Trabajo Fín de Grado
Grado en Ingeniería de las Tecnologías Industriales

# Generación de trayectorias en tiempo real a partir de diagramas de Voronoi

Autor: Ángel Moya Carrasco
Tutores: José Antonio Cobano Suárez
Anibal Ollero Baturone

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

Sevilla, 2016

Trabajo Fín de Grado
Grado en Ingeniería de las Tecnologías Industriales

# Generación de trayectorias en tiempo real a partir de diagramas de Voronoi

Autor:
Ángel Moya Carrasco

Tutor:
José Antonio Cobano Suárez
Aníbal Ollero Baturone

Dep. de Ingeniería de Sistemas y Automática
Escuela Técnica Superior de Ingeniería
Universidad de Sevilla
Sevilla, 2016

Trabajo Fin de Grado: Generación de trayectorias en tiempo real a partir de diagramas de Voronoi

Autor: Ángel Moya Carrasco

Tutor: José Antonio Cobano
Aníbal Ollero Baturone

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

Presidente:

Vocales:

Secretario:

Acuerdan otorgarle la calificación de:

Sevilla, 2016

El Secretario del Tribunal

*A los que aún están*

*A los que se han ido*

# Agradecimientos

A mis padres, por todo el empeño que han tenido en que me esfuerce y de lo máximo de mí, a mi novia, por apoyarme en todos los momentos difíciles, y a mis tutores, tanto a D. Aníbal Ollero por enseñarme las bases de la robótica que han despertado mi interés en este ámbito, como a D. José Antonio Cobano que ha dedicado su tiempo y dedicación para orientarme en este proyecto.

# Resumen

Este trabajo aborda el problema de planificación local de trayectorias en entornos desconocidos. Para ello se utilizan diagramas de Voronoi para modelar el entorno percibido mediante un grafo permitiendo determinar las zonas más alejadas de los obstáculos detectados. Una vez generado el grafo, se hace uso del algoritmo Dijsktra para la búsqueda del camino más adecuado al destino, ya que es un algoritmo de gran sencillez para grafos con una cantidad de puntos reducida.

El desarrollo del proyecto está dividido en dos partes bien diferenciadas; la primera de estas consiste en la obtención de trayectorias para entornos conocidos utilizando la herramienta informática Matlab. En la segunda se hace uso de lo aprendido en la primera parte para realizar un sistema de evitación de obstáculos utilizando el Sistema Operativo Robótico (en inglés *Robot Operating System*, ROS) que es un framework para el desarrollo de software para robots que provee la funcionalidad de un sistema operativo. En este framework se ha utilizado el robot Turtlebot con una cámara Kinect a bordo en el entorno de simulación Gazebo. También se ha utilizado el Robotic System toolbox de Matlab para el conexionado con ROS en tiempo real.

# Abstract

This project addresses the problem of trajectories planning using a local path planning method for unknown environments. Voronoi diagrams are used as the main algorithm to generate a visibility graph. These diagrams permit determinate the zone in which the distance from a set of points is maximum. Then, the Dijsktra´s algorithm is used for the research of the shortest path from the origin to the goal because of its simplicity and good results for graph of a reduced quantity of points.

The development of the project is divided in two well differentiable parts; the first one consists on the computation of trajectories for known environments using the informatics tool Matlab. The second parts is based on the first one results by implementing an obstacles avoiding system in a reactive way using ROS (Robot Operating System), and Robotic Systems toolbox from Matlab in order to connect both programmes in real time. Simulations have been performed with a Turtleboot robot and a Kinect camera on board, tackling all the cases in the simulation environment tool Gazebo.

# Índice

# ÍNDICE DE FIGURAS

# Notación

| | |
|---|---|
| e | número e |
| π | número pi |
| Θ | Theta |
| α | Alfa |
| β | Beta |
| ⊆ | Contenido o igual |
| E | Perteneciente a |
| 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* |
| log | Función logaritmo en base 10 |
| : | Tal que |
| < | Menor o igual |
| > | Mayor o igual |
| \ | Backslash |
| ⇔ | Si y sólo si |
| min | Mínimo |
| max | Máximo |

# 1 INTRODUCCIÓN Y ALCANCE DEL PROYECTO

*Que extraño, cuanto más me esfuerzo mas suerte tengo.*

*- Henry Ford -*

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 es la mejora en la calidad de vida de las personas, que se logra gracias a la liberación de funciones del ser humano en su propio hogar, con aspiradoras autónomas por ejemplo, hasta el transporte de productos en naves de fabricación industrial de forma autónoma con sistemas de AGVs, o de posicionamiento por balizas.

En este proyecto se propone una forma de resolver estos problemas que pueden presentarse en diversos campos, para poder aportar nuevas ideas en el campo de la autonomía de robots, y la planificación de trayectorias en entornos desconocidos.

## 1.1. Objetivos y motivaciones del Proyecto

A la hora de generar un camino o trayectoria segura que seguir en entornos desconocidos, debemos recoger información del entorno a partir de sensores para poder solucionar el problema, es decir, posibles colisiones con obstáculos. También en necesario localizar nuestro vehiculo en el entorno. Esto require importantes esfueros donde en un primer instante no conocemos más que el estado dinámico del robot, a través del estado de sus sensores internos.

Para obtener información del medio, son necesarios ciertos sensores de detección de objetos. La precisión de estos sensores está muy condicionada por el presupuesto y las condiciones de nuestro robot. Este proyecto considera un enfoque económico que pueda resolver el problema para pequeños robots que puedan ser programados para probar la fiabilidad del método de forma real, por lo que se utilizará el sensor de una pequeña cámara Kinect como detector de objetos. Esta es una cámara muy utilizada en entornos de investigación y académicos por su gran flexibilidad y reducido precio, otorgando una lectura de obstáculos en 2D en un rango de distancia de entre 0.5 y 5 metros, y un rango angular de 57 grados.

La generación de trayectorias requiere un primer paso en el que se detectan los obstáculos. Una vez detectados, deberemos modelar la forma de los obstáculos para poder saber con precisión la posición de los obstáculos.

Un requisito fundamental del Sistema de generación de trayectorias será la presencia de una alta eficiencia, pues

la simplicidad de computación permitirá unos buenos resultados ya que al ser un Sistema que estará trabajando a tiempo real debe tener una capacidad de reacción elevada. Además una menor carga computacional permitirá abaratar costes para unas futuras implementaciones en robots reales.

El fin del proyecto es servir como antecedente para implementar el Sistema de generación de trayectorias que se va a desarrollar en aplicaciones reales de robots, utilizando únicamente un Sistema de posicionamiento como pueda ser un GPS, o el posicionamiento por balizas, y un simple sensor de detección de obstáculos para poder recorrer trayectorias seguras en entornos desconocidos. Se trabajará con el entorno de programación de robots ROS (Robot Operating System), herramienta ampliamente utilizada en el entorno robótico, con la que implementar el sistema aquí desarrollado se convierte en algo trivial si se dispone del hardware adecuado.

## 1.2. Estructura del Proyecto

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

- Método propuesto (Capítulo 3): En esta parte se explican los algoritmos elegidos para desarrollar las dos partes fundamentales de la generación de trayectorias: la generación del grafo de puntos conectados; y la generación del camino más adecuado en función de este grafo.
- Implementación (Capítulo 4): Esta es la parte principal del proyecto, en la que se explican todas las simulaciones, desde las pruebas iniciales para entornos conocidos en la herramienta Matlab, hasta el paso al sistema ROS, donde también se establecerá una comunicación con Matlab a través del *Robotic System Toolbox* incluido en la última versión de Matlab para permitir la comunicación entre ambas plataformas.
- Experimentación (Capítulo 5): En esta parte se mostrarán algunos ejemplo del resultado obtenido, para lo que se utiliza la herramienta de ROS de simulación de robots Gazebo, en la que se puede representar de forma muy precisa la respuesta del robot, incluyendo factores de deslizamiento, errores de posicionamiento, y toda la dinámica y características físicas del vehículo. Todas las pruebas engloban los resultados de todo el desarrollo del proyecto, por lo que es una parte fundamental del mismo. Además muestra la respuesta de todo el sistema y permite observar donde se encuentran los puntos fuertes y los defectos del sistema.
- Conclusiones y posibles trabajos futuros (Capítulo 5): En esta parte se resumen todas las impresiones que ha generado la realización del proyecto, las conclusiones sacadas de todo el desarrollo, las posibles mejoras, modificaciones, e implementaciones que pueden realizarse en vistas a un posible Trabajo Fin de Master o investigaciones futuras para sistemas reales.

# 2 ESTADO DEL ARTE

*El fracaso es, a veces, más fructífero que el éxito.*

*- Henry Ford -*

La planificación de trayectorias o caminos eficientes y seguros para aplicaciones con robots móviles autónomos juega un papel fundamental en todos las funcionalidades que se están abordando en estos últimos años. Concretamente, los robots deben calcular sus trayectorias de manera eficiente para llevar a cabo las misiones de forma segura a partir de la información sobre el entorno dado por los sensores de abordo.

Una clasificación de algoritmos de planificación y evitación de colisiones es presentada en [1] y [15] respectivamente. Entre los más utilizados en la literatura destacan los algoritmos evolutivos como genéticos [2], optimización de enjambres de partículas (Particle Swarm Optimization) [7] o basados en fuegos artificiales (Fireworks Algorithms) [13], métodos de programación no-lineal [3], mixed-integer linear programming [4][5], métodos de colocación [6], algoritmos de planificación RRT (Rapidly-Exploring Random Trees) [8], RRT* [14], cocido simulado (Simulated Annealing) [9], colonias de hormigas (Ant Colony) [10] y otros.

La representación del entorno es fundamental en cualquier algoritmo de planificación de caminos. Atendiendo a este requisito, los algoritmos se pueden dividir en: utilización de cuadrículas (grids en inglés) uniformes o no-uniformes, una descomposición exacta de celdas, roadmaps probabilísticos, grafos de visibilidad o diagramas de Voronoi. Estos dos últimos se pueden englobar en los métodos de roadmaps.

En cuanto a los métodos que representan el entorno mediante cuadrículas, se distinguen los que trabajan con grids uniformes, y no uniformes. Las cuadrículas uniformes dividen el entorno en una serie de celdas con las mismas dimensiones mientras que las no uniformes dividen el entorno en celdas cuyo tamaño es variable.

La descomposición exacta de celdas suele utilizar la descomposición trapezoidal. Este método consiste en dividir el entorno libre de obstáculos en celdas, de forma que éstas representen el espacio seguro. La forma y posición de las celdas, por tanto, vienen determinadas por la posición de los obstáculos. El uso de la descomposición trapezoidal viene determinado por posición de los obstáculos que puede dar lugar a celdas trapezoidales, que representan el espacio libre.

Los métodos basados en roadmaps convierten el entorno multidimensional en una red de curvas unidimensionales, que se encuentran en una zona libre de obstáculos. A estas curvas se les denomina roadmaps.

Los roadmaps probabilísticos buscan crear un mapa generado de forma aleatoria y libre de colisiones, de forma que conecte de forma rápida la posición del robot con el objetivo.

Los métodos basados en roadmaps más utilizados son los grafos de visibilidad y los diagramas de Voronoi. En los grafos de visibilidad, se modela el entorno y obstáculos mediante vértices que se unen con el resto de vértices mediante líneas rectas y comprueban si se intercepta algún obstáculo o no. Esto equivale a comprobar qué vértices son visibles desde un determinado vértice. Una vez repetido este proceso para cada vértice, se obtiene una red de líneas o grafo que unen los distintos vértices cuya conexión no intercepta a ningún obstáculo. Una vez generado el grafo de visibilidad, un algoritmo de búsqueda generará los vértices que conforman la secuencia a seguir para conectar el punto inicial y el punto final de forma óptima. Con este método el camino solución suele pasa muy cerca de los obstáculos, ya que los puntos de paso del camino serán vértices de los obstáculos aumentado para considerar un margen de seguridad.

Por otro lado, los diagramas de Voronoi también generan un grafo pero el objetivo en este caso es separarse lo máximo posible de los obstáculos. Dado un conjunto de obstáculos en el plano, el diagrama de Voronoi es el lugar geométrico de los puntos del plano que equidista de los dos obstáculos más cercanos en todo momento. Por tanto, en este caso prima ante todo la seguridad.

Existen varios métodos para generar los diagramas de Voronoi. Se pueden construir en espacios continuos [11] [12] o discretos, por ejemplo grids. En navegación de robots móviles se suelen usar representaciones basadas en cuadrículas (grids) y los diagramas de Voronoi se generan sobre un conjunto finito de celdas.

Tras generar un diagrama de Voronoi, los algoritmos de planificación deben calcular el camino para unir un punto inicial y final. El diagrama de Voronoi proporciona un grafo sobre el un algoritmo de búsqueda debe calcular el camino más corto. Entre los algoritmos de búsqueda se destacan los siguientes: Dijkstra, Bellman-Ford, A*, D*, Búsqueda en profundidad (Deep-First Search), Búsqueda en amplitud (Breadth-First Search).

En este trabajo se utiliza un Dijkstra a partir de un grafo definido por nodos o vértices y aristas que unen estos nodos y que poseen pesos (coste que supone ir de un nodo a otro). El algoritmo de Dijkstra evalúa, desde el nodo inicial, el coste invertido en ir a cada uno de los nodos adyacentes, y se desplaza al que menor coste acumulado suponga, marcándolo como permanente y repitiendo el proceso a partir del mismo. De esta forma, en cada paso se desplaza al nodo adyacente que presente menor coste acumulado, si no se ha marcado previamente como permanente. Cuando un nodo ha sido marcado como permanente, quiere decir que se ha encontrado el camino de menor coste desde el nodo inicial hasta dicho nodo. De esta forma, el algoritmo de Dijkstra es capaz de encontrar el camino de menor coste desde el nodo inicial hasta cada uno de los nodos del grafo.

# 3 MÉTODO PROPUESTO

*Que el futuro diga la verdad y evalúe a cada uno según su trabajo y sus logros.*

*- Nikola Tesla -*

El desarrollo del proyecto gira en torno al algoritmo Voronoi, ampliamente utilizado en una gran variedad de campos del conocimiento, como son la Topología, Robótica, Diseño Gráfico, Ordenación Forestal, etc. La gran cantidad de campos en que este diagrama es utilizado no es casualidad, y es que permite obtener a partir de un mapa de representación regiones, las cuales constituyen una por una un lugar geométrico en el que la distancia al punto incluido en esa sección es menor que la distancia al resto de puntos.

Se ha elegido este diagrama para el sistema a implementar ya que es de gran interés las fronteras entre las regiones, pues determinan la zona más alejada de los puntos del mapa y por tanto, la seguridad.

Considerando el al diagrama de Voronoi obtenido, se podrá utilizar el algoritmo Dijsktra para calcular las trayectorias . Este algoritmo goza de una gran eficiencia para grafos de puntos reducidos, como será el caso a tratar.

## 3.2 Grafo geométrico

En matemáticas y ciencias de la computación, un grafo (del griego grafos: dibujo, imagen)[16] es un conjunto *V* de vértices o nodos unidos por enlaces llamados aristas o arcos, que forman el conjunto *E*, que permiten representar relaciones binarias entre elementos de un conjunto.

Típicamente, un grafo se representa gráficamente como un conjunto de puntos (vértices o nodos) unidos por líneas rectas o curvas (aristas), aunque existe todo un campo de investigación relacionado con los diversos modos de representación como se ha descrito en el capítulo 2.

Los grafos permiten estudiar las interrelaciones entre unidades que interactúan unas con otras. Por ejemplo, una red de distribución de mercancías puede representarse mediante un grafo, en el cual los vértices representan puntos de recogida y/o llegada de mercancía estableciéndose relación entre los puntos en función del coste del camino por ejemplo. Prácticamente cualquier problema puede representarse mediante un grafo, y su estudio trasciende a las diversas áreas de las ciencias exactas y las ciencias sociales.

