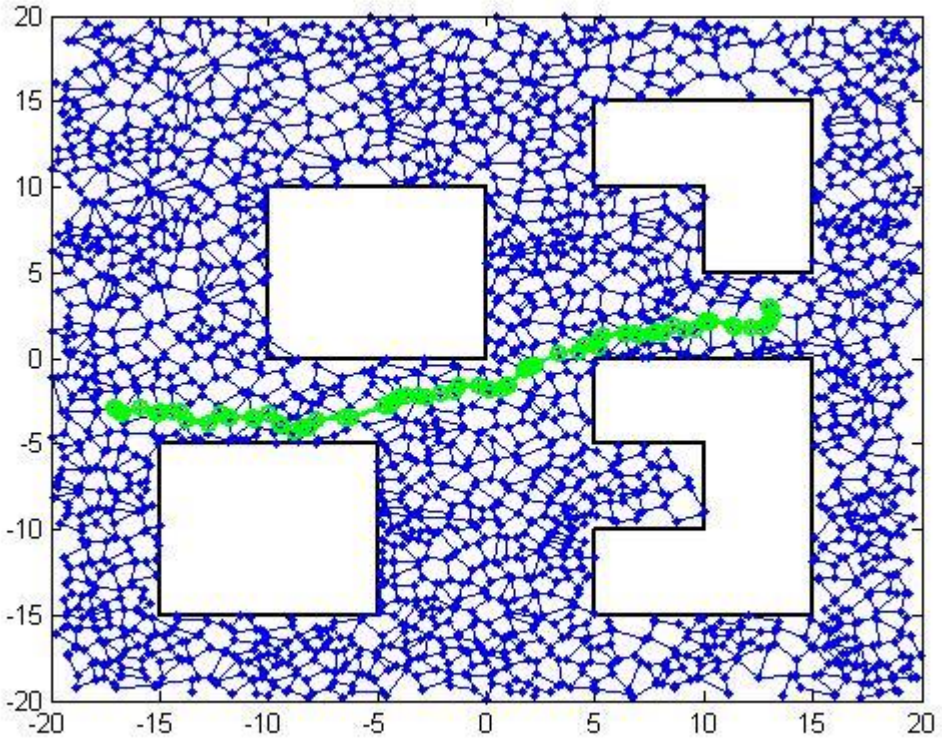

Figura 3-12 Simulación Voronoi (Simulación 2) v2

Para este caso podemos ver como coinciden el A* y el Voronoi como anteriormente pero también lo hace el RRT para distancia 1.

La explicación de por qué ambos RRT son distintos es porque a ser distinta la distancia el recorrido será distinto al tener menos opciones por el que realizar la trayectoria.

3.2 Comparaciones

Como se pudo observar en los casos anteriores hay diferentes formas de llegar al objetivo desde un mismo punto de partida. Para saber el por qué de esto explicaré cada algoritmo por separado y se verá cuál es mejor en cada caso.

El A* sigue un camino más directo puesto que este algoritmo sigue los nodos más cercanos al final que tienen menos costes. En caso de que dos nodos tengan el mismo coste se elige el último nodo que el algoritmo ha mirado (esto depende de como esté hecho el algoritmo, en mi caso es así).

En la descomposición de celdas no importa si la ruta seguida es la más larga o la más corta, sigue la trayectoria en el que el coste entre nodo y nodo sea menor.

Para el caso de RRT el problema es el coste computacional, cuanto mayor sea el mapa, más necesita para calcular la ruta a seguir. La ventaja que tiene es que si utiliza mucho coste computacional, a cambio consigue la ruta más corta para llegar.

Finalmente, con Voronoi y Dijkstra se consigue el camino más corto pero como calcula la ruta vértice a vértice quizás en una iteración siguiente uno de los vértices excluidos puede ser el de menor costo.

Para un mayor entendimiento a la hora de compararlos, se han tomado varias medidas de tiempo los cuáles muestran cuanto tardan en ejecutarse. El caso del RRT es especial y se mostrará por separado. Los valores son los siguientes:

Caso 1

Algoritmo	Tiempo 1 (s)	Tiempo 2 (s)	Tiempo 3 (s)	Media (s)
A*	0.03	0.03	0.04	**0.03**
Voronoi	1.58	1.72	1.64	**1.64**
Descomposición En Celdas	0.14	0.12	0.13	**0.13**

Tabla 4 Tabla de tiempos para Caso 1

Caso 2

Algoritmo	**Tiempo 1 (s)**	**Tiempo 2 (s)**	**Tiempo 3 (s)**	**Media**
A*	0.09	0.1	0.1	**0.1**
Voronoi	1.50	1.54	1.43	**1.49**
Descomposición En Celdas	0.12	0.13	0.13	**0.13**

Tabla 5 Tabla de tiempos para Caso 2

RRT

Como se dijo anteriormente, el caso de RRT es especial. Para una mejor aclaración se mostrarán los datos a continuación y se explicarán:

Caso 1		Caso 2	
Iteraciones	**Tiempo (s)**	**Iteraciones**	**Tiempo (s)**
733	32.40	447	25.45
502	28.37	120	12.04
1279	47.08	341	21.75

Tabla 6 Tabla de tiempos para ambos caso en RRT

Como se puede observar, los valores son bastante distintos entre ellos y se puede ver una clara relación entre el número de iteraciones con el tiempo de ejecución. Este es el motivo por el que no se puede comparar objetivamente con los otros casos pues depende de la simulación puede ser un tiempo u otro.