Llamamos grafo dirigido al grafo G = (V, E) cuyas aristas son un conjunto de pares ordenados de elementos de *V*. Dada una arista (*a, b*), *a* es su nodo inicial y *b* su nodo final. Terminología de los grafos:

- Adyacencia: dos aristas son adyacentes si tienen un vértice en común, y dos vértices son adyacentes si una arista los une.
- Incidencia: una arista es incidente a un vértice si este es uno de sus extremos.
- Ponderación: función que a cada arista le asocia un valor (coste, peso, longitud, etc.), para aumentar la expresividad del modelo.
- Etiquetado: distinción que se hace a los vértices y/o aristas mediante una marca que los hace unívocamente distinguibles del resto.

En matemáticas, un grafo geométrico es un grafo en cuyos vértices o aristas están asociadas a objetos o configuraciones geométricas. Un grafo euclídeo concretamente es un grafo cuyos vértices representan puntos del plano, y a sus aristas se les asignan pesos iguales a la distancia euclídea entre sus dos vértices.

Los grafos pueden ser clasificados de muchas formas, pero a conveniencia del enfoque que va a tener para el proyecto se pueden clasificar en dos conjuntos en función de la reciprocidad o no de las conexiones entre dos mismos grafos.

### 3.2.1 Grafos dirigidos o dígrados

Son aquellos en los cuales el conjunto de los vértices tiene una dirección definida. A veces un digrafo es denominado digrafo simple para distinguirlo del caso general del multigrafo dirigido, donde los arcos constituyen un multiconjunto, en lugar de un conjunto. En este caso, puede haber más de un arco que una dos vértices en la misma dirección, distinguiéndose entre sí por su identidad, por su tipo (por ejemplo un tipo de arco representa relaciones de amistad mientras que el otro tipo representa mensajes enviados recientemente entre los nodos), o por un atributo como por ejemplo su importancia o peso. Formalmente podría definirse un grafo dirigido $G$ tal que:

$$V \neq 0$$

$$E \subseteq \{(a,b) \in V\ x\ V : a \neq b\}$$


A menudo también se considera que un digrafo simple es aquél en el que no están permitidos los bucles. Un bucle es un arco que une un vértice consigo mismo.

### 3.2.2 Grafos no dirigidos

Sea *G(V,E)* un grafo no dirigido, se establece que:

$$V \neq 0$$

$$E \subseteq \{x \in P(V) : \lceil x \rceil = 2\}$$


### 3.2.3 Grafos mixtos

Aquellos grados que se definen con la capacidad de poder contener aristas tanto dirigidas como no dirigidas.

## 3.3 Diagrama de Voronoi

Los diagramas de Voronoi[17], que toman su nombre del matemático ruso Guergui Voronoi, también conocidos como polígonos de Thiessen, en honor al meteorólogo estadounidense Alfred H. Thiessen, son una construcción geométrica que permite realizar una partición del espacio euclídeo. Es uno de los métodos más utilizados para obtener grafos de proximidad.

Constituye uno de los métodos de interpolación más simples y uno de los más extendidos debido a su sencillez, basándose en la distancia euclídea. Los polígonos generados se obtienen al trazar las mediatrices de los segmentos que unan dos puntos entre sí. La intersección de estas mediatrices generará 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.

### 3.3.1 Definición Formal

En primer lugar consideramos la distancia euclídea entre dos puntos $p$ y $q$ en el plano como:

$$\|p - q\| = \sqrt{(p_x - q_x)^2 + (p_y - q_y)^2}$$

Definimos ahora el conjunto de puntos en el plano:

$$P = p_1, p_2, \ldots, p_n$$

Siendo $n$ el número de puntos distintos en el plano, así, se define el diagrama de Voronoi de $P$ como la subdivisión del plano en $n$ regiones, definiéndose una región para cada $p_i$ ε P, cumpliendo la propiedad de proximidad en la que un punto $q$ pertenece a la región de un sitio $p_i$ si y solo sí:

$$\|q - p_i\| = \|q - p_j\| \; para\, cada \; p_j \in P, j \neq i$$

Denotándose al diagrama de Voronoi de $P$ mediante *Vor(P)*. Cada región que corresponde a un sitio $p_i$ se denotará como $\gamma(p_i)$ y será llamada región de Voronoi de $p_i$.

La región de Voronoi para un sitio $p_i$ está construida a partir de las intersecciones de los semiplanos formados al trazar los bisectores de $p_i$ hacia los sitios $p_j$, $j \neq i$. Tomando el caso donde únicamente hay dos sitios $p$ y $q$, se traza el segmento de recta de $pq$ y posteriormente se traza el bisector de $pq$. Este bisector parte el plano en dos semiplanos, donde el semiplano que contiene a $p$ (representado como $h(p,q)$) representa el lugar geométrico de todo los puntos más cercanos a $p$ que a $q$; y el semiplano que contiene a $q$ ($h(q,p)$) alberga a todos los puntos más próximos a $q$ que a $p$. De acuerdo con esto, se puede establecer de forma general cómo se define la región

de Voronoi para un sitio $p_i$:

$$\gamma(p_i) = \bigcap_{j=1, j\neq i}^{n} h(p_i, p_j)$$

$\gamma(p_i)$ 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 como máximo *n-1* vértices y como máximo *n-1* aristas. Se tienen estas cantidades de vértices y aristas debido a que los sitios más lejanos suelen tener asociadas regiones de Voronoi no acotadas conformadas por aristas y semi-aristas (aristas que tienen un vértice de inicio pero no uno de final).

La Figura 3-1 muestra un simple ejemplo con 3 puntos donde las líneas rojas constituyen los segmentos que subdividen el plano euclídeo:

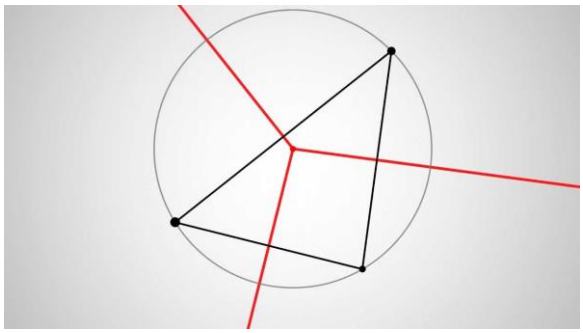

Figura 3-1. Divide y vencerás

### 3.3.2 Algoritmos para la construcción del diagrama de Voronoi

#### 3.3.2.1 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 $p_i \varepsilon P$ 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 $\gamma(p_i)$.

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:

$$O(n^2 \log n)$$

#### 3.3.2.2 Algoritmo divide y vencerás

Esta estrategia constituye un poderoso paradigma para definir algoritmos eficientes. 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 $1 \leq k \leq n$), cada uno con una entrada de tamaño *n/k* y donde $0 \leq n/k < n$. 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 $P_1$ y $P_2$, con aproximadamente el mismo tamaño, de los que se debe encontrar su diagrama de Voronoi independientemente. Finalmente, $Vor(P_1)$ y $Vor(P_2)$ deben ser unidos para poder obtener $Vor(P)$.

Dada una petición $P_1$ $P_2$ de *P*, sea $\sigma(P_1,P_2)$ el conjunto de aristas de Voronoi que son compartidas por pares de regiones de Voronoi $\gamma(p_i \varepsilon P_1)$ y $\gamma(p_i \varepsilon P_2)$. La colección de aristas $\sigma(P_1,P_2)$ es el conjunto de aristas de una subgráfica de $Vor(P)$ con las siguientes propiedades:

- $\sigma(P_1,P_2)$ 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 $P_1$ y $P_2$ 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), $\sigma(P_1,P_2)$ 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 $\sigma$ parte al plano en una porción izquierda $\pi_L$ y una porción derecha $\pi_R$. Con base en esto, se tiene la siguiente propiedad:

Si $P_1$ y $P_2$ son linealmente separados por una línea vertical con $P_1$ a la izquierda y $P_2$ a la derecha, entonces el diagrama de Voronoi Vor(P) es la unión de $Vor(P_1) \cap \pi_L$ y $Vor(P_2) \cap \pi_R$.

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

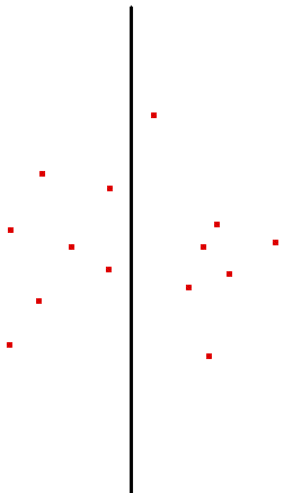

Figura 3-2. Divide y vencerás 1

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

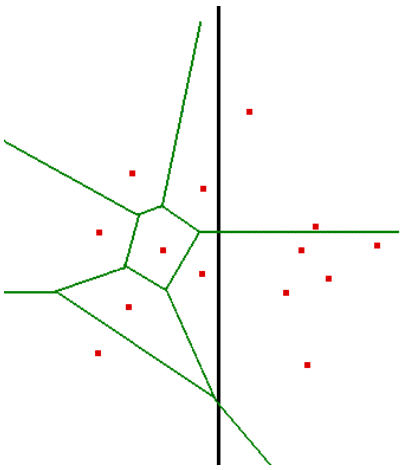

Figura 3-3. Divide y vencerás 2

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

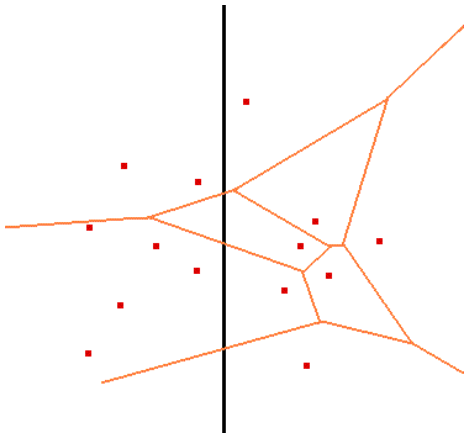

Figura 3-4. Divide y vencerás 3

4. Paso 4: Se calcula la cadena divisoria

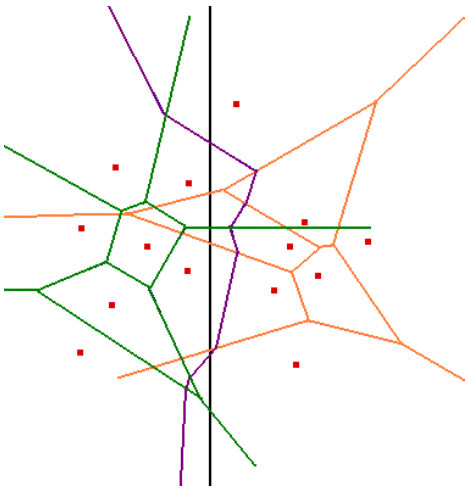

Figura 3-5. Divide y vencerás 4

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

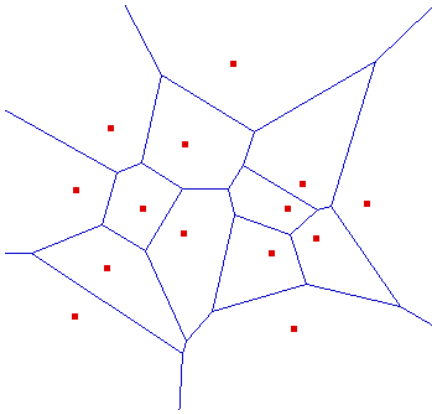

Figura 3-6. Divide y vencerás 5

#### 3.3.2.3 Algoritmo de Fortune (barrido de recta)

El algoritmo de fuerza bruta previamente revisado tiene una complejidad $O(n^2 \log n)$ , sin embargo, teniendo en cuenta la propiedad siguiente:

*Para $n \geq 3$, 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:

$$O(n \log n)$$

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.

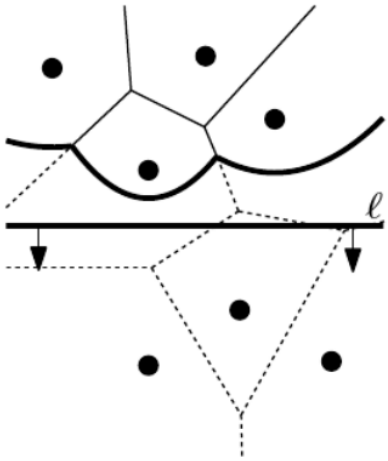


Figura 3-7. Fortune (Barrido recta)

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:

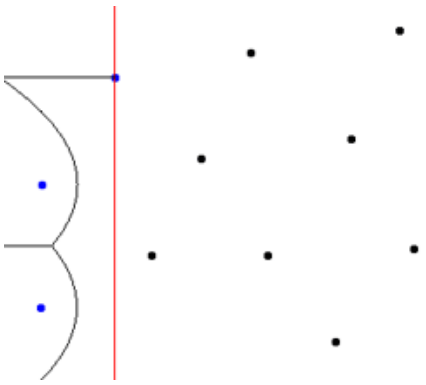

Figura 3-8. Fortune Progreso 1

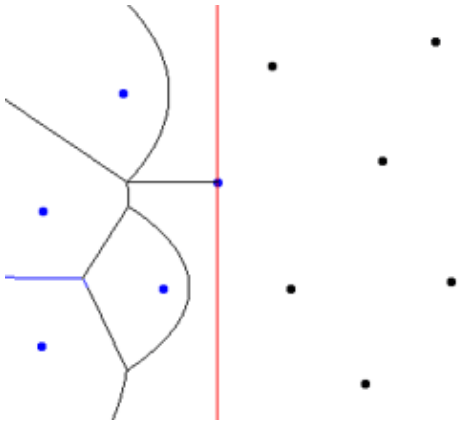

Figura 3-9. Fortune Progreso 2

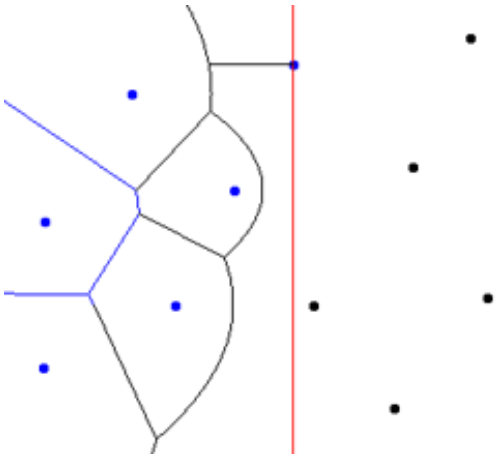

Figura 3-10. Fortune Progreso 3

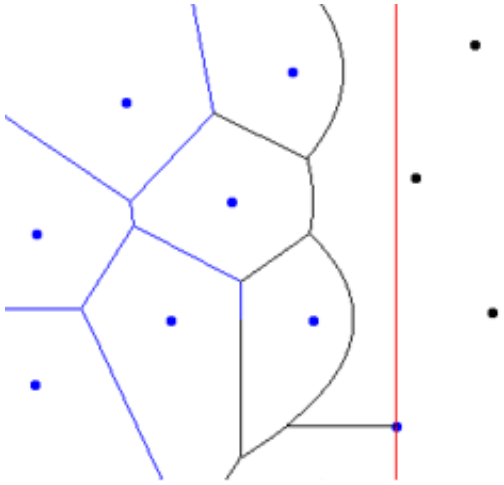

Figura 3-11. Fortune Progreso 4

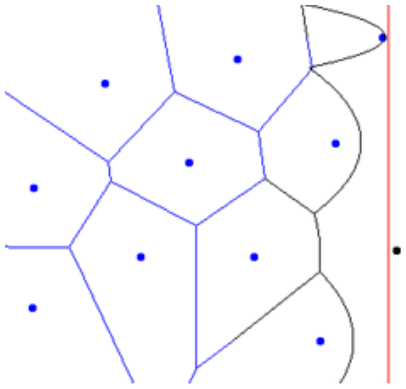

Figura 3-12. Fortune Progreso 5

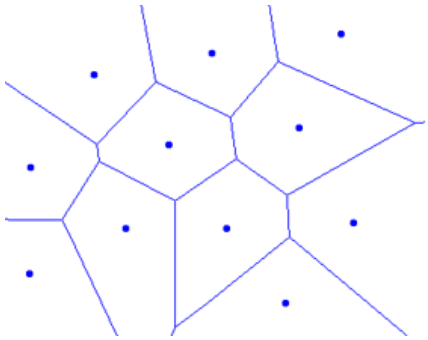

Figura 3-13. Fortune Progreso 6

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.

## 3.4 Algoritmo Dijkstra

Es un algoritmo de búsqueda en grafos para la determinación del camino más corto dado un vértice origen *f* al resto de vértices en un grafo con pesos en cada arista. Cobra su nombre en honor a su descubridor Edsger Dijkstra. La idea de 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.

De manera iterativa se construye un conjunto de vértices seleccionados *S* que se toman del conjunto de vértices candidatos *C* según el menor peso (distancia) desde *f*. El algoritmo termina cuando no hay más vértices de *G* fuera del conjunto formado. El paradigma usado corresponde al método voraz, en el que se trata de optimizar una función sobre una colección de objetos (menor peso).

Se usa un vector *d[v]* para almacenar la distancia de *v* a *f*, cuando se añade un vértice al conjunto *S*, el valor de *d[v]* contiene la distancia de *f* a *v*. Cuando se añade un nuevo vértice *u* al conjunto *S*, es necesario comprobar si

*u* es una mejor ruta para sus vértices adyacentes *z* que están en el conjunto *C*. Para ello se actualiza *d* con la relajación de la arista *(u,z)*:

$$d[z] = \min(d[z], d[u] + w[u, z])$$

En forma de pseudocódigo, el desarrollo del algoritmo queda:

```
Dijkstra(G, f)
S = { f }, C = { V }
d[f] = 0
d[u] = ∞          ∀u ≠ f
while (C ≠ ∅) {
  seleccionar vértice w ∈ C / d[w] es mínimo
  S = S ∪ {w}, C = C - {w}
  for cada vertex v ∈ Adyacente[w] {
    if (d[v] > d[w] + w(w, v))
        d[v] = d[w] + w(w, v)
  }
}
```

Figura 3-14. Dijkstra pseudocódigo

El método gráfico es el más adecuado para comprender como trabaja este algoritmo, así, para un simple grafo de 5 puntos dirigido podemos ver el desarrollo del algoritmo:

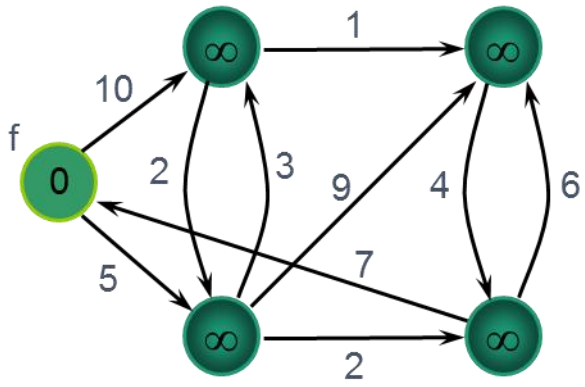


Figura 3-15. Dijkstra ejemplo 1.1

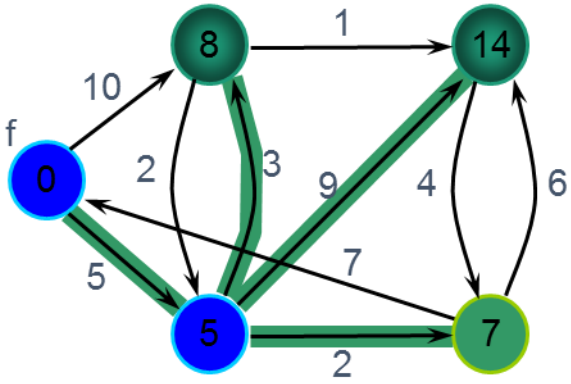


Figura 3-16. Dijkstra ejemplo 1.2

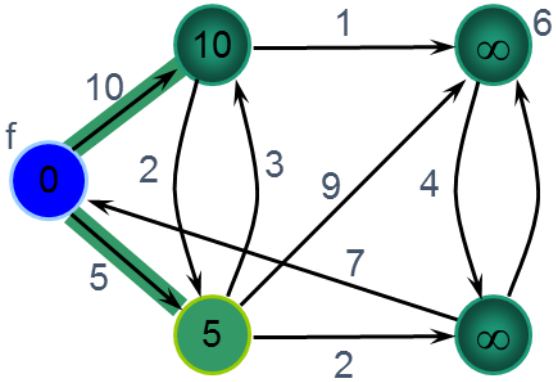


Figura 3-17. Dijkstra ejemplo 1.3

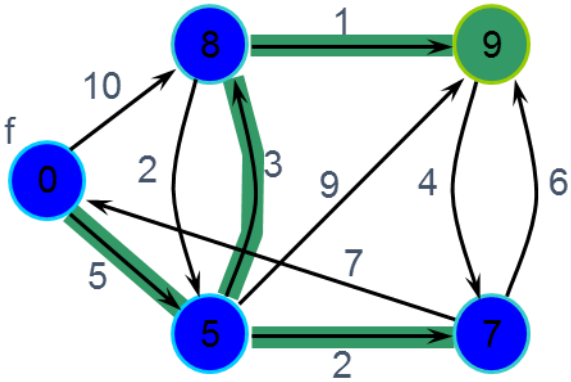


Figura 3-18. Dijkstra ejemplo 1.4

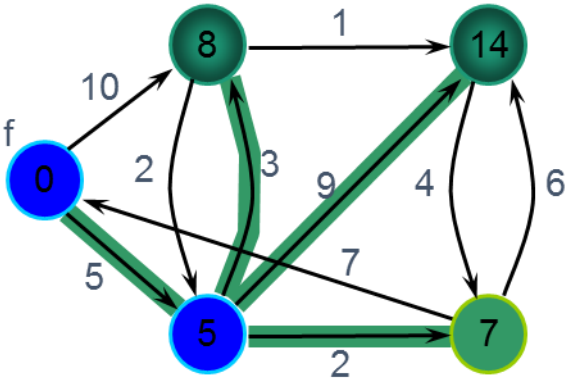


Figura 3-19. Dijkstra ejemplo 1.5

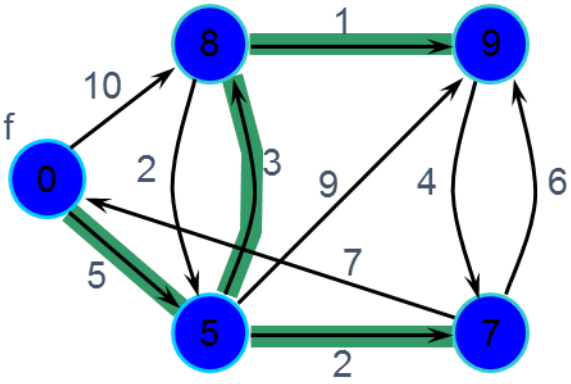


Figura 3-20. Dijkstra ejemplo 1.6

Para reconstruir los vértices del camino más corto desde el vértice fuente a cada vértice se usa otro array *p[v]* que contiene el vértice anterior a *v* en el camino se inicializa $p[v] = f$ para todo $v \neq f$, se actualiza p siempre que d[*v*] > d[*w*] + w(*w*, *v*)

El camino a cada vértice se halla mediante una traza hacia atrás en el array *p*.

# 4 IMPLEMENTACIÓN

*La imaginación es más importante que el conocimiento.*

*- Albert Einstein -*

El objetivo del proyecto es ser capaz de implementar un algoritmo que funcione en tiempo real para esquivar obstáculos cuyas ubicaciones son en principio desconocidas, hasta llegar a un destino especificado. La forma de proceder constará de varias etapas para poder ir comprendiendo bien cómo funcionan los algoritmos que queremos utilizar, y sobre todo comprobar los tiempos de computación de estos para ver si pueden ser usados en tiempo real y cómo modificarlos para reducir el tiempo de cálculo al mínimo.

Para generar el grafo utilizaremos los diagramas de Voronoi. La idea del diagrama de Voronoi se basa fundamentalmente en la proximidad. Dado un conjunto finito de puntos en el plano $P = \{p_1,...,p_n\}$ (con $n$ mayor o igual que dos) y a cada $p_j$ le asociamos aquellos puntos del plano que están más cerca o igual suya que de cualquier otro de los $p_i$ con $i$ distinto de $j$. Todo punto del plano queda así asociado a algún $p_i$, formándose conjuntos que recubren a éste. Existirán puntos que disten lo mismo de dos elementos de $P$ y que formarán la frontera de cada región. Los conjuntos resultantes forman una teselación del plano, en el sentido de que son exhaustivos (todo punto del plano pertenece a alguno de ellos) y mutuamente excluyentes salvo en su frontera.

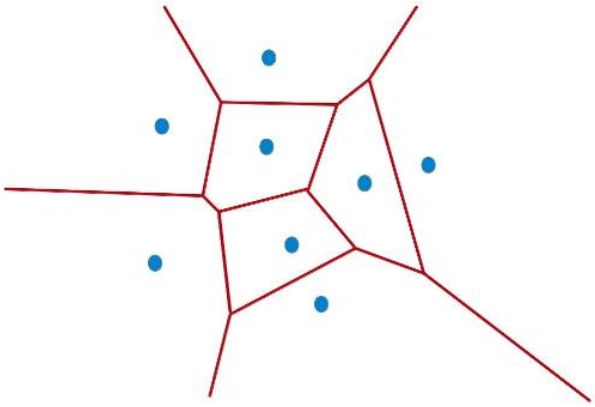

Figura 4-1. Diagrama Voronoi

Para nuestra aplicación, donde se fomenta la seguridad, lo que verdaderamente nos importará serán estas fronteras entre puntos obstáculos ya que serán las trayectorias más alejadas de estos. Dentro de los distinto algoritmos para obtener el diagrama de Voronoi utilizaremos el de Fortune, que es más sencillo de implementar y el más eficiente. Utilizaremos la herramienta Matlab, que implementa una extensa biblioteca de útiles funciones para nuestro propósito entre las que se encuentra la función de Voronoi, para poder comprender bien como trabaja este algoritmo, y el algoritmo Dijkstra, que será el utilizado para la búsqueda del camino más corto a partir del grafo.

Como simulaciones preliminares, vamos a realizar una primera fase del proyecto en el que trabajaremos con mapas conocidos *apriori* para comprobar la efectividad de lo que se quiere realizar, y el modo de hacerlo.

## 4.1 Generación de grafos y trayectorias estáticas

Para generar nuestro diagrama de Voronoi utilizaremos la función "Voronoi" de la biblioteca de Matlab, la cual nos generará dos matrices de pares de puntos que determinarán los segmentos que forman el grafo inicial. Este grafo tendrá que ser modificado en varios aspectos que se explicarán más adelante.

### 4.1.1 Elección de puntos obstáculo

Los primeros puntos obstáculos que vamos a ir almacenando en un vector serán los puntos esquina de los rectángulos, por ser puntos singulares y que claramente modelarán de buena forma el contorno del rectángulo. Para ello repasaremos todos los puntos obstáculo de nuestra matriz, comprobando sus puntos adyacentes, con lo que fácilmente podemos determinar:

- Punto esquina exterior: 5 puntos libres adyacentes.
- Punto esquina interior: 1 punto libre adyacente.
- Punto línea de borde: 3 puntos libres adyacentes.

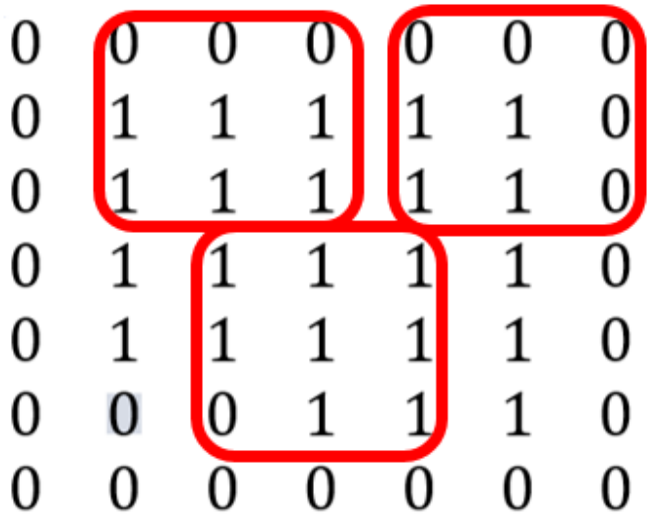


Figura 4-2. Modelado obstáculos

En las diferentes pruebas realizadas se observaba una falta importante de precisión y suavidad en el resultado

del grafo, por lo que consideré útil realizar un duplicado de puntos obstáculo, que consiste en añadir puntos obstáculo en la dirección vertical y horizontal de cada punto que tenemos en el instante en el que esta dirección vertical u horizontal se encuentre con un borde de la matriz o con un rectángulo, mejorando de esta forma la suavidad del grafo:

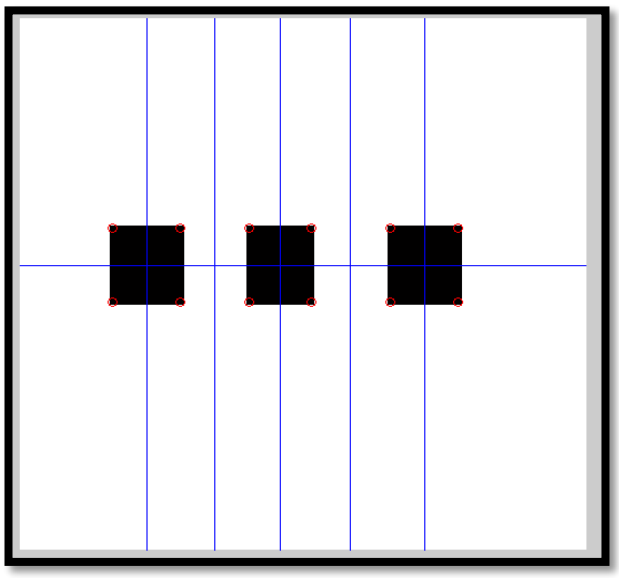

Figura 4-3. Sin duplicado 1

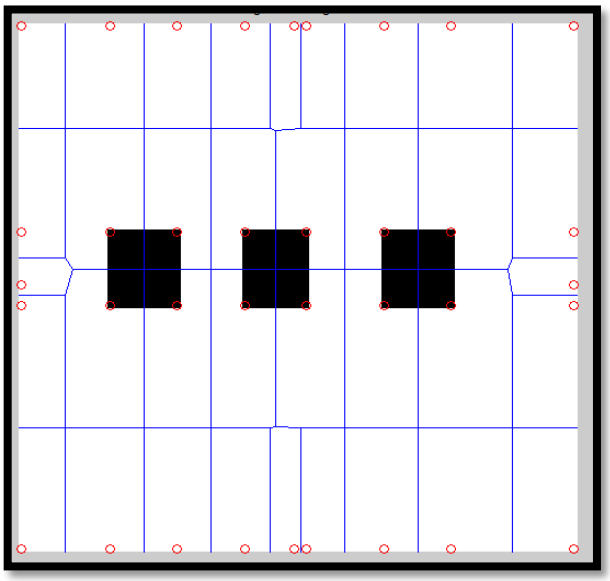

Figura 4-4. Con duplicado 1

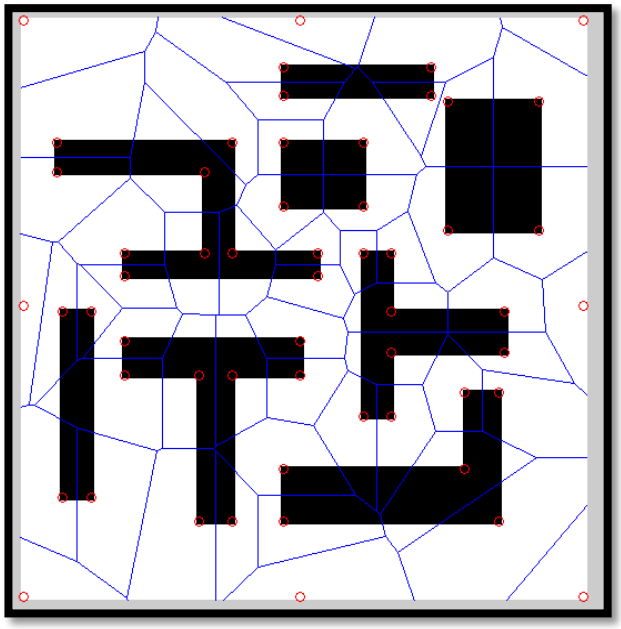

Figura 4-5. Sin duplicado 2

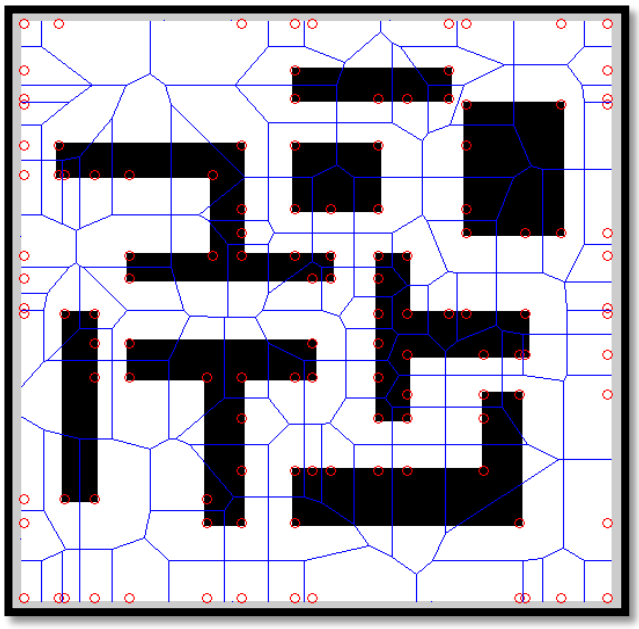

Figura 4-6. Con duplicado 2

Como se puede observar la mejora de precisión es importante, permitiendo una mejor caracterización de las fronteras. Especialmente en las zonas en las que un obstáculo se encuentra a bastante distancia de otro, pues caracterizar los bordes como obstáculos permite que aparezcan trayectorias por esa zona mientras que antes no existían.