En todo caso, los tiempos aquí mostrados son bastante superiores a los casos anteriores; esto se debe a (cómo se explicó en el apartado del RRT) que el algoritmo es más eficiente cuanto más nodos haya y mayor sea el mapa en cuanto a los demás.

Una vez hechas estas rápidas explicaciones podríamos determinar lo siguiente:

- El algoritmo de menor costo es el de descomposición de celdas puesto que la ruta que sigue es siempre siguiendo los nodos que menos coste tienen.
- Tanto el A*, el Voronoi como el RRT (dependiendo de la exactitud que se busque en el algoritmo) el camino que toman es el más corto. Sin embargo hay peros.

  El A* tiene un problema y es que si el mapa es muy grande y tiene muchos obstáculos el cáculo computacional que necesita es enorme, entonces es muy buen método para zonas pequeñas y con pocos obstáculos.

  RRT tiene un problema similar pero en este caso se puede solucionar utilizando el RRT*, el cuál el coste es menor para mapas más grandes. A pesar de esto, este algoritmo es más eficiente que en los otros casos.

  El Voronoi depende del mapa, el cuál, al igual que los anteriores, si la región en la que el vehículo tiene que moverse es muy grande necesita hacer muchos cálculos entre vértices el cual puede ser demasiado grande.

  Estos problemas dependen mucho del caso en el que se utilicen o como de preciso es el algoritmo.
- El tiempo de cálculo de las trayectorias para los 4 algoritmos son diferentes. Dicho esto, podemos decir que para este caso, el más rápido es el algoritmo de A*. Sin embargo, el RRT depende de la ruta que siga al buscar el objetivo las iteraciones pueden ser muy pocas, por lo tanto el cálculo puede ser menor que otros casos o todo lo contrario, el que más tarda.
- El más fácil de implementar es el A*.
- Tomando todo esto en consideración podemos afirmar que la mejor opción en general es el RRT, puesto que es el mas eficiente en la mayoría de los casos, aunque como he dicho antes, depende del caso concreto en el que se trabaje es mejor uno u otro, todo depende de lo que se quiera buscar.

4 CONCLUSIONES Y DESARROLLOS FUTUROS

Este capítulo es el último. En el mostraré las conclusiones que he obtenido de este trabajo y los posibles desarrollos futuros que se podrían hacer a partir de aquí.

4.1 Conclusiones

A modo de resumen, se han estudiado diferentes métodos que se utilizan en los ámbitos de la róbotica móvil. Se ha visto diferentes simulaciones con las que poder entender estos algoritmos y entenderlos mejor. Con estas simulaciones y para el caso particular con los 4 obstáculos se llegó a la conclusión de cual era el mejor en cada caso, teniendo en cuenta las diferentes posibilidades (como el número de iteraciones en el RRT o el tamaño del mapa).

Todas estas simulaciones se han llevado a cabo en Matlab, el cuál, es bastante útil para un estudio teórico de los métodos y poder hacerse una idea de cuál es más conveniente para el caso que queremos utilizar alguno de los algoritmos.

4.2 Desarrollos Futuros

Como objetivos futuros para este trabajo pienso que una buena forma sería desarrollar estos algoritmos en otros lenguajes, como C. También poder desarrollarlos en ROS para ver como funciona en un ambiente distinto de Matlab.

Otra cosa podría ser aplicarlos en vehículos reales para ver su funcionamiento a través de obstáculos reales y ver si de verdad fuera de las simulaciones funcionan correctamente tal y como han mostrado los resultados de este trabajo.

Aunque anteriormente he dicho que estos algoritmos se emplean en robótica móvil, no es cierto del todo, puesto que también se usan en ámbitos como la róbotica aérea. Otra de las posibles mejores podría ser el estudio de estos métodos para casos en los que la robótica aérea trabaje.

5 BIBLIOGRAFÍA

[1] https://github.com/aearanky/Rapidly-Exploring-Random-Trees/blob/master/RRTpathplan.m

[2] https://es.wikipedia.org/wiki/Algoritmo_voraz

[3] https://es.wikipedia.org/wiki/Algoritmo_de_b%C3%BAsqueda_A*

[4] https://www.ecured.cu/Algoritmo_de_B%C3%BAsqueda_Heur%C3%ADstica_A*

[5] http://www.redalyc.org/articulo.oa?id=413940771004

[6] https://es.wikipedia.org/wiki/Polígonos_de_Thiessen

[7] https://en.wikipedia.org/wiki/Rapidly-exploring_random_tree

[8] http://msl.cs.uiuc.edu/~lavalle/papers/Lav98c.pdf

[9] http://ejournals.ukm.my/apjitm/article/view/4294

[10] http://journalarticle.ukm.my/6645/1/4294-9959-1-PB.pdf

[11] https://es.wikipedia.org/wiki/Algoritmo_de_Dijkstra

[12] http://math.mit.edu/~rothvoss/18.304.3PM/Presentations/1-Melissa.pdf

[13] https://brilliant.org/wiki/dijkstras-short-path-finder/