Se observa también que cuando el obstáculo es suficientemente largo se producen problemas pues el rectángulo no se modela de forma adecuada. Como se puede ver en la Figura 3-6 con duplicado de puntos, el rectángulo inferior izquierdo no está bien caracterizado, provocando la pérdida de una posible trayectoria en la zona inferior

izquierda junto al borde de la matriz. Para evitar este fallo en el modelado he introducido un largo máximo a la hora de determinar los puntos obstáculos, de este modo cuando se recorre el borde de un rectángulo (3 puntos libres adyacentes) se van añadiendo puntos obstáculos cada un cierto valor de puntos:

Largo máximo = 10  Número de puntos=244

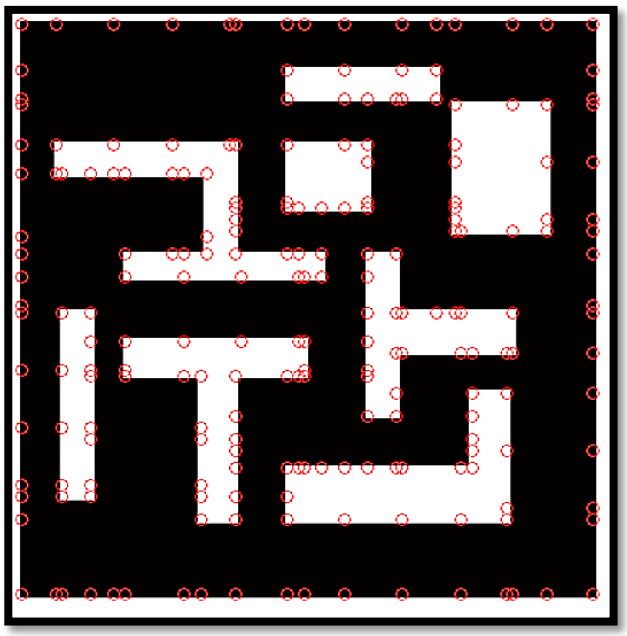

Figura 4-7. $L_{maximo}$ =10 $N_{puntos}$=244  a

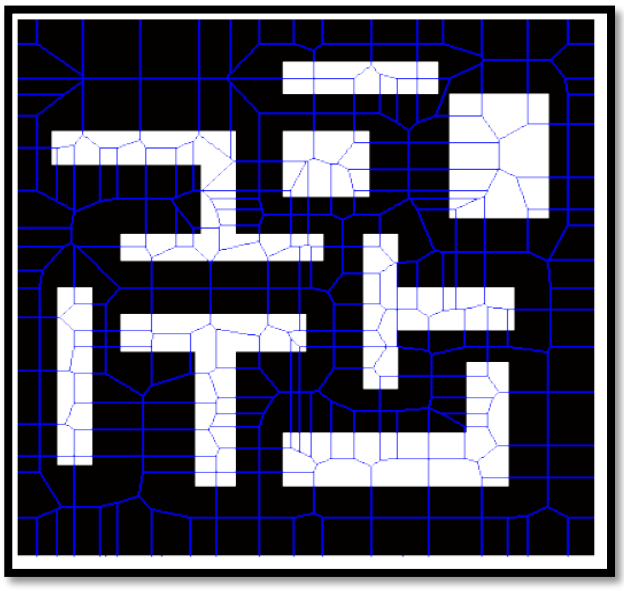

Figura 4-8. $L_{maximo}$ =10 $N_{puntos}$ =244  b

Largo máximo=5 Número de puntos=386

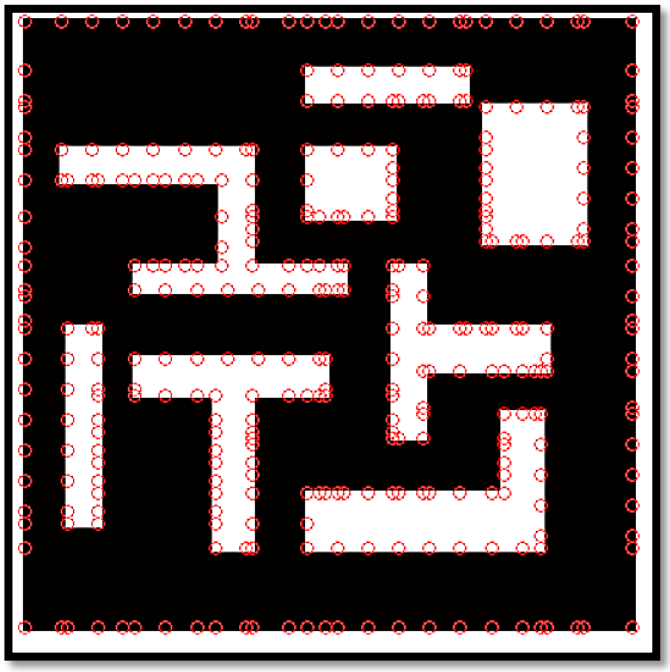

Figura 4-9. $L_{maximo}$ =5 $N_{puntos}$ =386 a

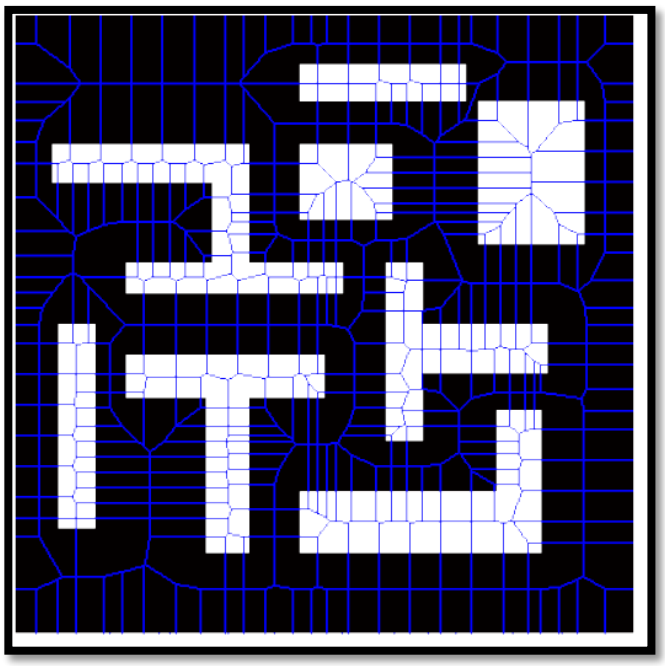

Figura 4-10. $L_{maximo}$ =5 $N_{puntos}$ =386 b

Largo máximo=1 Número de puntos = 2992

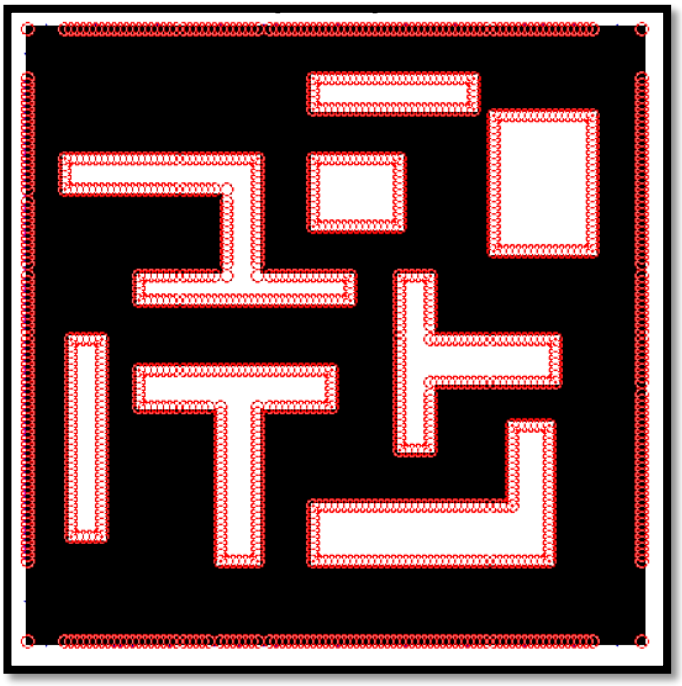

Figura 4-11. $L_{maximo}$ =1 $N_{puntos}$ =2992 a

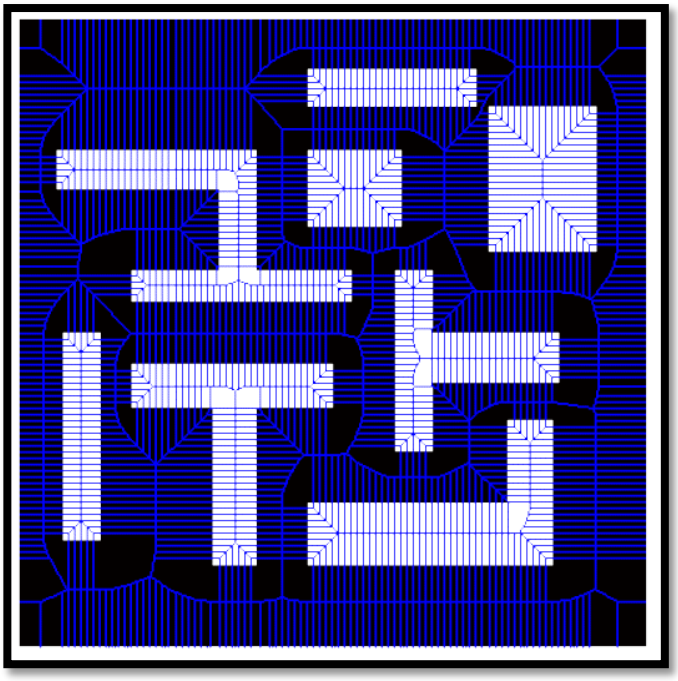

Figura 4-12. $L_{maximo}$ =1 $N_{puntos}$ =2992 b

Como se puede observar, a medida que reducimos el largo máximo, el número de puntos de modelado aumenta de forma importante a la vez que la suavidad y precisión de nuestro grafo aumenta. De forma ideal, a mejor modelación todo son ventajas, sin embargo a la hora de implementarlo en un vehículo móvil que trabaja con restricciones temporales y depende del tiempo de respuesta y de la capacidad de generar una trayectoria a cada cambio de situación de un computador, deberemos tomar una decisión de compromiso que vendrá determinada por las condiciones de diseño.

### 4.1.2 Comprobación de caminos

Una vez tenemos nuestro diagrama de Voronoi, determinado por dos matrices, una para las coordenadas *Y* otra para las coordenadas en *X* de los puntos extremos de cada segmento del grafo, tendremos que eliminar todas las líneas que pasen por algún obstáculo, para ello he utilizado dos métodos para a continuación decidir el más apropiado:

#### 4.1.2.1 Comprobación de puntos del grafo:

Conlleva en principio un menor coste computacional, y es muy sencillo de implementar:

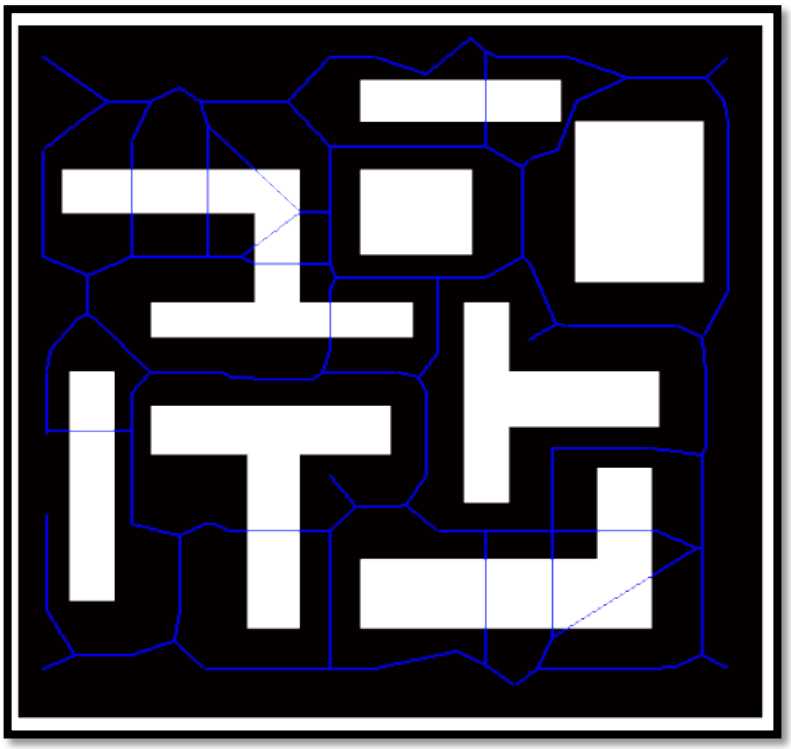

Figura 4-13. Sin largo máximo

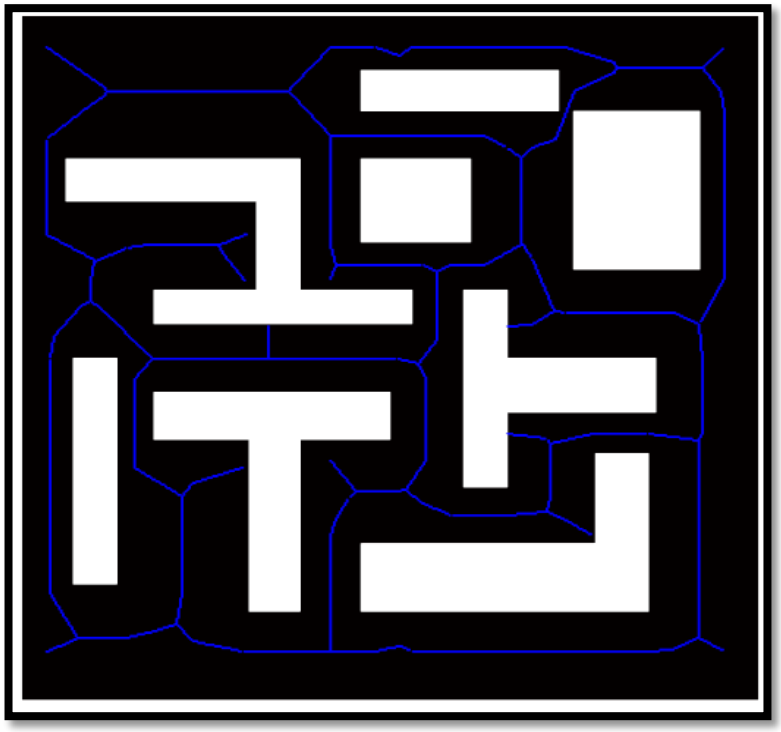

Figura 4-14. $L_{maximo}$=10

Como se puede ver, para un modelado muy preciso de los obstáculos este método se presenta correcto, cumpliendo su función, además de ser poco costoso computacionalmente. Sin embargo, para obstáculos con un peor modelado tiene fallos claros en algunas líneas que podrían tener consecuencias desastrosas en un caso real. Por lo tanto, no es un método adecuado para implementar pues no conocemos el resultado final hasta probarlo, y nunca tendremos garantías de su correcto funcionamiento.

#### 4.1.2.2 Comprobación línea por línea

Método más costoso que el anterior pero con garantías de efectividad, consiste en ir comprobando cada segmento del grafo con salto de unidades entre comprobación y comprobación, que serán redondeadas a la casilla de la matriz mapa más cercana:

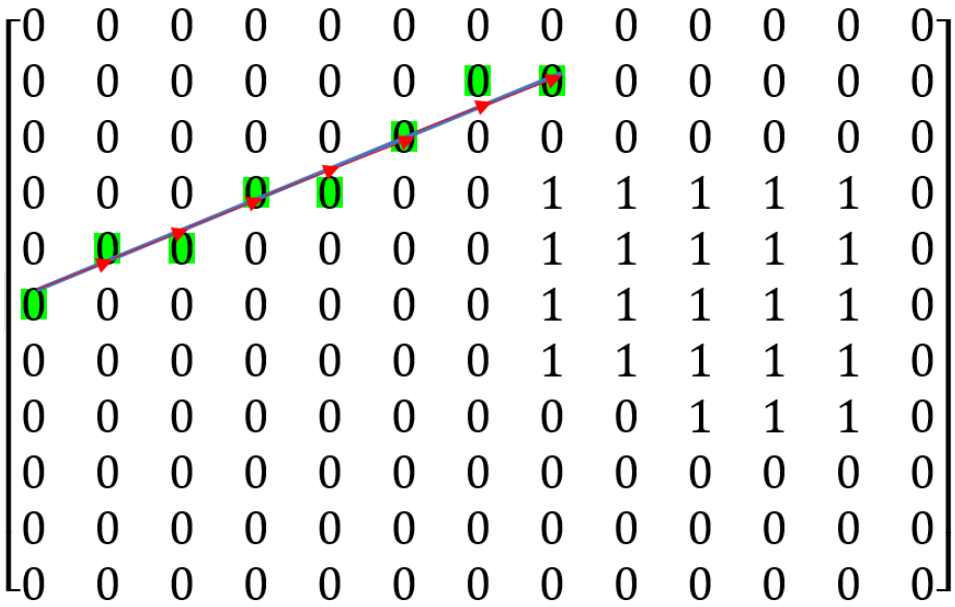


Figura 4-15. Comprobación por líneas

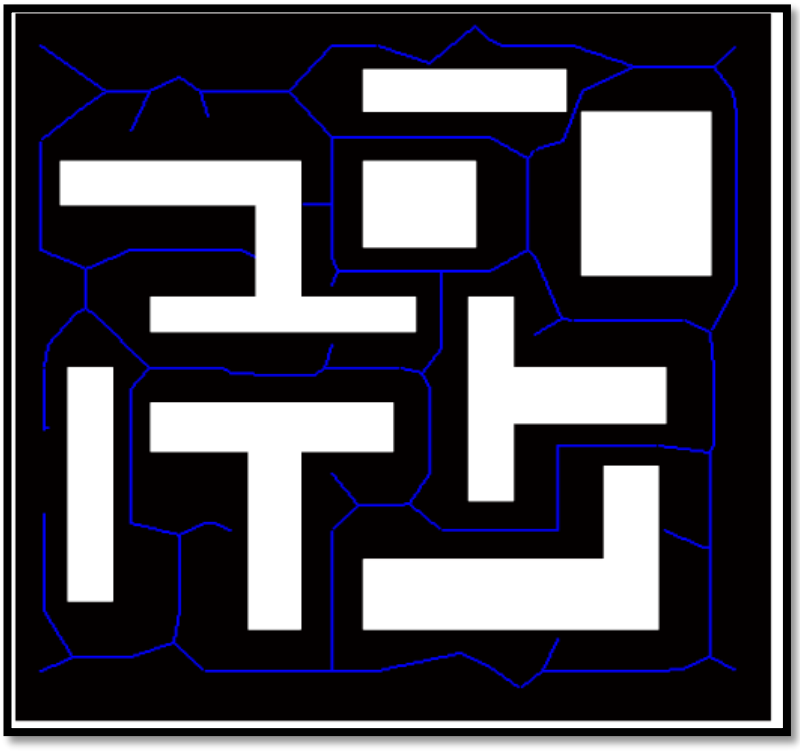

Figura 4-16. Comprobación por líneas y sin largo máximo

Como era lógico no presenta ningún problema en la eliminación de las rectas, ni en el caso arriba representado ni en ningún otro de todas las pruebas realizadas.

En conclusión, el método elegido será el de comprobación línea a línea para poder tener una implementación futura en sistemas reales sin que pueda suponer un problema la incertidumbre que se genera sin la comprobación por línea.

### 4.1.3 Distancia de seguridad

Otro aspecto importante a tener en cuenta es la geometría del vehículo que pueda seguir la trayectoria, pues hay zonas por las que aunque no existan obstáculos le podría resultar imposible pasar. Por tanto se presenta interesante establecer una distancia de seguridad en función de la geometría del vehículo.

Al igual que para la eliminación de líneas, esta comprobación se realizará línea por línea para garantizar una distancia de seguridad homogénea en toda la trayectoria. He implementado dos formas de realizar esta comprobación:

#### 4.1.3.1 Comprobando todas las celdas adyacentes

Constituye un método útil para vehículos con la forma lo más circular posible:

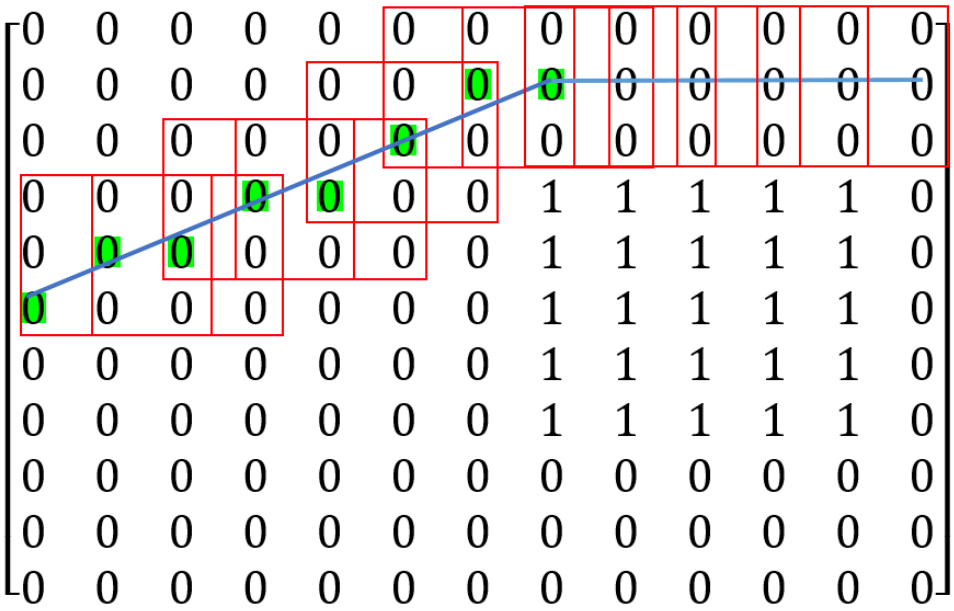

Figura 4-17. Distancia de seguridad 1

Se hace necesario comprobar una gran cantidad de puntos con este método y requiere sobredimensionar el vehículo para que se cumpla la distancia de seguridad, pues en función de la dirección protegerá una distancia igual a media diagonal del cuadro de puntos adyacentes, y en otros solo medio lado de dicho cuadrado.

#### 4.1.3.2 Comprobando perpendicularmente a la dirección del movimiento

Comprueba una menor cantidad de puntos y se adapta a la dirección de la trayectoria:

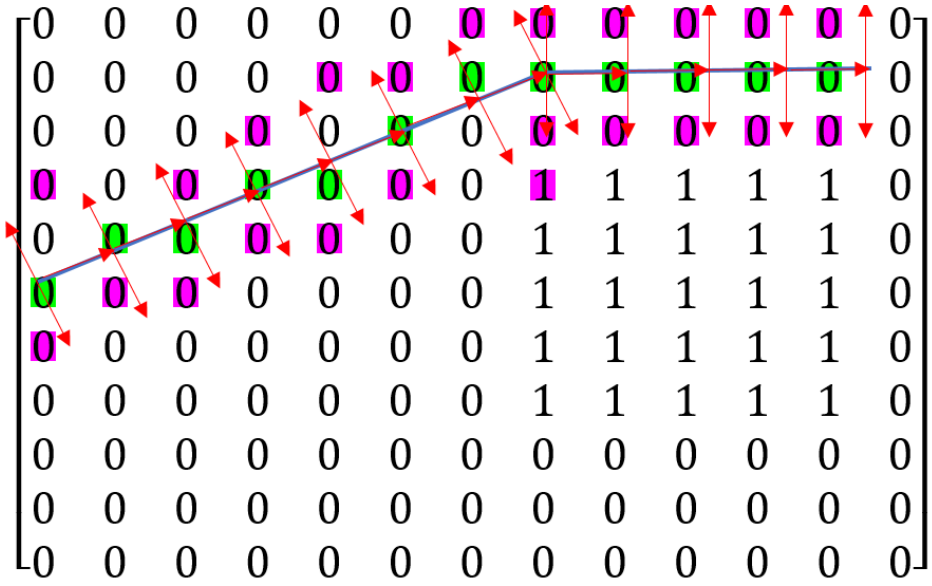


Figura 4-18. Distancia de seguridad 2

La Figura 3-18 muestra cómo se va comprobando la distancia de seguridad, ilustrando en morado los puntos que se comprueban.

#### 4.1.3.3 Conclusiones y pruebas

En cualquiera de los dos métodos, cuando se encuentren con un obstáculo se eliminará la línea completa y se pasará a comprobar la siguiente.Cuando acabe la comprobación, todas las líneas que queden “sueltas”, por no estar unidas a ninguna otra, serán también eliminadas pues no resultan de interés porque el vehículo no podrá acceder a ellas de ninguna manera.

El resultado de establecer determinadas distancias de seguridad es el siguiente:

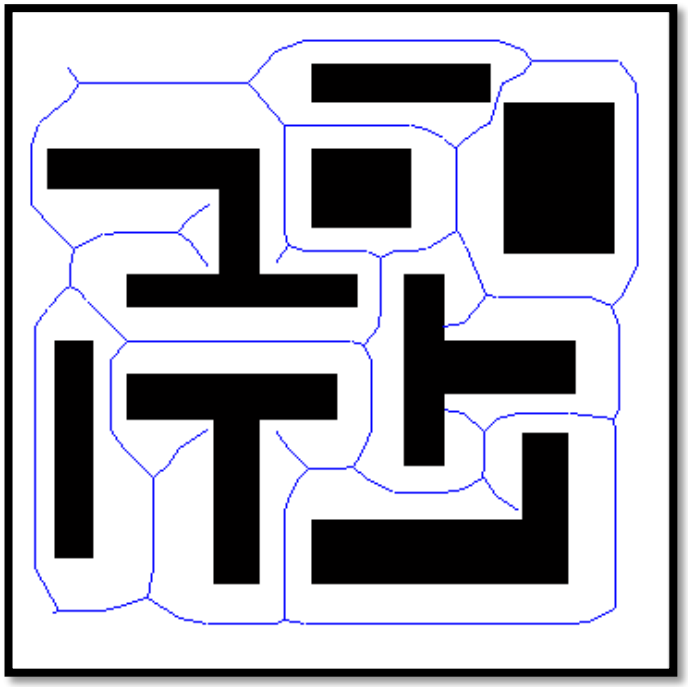

Figura 4-19. Sin distancia de seguridad

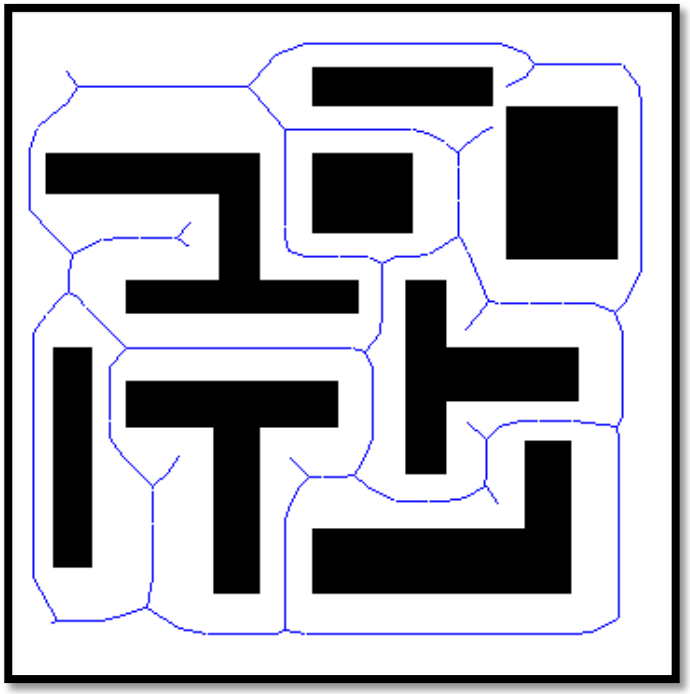

Figura 4-20. Distancia de seguridad 5x5

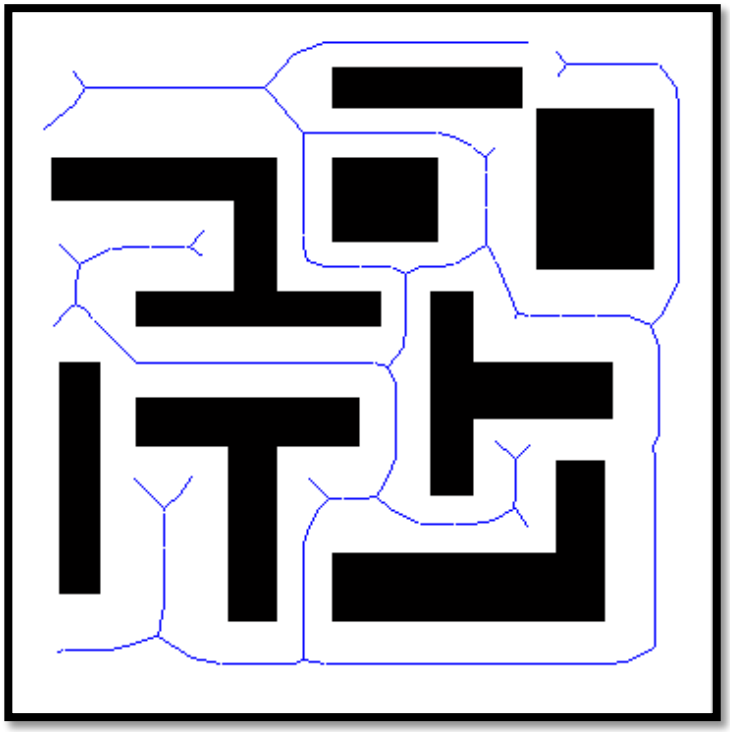

Figura 4-21. Distancia de seguridad 7x7

En otro mapa de mayor tamaño, o de mayor resolución, según como quiera interpretarse:

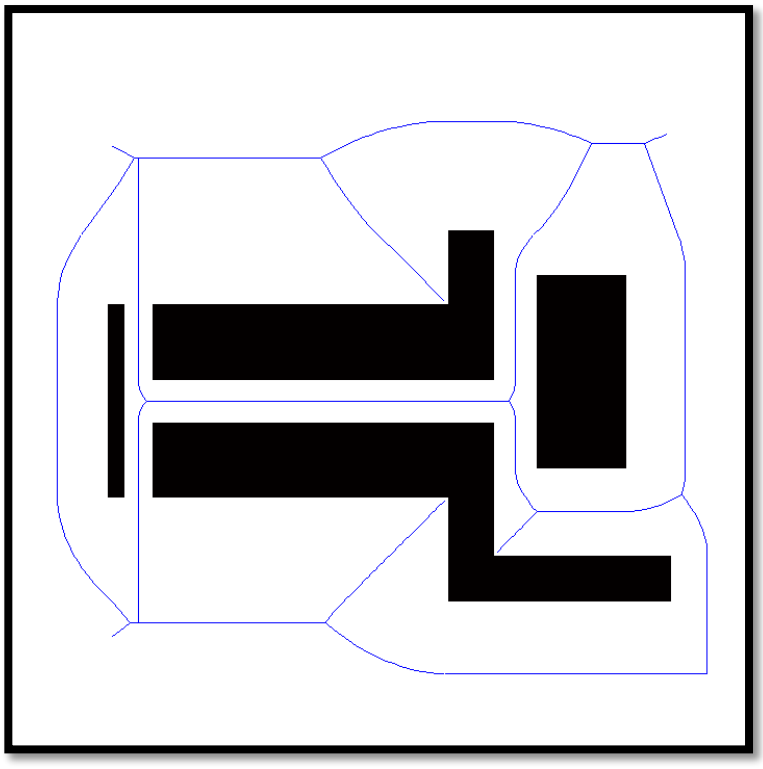

Figura 4-22. Sin distancia de seguridad

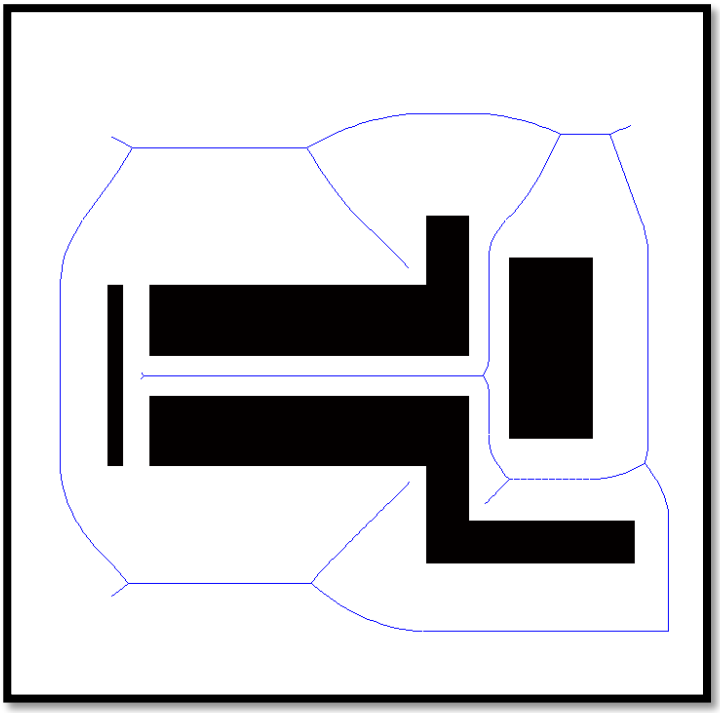

Figura 4-23. Distancia de seguridad 22x22

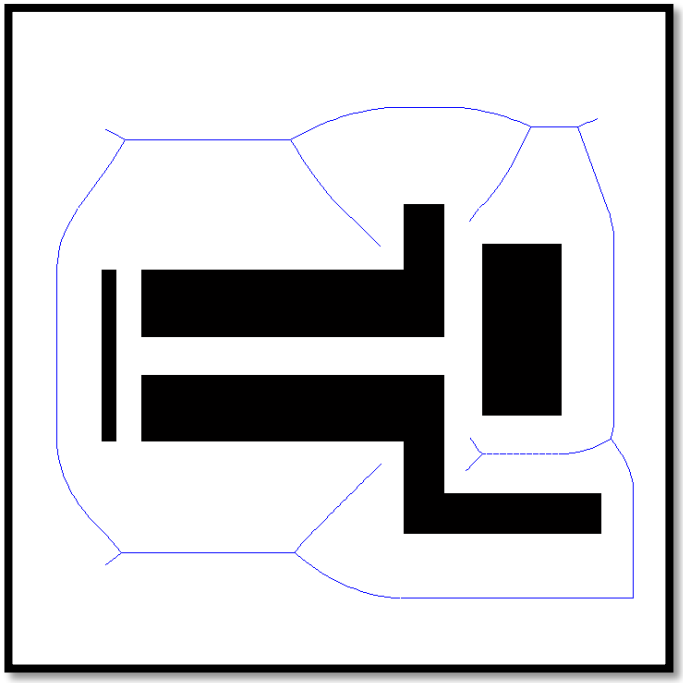

Figura 4-24. Distancia de seguridad 33x33

Se observa como al aumentar la distancia de seguridad las zonas de paso se vuelven mas restrictivas, a la vez que necesita un mayor tiempo de cálculo ya que tiene que comprobar un gran número de celdas.

### 4.1.4 Tiempos de cálculo

Para comprobar el coste computacional aproximado de cada operación he realizado varias pruebas con distintas

configuraciones. Así, he elaborado dos tablas de tiempos de pruebas realizadas al mismo mapa, pero con su resolución aumentada, pasando de ser un mapa 100x100 a un mapa 500x500, siendo *L* el número de celdas en los bordes de los obstáculos entre los que se establecen puntos clave:

| | Tiempo de calculo puntos obstáculo (seg) | Tiempo en calcular diagrama Voronoi (seg) | Número de puntos obstáculo | Tiempo en comprobar línea a línea (seg) | Tiempo en establecer distancia de seguridad=4 (seg) | Tiempo en establecer distancia de seguridad=2 (seg) | Tiempo en establecer distancia de seguridad=1 (seg) | Tiempo en generar matriz de adyacencia (seg) | Tiempo total (seg) |
|---|---|---|---|---|---|---|---|---|---|
| **L máx=1** | 0.037 | 0.029 | 2992 | 0.027 | 0.268 | 0.153 | 0.088 | 1.016 | <1,40 |
| **L máx=5** | 0.016 | 0.022 | 386 | 0.019 | 0.242 | 0.153 | 0.088 | 0.128 | <0.50 |
| **L máx=10** | 0.015 | 0.021 | 244 | 0.018 | 0.242 | 0.149 | 0.088 | 0.063 | <0.40 |
| **Sín L máx** | 0.008 | 0.019 | 136 | 0.018 | 0.233 | 0.148 | 0.088 | 0.038 | <0.35 |

*Valores calculados realizando un ensayo de 6 resultados, descartando valores que excedan un 5% de la media aritmética.

Figura 4-25. Tabla de tiempos para mapa de 100x100

| | Tiempo de calculo puntos obstáculo (seg) | Tiempo en calcular diagrama Voronoi (seg) | Número de puntos obstáculo | Tiempo en comprobar línea a línea (seg) | Tiempo en establecer distancia de seguridad=4 (seg) | Tiempo en establecer distancia de seguridad=2 (seg) | Tiempo en establecer distancia de seguridad=1 (seg) | Tiempo en generar matriz de adyacencia (seg) | Tiempo total (seg) |
|---|---|---|---|---|---|---|---|---|---|
| **L máx=1** | 0.902 | 0.059 | 15400 | 0.061 | 6.086 | 3.833 | 0.169 | 16.530 | <24.0 |
| **L máx=5** | 0.352 | 0.029 | 1592 | 0.033 | 5.850 | 3.775 | 0.168 | 1.149 | <7.50 |
| **L máx=50** | 0.310 | 0.025 | 242 | 0.0028 | 5.535 | 3.821 | 0.168 | 0.046 | <6.0 |
| **Sín L máx** | 0.298 | 0.019 | 136 | 0.031 | 5.657 | 3.654 | 0.169 | 0.043 | <6.0 |

*Valores calculados realizando un ensayo de 6 resultados, descartando valores que excedan un 5% de la media aritmética.

Figura 4-26. Tabla de tiempos para mapa de 500x500

#### 4.1.4.1 Conclusiones

Las tablas permiten sacar muchas conclusiones, la primera de ellas es que establecer una distancia de seguridad se antoja como la operación más costosa cuando esta distancia de seguridad aumenta ligeramente, llegando a límites de más de 6 segundos de cálculo en las peores situaciones, lo cual no se podría aplicar en tiempo real.

La distancia límite también provoca un gran retraso en el cálculo total, ya que aumenta en gran medida los puntos obstáculos a calcular y por tanto el tiempo de cálculo. Esto se puede solucionar en general controlando la resolución del mapa, ya que con un mapa de 100x100 el máximo tiempo de cálculo no excede el segundo y medio, lo que podría permitir operaciones en tiempo real con un tiempo de respuesta aceptable.

Sin distancia de seguridad todos los tiempos son muy reducidos, por lo que se puede decir que la generación del diagrama de Voronoi y su posterior eliminado de líneas que pasan por obstáculos no tiene apenas coste.

Para un número elevado de puntos obstáculos (>1000), los tiempos de cálculo de la matriz de adyacencia aumentan demasiado (>1seg), por lo que conviene tener controlados los parámetros, pues el cálculo de la matriz de adyacencia es el último paso para poder generar planificación de trayectorias utilizando algoritmos de búsqueda del camino óptimo.

### 4.1.5 Matriz de adyacencia y algoritmo Dijkstra

La matriz de adyacencia es una matriz cuadrada que establece relaciones entre distintos puntos. En función de si estos puntos están o no conectados en el lugar correspondiente aparecerá un 0, o la distancia entre ambos puntos.

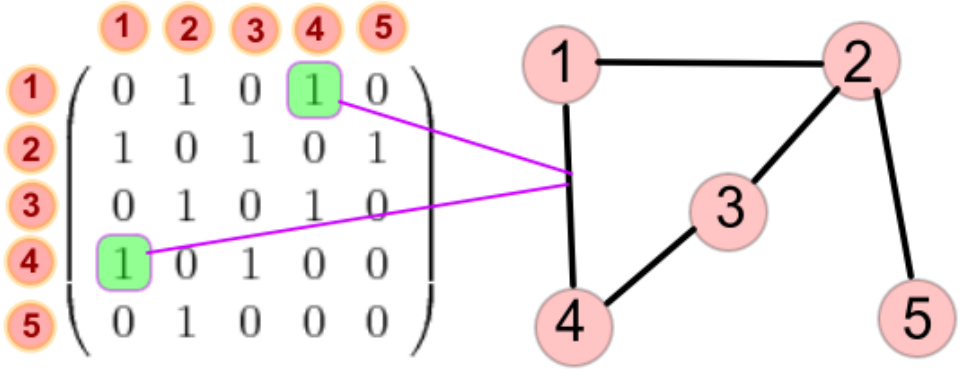


Figura 4-27. Esquema Dijkstra

Para nuestro caso la matriz será no dirigida, es decir, no influye si vas de *A* a *B*, o al contrario pues consideraremos todos los caminos como bidireccionales, por lo que nuestra matriz será simétrica.

Para obtener la matriz de adyacencia en Matlab he tenido que utilizar variables tipo *cell* para crear una celda de puntos en la que cada celda almacene los puntos a los que está conectado, y a continuación realizar una comprobación en bucle para ir rellenando valores en nuestra matriz, que inicialmente será una matriz de *npuntos* x *npuntos* de ceros.

La gran ventaja de este algoritmo es que depende únicamente de una variable de coste, por lo que funciona muy bien para pocos nodos, sin embargo cuando el número de nodos aumenta mucho el algoritmo se hace lento de cálculo debido a la gran cantidad de nodos que tiene que explorar.

Le pasaremos a una función que implementa este algoritmo en Matlab la matriz de adyacencia junto con el punto inicial y el de destino, y nos devolverá la secuencia de puntos que debemos seguir, dando por ejemplo como

resultados:

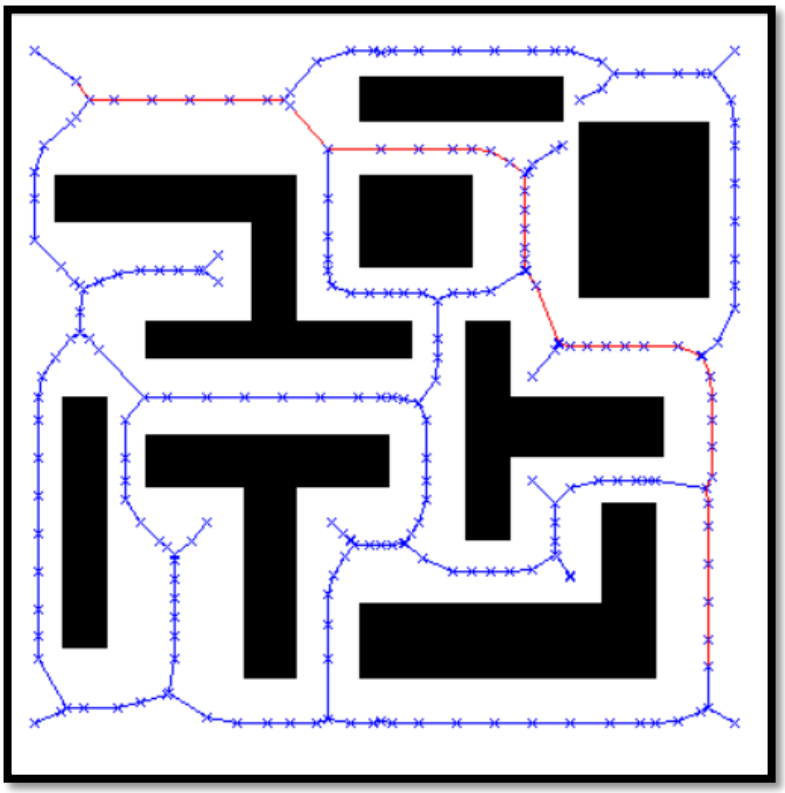

Figura 4-28. Ejemplo Dijkstra 1

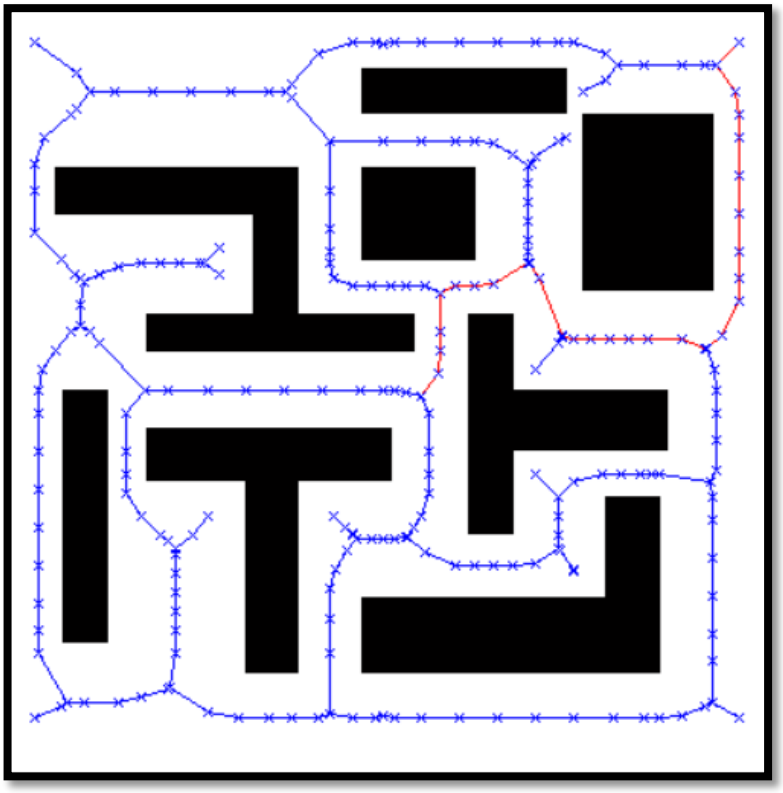

Figura 4-29. Ejemplo Dijkstra 2

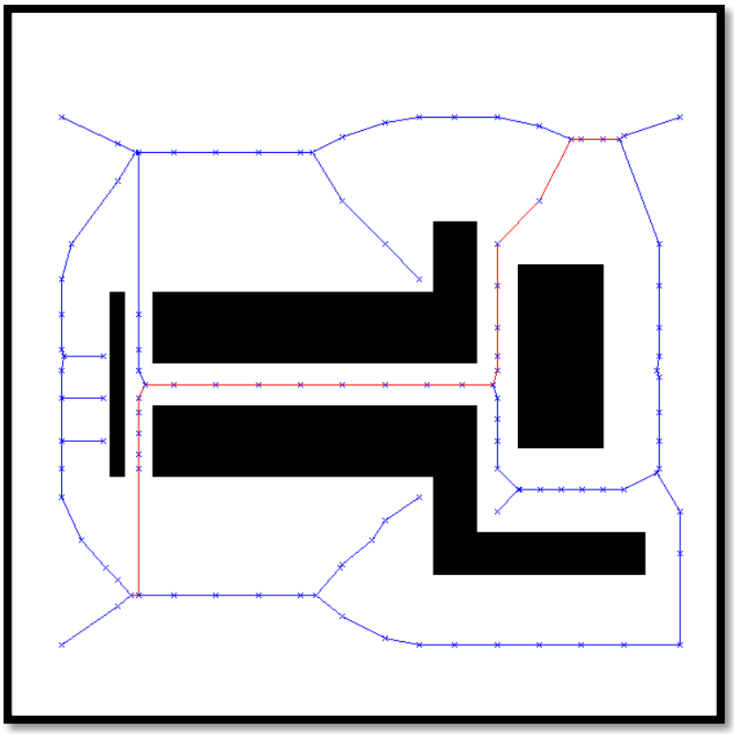

Figura 4-30. Ejemplo Dijkstra 3

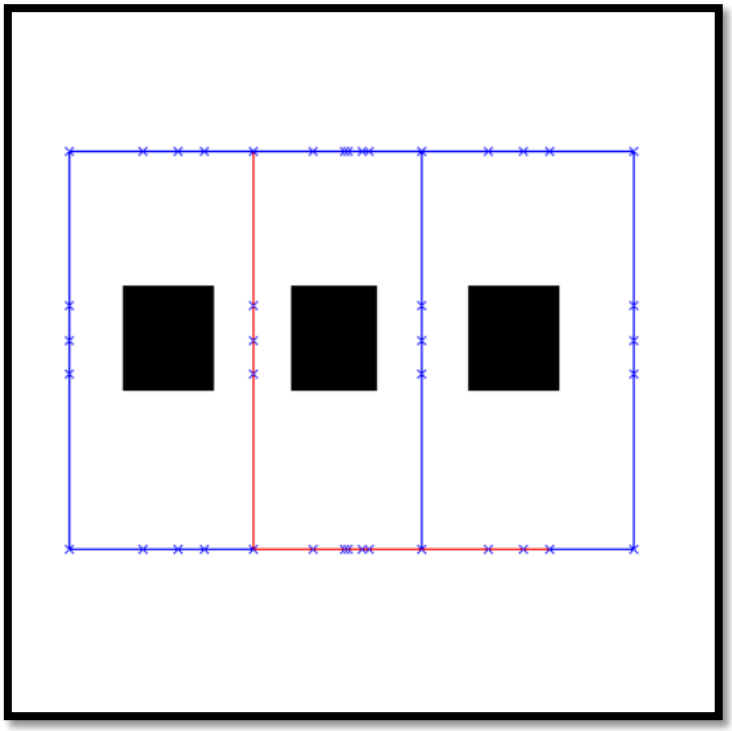

Figura 4-31. Ejemplo Dijkstra 4

El algoritmo nos devuelve de forma eficiente y correcta el camino más corto a seguir, tardando unos tiempos tan reducidos que son difícilmente medibles para estas cantidades de nodos.

### 4.1.6 Conclusiones

A partir de las pruebas realizadas, he podido observar la versatilidad del diagrama de Voronoi para la generación de grafos para seguimiento de trayectorias., También se puede determinar de forma fiable cómo debo modelar un mapa de obstáculos para conseguir un grafo de puntos adecuado para poder pasarle un algoritmo de búsqueda de camino más corto.

Con todo lo expuesto, se tienen los precedentes y la base utilizar el diagrama de Voronoi para generar grafos de puntos en tiempo real, para lo que se utilizará el sistema de programación de robots ROS.

Gracias a las pruebas aquí realizadas, se establecen los métodos más adecuados para tener un coste reducido de cálculo y modelar obstáculos.

## 4.2 Generación de grafos y trayectorias en tiempo real

Es ahora cuando, una vez conocidos los algoritmos idóneos con los que trabajar, teniendo establecida la forma de modelar los obstáculos, y sabiendo cómo depurar los datos obtenidos de un diagrama de Voronoi “en bruto”, se puede comenzar a trabajar en el diseño del sistema de generación de trayectorias que pueda ser aplicado en tiempo real.

Se va a trabajar con dos herramientas a destacar, la primera será el sistema de programación de robots ROS (*Robot Operating System*), que nos permitirá las comunicaciones con el robot para simularlo en tiempo real. Se puede obtener la telemetría y datos de los sensores abordo, al mismo tiempo que se comanda el robot de forma paralela.

El robot utilizado es el muy conocido Turtlebot (ver Figura 3-30), por su flexibilidad para proyectos de investigación y la posibilidad de hacer experimentos reales en un futuro al tener un ejemplar a disposición del

departamento. A bordo llevará un sensor Kinect que proporciona la distancia de los obstáculos que tiene delante en una dimensión 2D, y con un rango angular de 57 grados, y un rango lineal de entre 0.45 y 5 metros.

Figura 4-32. Robot Turtlebot

### 4.2.1 Modelado del espacio visible

El primer paso para generar una trayectoria segura para el robot será obtener información del entorno a partir del sensor considerado. Una vez representado el mapa del entorno se debe interpretar la información del sensor Kinect. En realidad esta información equivale a la proporcionada por un sensor láser en 2D. Este devolverá de forma cíclica un vector de *n* valores, cada uno de esos valores corresponden a una determinada orientación angular respecto a la dirección del sensor, y su valor corresponderá al módulo del vector con dicha orientación y destino el obstáculo que encuentre en esa dirección, si fuese el caso de que en esa dirección no hubiese un obstáculo el valor de una lectura determinada sería NaN.

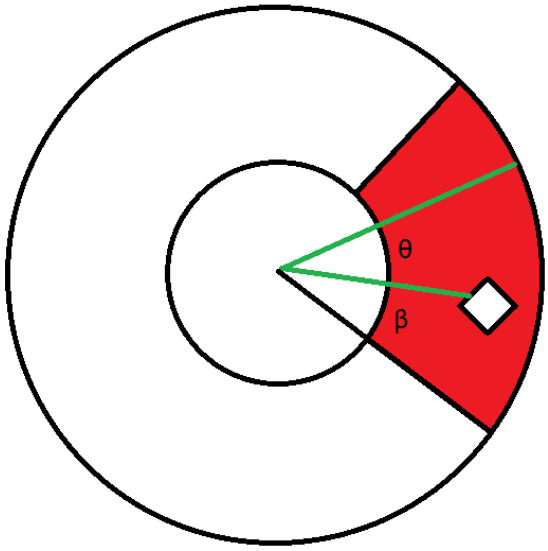


Figura 4-33. Rango de visión Sensor

En la Figura 3-31 el valor del vector para los dos ángulos mostrados sería:

$$V[n_1] = dist(origen\,, obstáculo)\,, para\; n_1 * \Delta angulo = \beta$$
$$V[n_2] = NaN\,, para\; n_2 * \Delta angulo = \beta + \theta$$

Para discretizar el espacio visible vamos a trasladar los puntos a una matriz de ceros y unos en la que denotaremos como uno los obstáculos presentes en una celda y como ceros las zonas vacías. Para esto, primero debemos acotar el rango de visión del sensor para que sea compatible con nuestra zona:

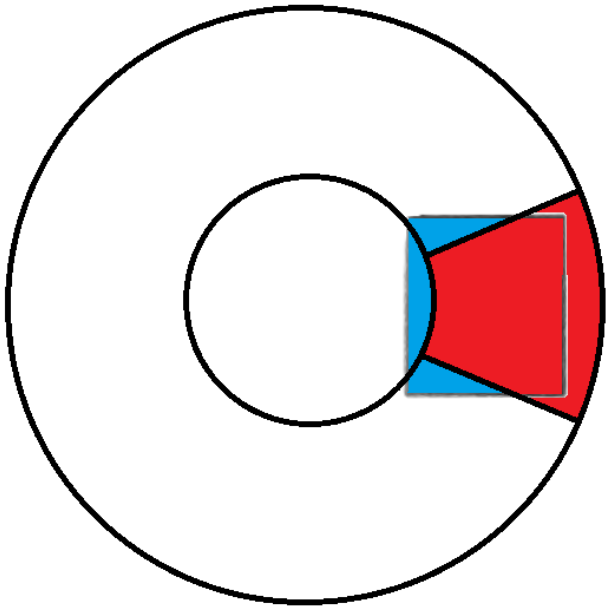

Figura 4-34. Rango de acotado

Como se observa en la Figura 3-32 hay zonas de nuestra matriz que no serán cubiertas por nuestro sensor, en este caso vamos a utilizar una matriz que barra una zona de 5x5 metros, con una resolución de 10 cm, es decir, tendremos una matriz de modelado de obstáculos *Mapa* tal que:

$$Mapa = Mapa[50][50]$$

Al detectar un obstáculo se puede ver claramente que perdemos toda la información del plano en ese rango angular para distancias mayores que la del punto de detección del obstáculo, por lo que tendremos que considerarlas zonas muertas, al igual que los laterales de nuestra matriz que salgan fuera del rango del sensor. Para calcular la posición de cada valor *n* del vector de distancias *V*:

$$Vx[n] = 25 - (V[n] * 10 - 1) * \sin(\beta)\ para\ n * \Delta angulo = \beta$$
$$Vy[n] = (V[n] * 10 - 1) * \cos(\beta)\ para\ n * \Delta angulo = \beta$$

De esta forma obtenemos la posición de los puntos en la matriz de forma que pasando estos valores a entero tendremos las coordenadas discretizadas de los puntos.

Como ya se ha determinado en la parte de mapeo estático con Matlab, vamos a modelar los obstáculos como rectángulos, y duplicaremos los puntos de interés tal y como hemos hecho en los mapas conocidos, para obtener un mejor procesado y precisión de nuestro Voronoi más adelante.

Para nuestro caso concreto, el duplicado de obstáculos se realizará únicamente de manera lateral pues no es de interés establecer "obstáculos ficticios" en la zona de partida del robot salvo unos obstáculos fijos que colocaremos en las esquinas de la matriz y en una distancia determinada a los laterales del punto origen, que para nuestro caso será:

$$Origen = Mapa[25][0]$$

El sensor puede tener errores de precisión y enviar valores NaN entre dos variaciones de obstáculos consecutivas en zonas en las que hay un obstáculo, por lo que para asegurarnos el tener obstáculos sólidos que modelar los puntos detectados por el sensor serán engordados lateralmente, lo cual solo produce un incremento en el tamaño del obstáculo mínimo, de unos 10 cm por lado, que además permite un menor acercamiento a estos.

Podemos observar un ejemplo de como quedaría el modelado de un obstáculo en la imagen:

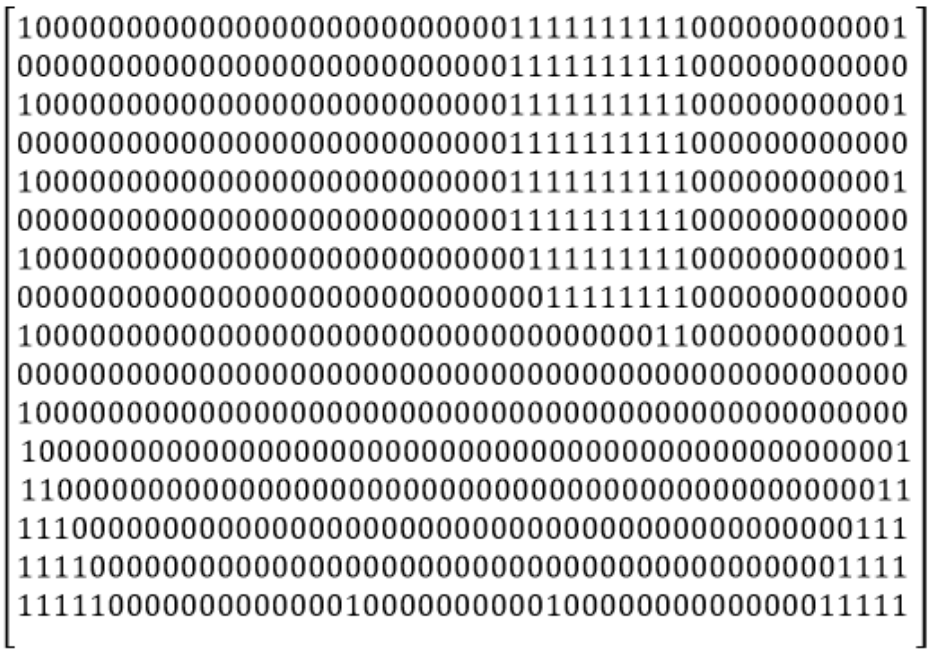


Figura 4-35. Modelado obstáculo

Una vez obtenido el modelado del espacio visible, se pasa a establecer los *puntos clave* que serán los utilizados para generar el diagrama de Voronoi, en este caso la selección de estos puntos clase será trivial pues ya se ha realizado la selección en Matlab.

### 4.2.2 Generación de trayectorias

Una vez modelados los obstáculos detectados, se debe establecer un plan de actuación según la situación del robot y estos obstáculos. Al comienzo el robot se dirige al punto de destino. Durante la navegación del robot y con una cierta frecuencia se irán analizando las lecturas del sensor, para comprobar si el robot puede alcanzar el destino o puede colisionar con un obstáculo. En caso de posible colisión, se lanza el algoritmo de Voronoi y el algoritmo Dijsktra para actualizar camino generando nuevos waypoints que aseguren la evitación de la colisión.

De esta forma, solo se computará el algoritmo de evitación de obstáculos cuando sea necesario, evitando una posible ralentización del sistema. Respecto al seguimiento de caminos, cuando el robot esté a una distancia menor de 10 cm del waypoint actual se dirigirá al próximo waypoint. Esto aporta fluidez y continuidad a las trayectorias, pues la velocidad lineal es reducida y no da lugar a problemas de aproximamiento excesivo a los obstáculos.

Se puede entender de forma más visual con el siguiente diagrama de flujo:

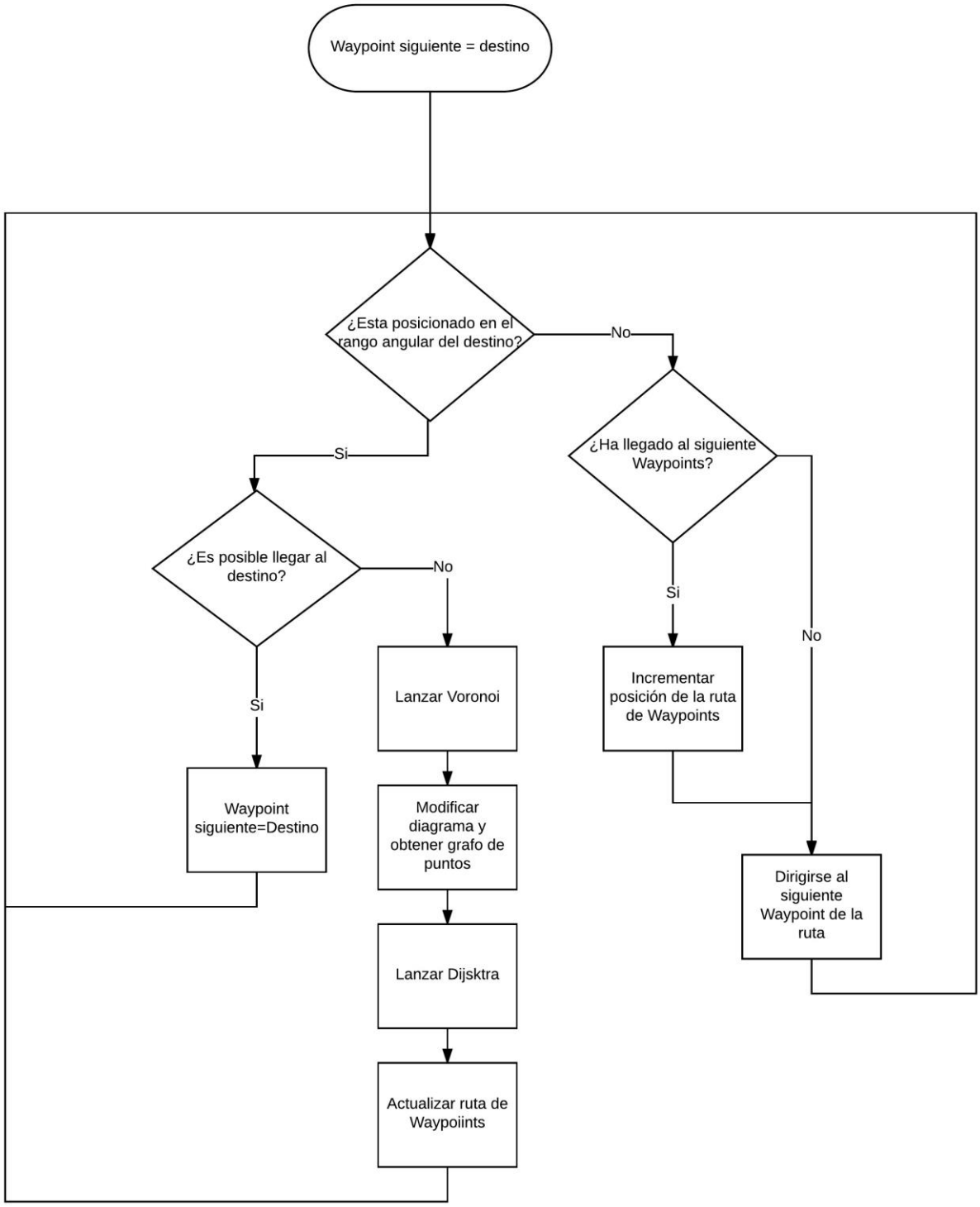


Figura 4-36. Plan de generación de trayectorias

Para determinar la posición de los waypoints respecto al sistema de coordenadas global debemos realizar una traslación y rotación de los ejes coordenados, pues el sensor da las coordenadas en su sistema de referencia local en coordenadas polares (módulo y ángulo), que posteriormente hemos pasado a un sistema de coordenadas

cartesianas local del sensor con la matriz de obstáculos La transformación es la siguiente:

$$POS_{x\ globales\ WP} = POS_{x\ globales\ robot} + (POS_{x\ locales\ WP} - 2.5) * \cos\left(theta - \frac{\pi}{2}\right) - POS_{y\ locales\ WP} * \sin\left(theta - \frac{\pi}{2}\right)$$

$$POS_{y\ globales\ WP} = POS_{y\ globales\ robot} + POS_{y\ locales\ WP} * \cos\left(theta - \frac{\pi}{2}\right) + (POS_{x\ locales\ WP} - 2.5) * \sin\left(theta - \frac{\pi}{2}\right)$$

Los cambios de coordenadas se realizan después de restar a las coordenadas locales en *x* una distancia de 2.5 metros, ya que la posición del sensor en su sistema de coordenadas locales son (2.5 0), es decir, el sensor se encuentra situado en la mitad de la matriz de obstáculos respecto al eje *x*. En la imagen podemos ver una representación en Gazebo de las coordenadas:

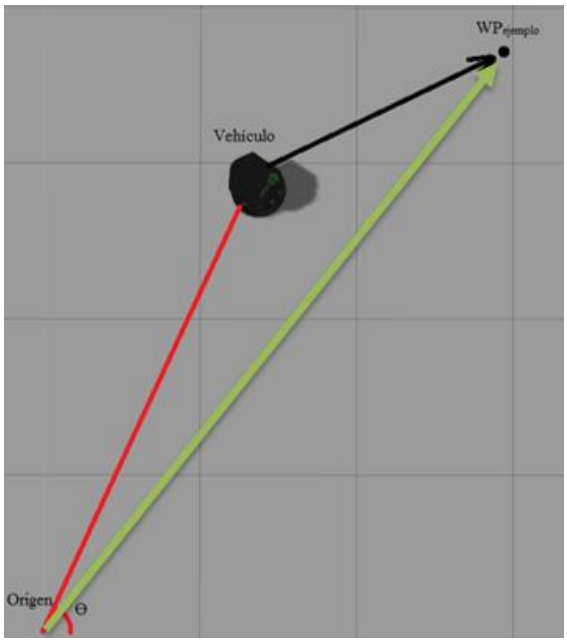


Figura 4-37. Cambio de Coordenadas respecto al sensor a respecto al origen

Para el guiado del vehiculo se establece una velocidad lineal constante, y una velocidad angular proporcional a la diferencia entre la orientación del robot, y la recta que une el robot con el siguiente waypoint (que sería la orientación deseada), siendo $\alpha$ la orientación del vehículo respecto al origen y $\beta$ la del waypoints respecto al vehículo, es decir:

$$w = K * (\alpha - \beta)\frac{rad}{s}, para\ K = cte$$

$$v = cte$$

Para comprender la diferencia angular entre el obstáculo se puede observar la siguiente representación gráfica:

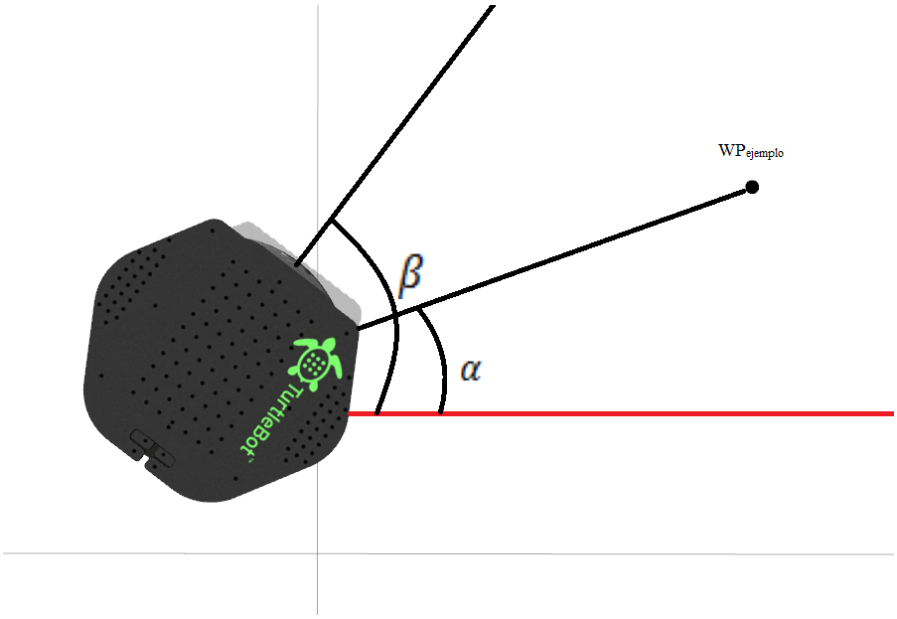


Figura 4-38. Cálculo velocidad angular

### 4.2.3 Toolbox de Matlab para conexionado con ROS y árbol de conexiones

Para obtener el diagrama de Voronoi se va a hacer uso de la novedosa herramienta de Matlab Robotics System Toolbox que ha incorporado en su versión 2015.a.Esta herramienta comunicar Matlab y ROS, con lo que se puede trabajar con sistemas en tiempo real, se puede monitorizar su estado, analizar los resultados, visualizar gráficas de estado, o interaccionar con Gazebo y V-REP para simulaciones de sistemas. La Figura 3-37 muestra sus utilidades.

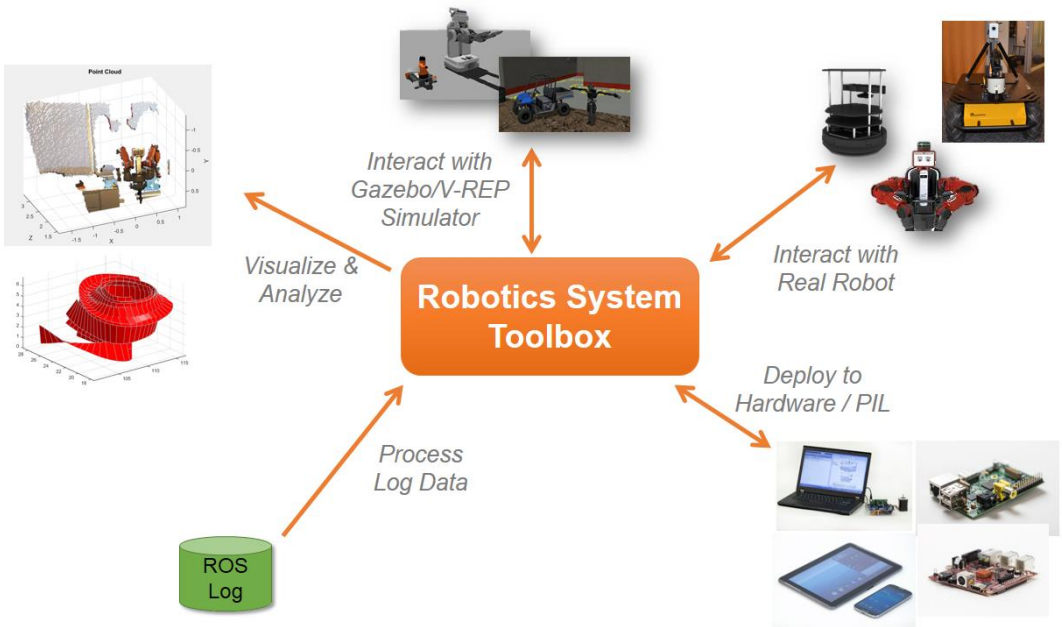


Figura 4-39. Robotics System Toolbox

Para nuestro caso concreto, estableceremos un conexionado entre nuestro nodo *main*, en donde se va comandando al vehiculo y calculando las trayectorias, y un nodo global que generará Matlab de forma automática. Ambos están conectados mediante un canal de comunicación "*topic*", a través del cual ROS solicitará el cálculo del diagrama de Voronoi a Matlab cada cierto tiempo, enviando un vector con los puntos clave con los que se pretende calcular el diagrama, y Matlab devolverá una matriz de [*n*][4], donde cada fila constituye una recta del diagrama, determinada por las coordenadas $X_{1n}$ y $Y_{1n}$ del punto inicial del segmento, y los puntos $X_{2n}$ y $Y_{2n}$. Podemos observar en el siguiente diagrama de conexiones el árbol de nodos y *topics* que constituyen todo el sistema:

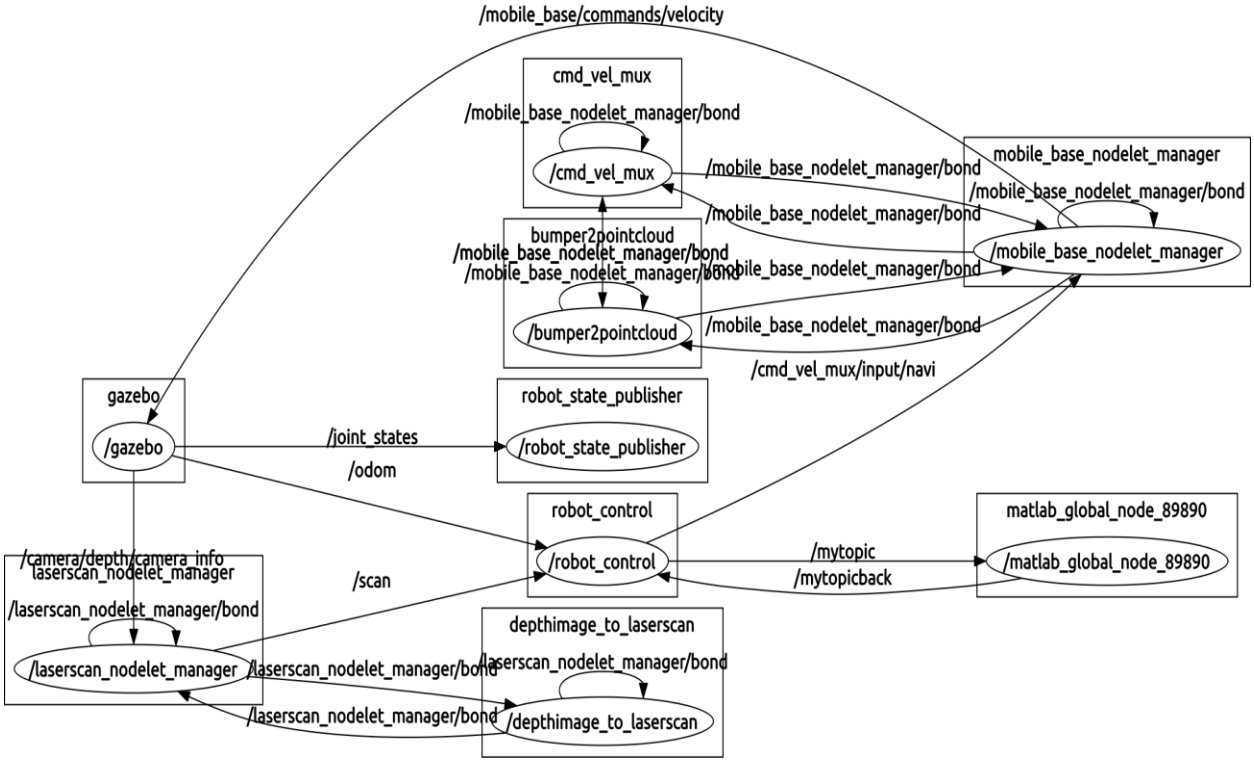


Figura 4-40. Árbol de conexiones

En el esquema se pueden observar los *topics* de conexión entre Matlab y el nodo *robot_control,* que son */mytopic* y */mytopicback*, el topics /scan para conectar las lecturas del láser que obtiene a partir de Gazebo con *robot_control*, y el topic */odom* para enviar el estado del robot desde Gazebo al control del robot.

La zona superior del esquema computa la dinámica del Turtlebot a partir de la velocidad impuesta por el control del robot, que envía a través del topic */cmd_vel_mux/imput/navi* que tras procesarla se la envía a Gazebo con el topic */mobile_base/commands/velocity.*

# 5 EXPERIMENTACIÓN

*El valor de una idea radica en el uso de la misma.*

*- Tomas A. Edison -*

En este capítulo vamos a mostrar algunos ejemplos de trayectorias seguidas por el vehiculo para distintos escenarios y para ver cómo evita obstáculos y llega al punto objetivo . Se ha utiilzado el entorno de simulación Gazebo de ROS para las pruebas, en vistas a poder probar el sistema para entornos reales.

Una representación en video plasma de mejor forma el comportamiento dinámico del robot y la evitación de obstáculos, sin embargo con una serie de imágenes de varios experimentos se puede tener una idea del comportamiento del sistema.

## 5.1 Experimento 1

En el primer experimento hemos fijado el objetivo a 18 metros del punto inicial del robot. Cada cuadrado de la superficie del terreno tiene unas dimensiones de 1x1 metros. Al iniciar el recorrido, el robot toma inicialmente la dirección del punto objetivo, y nada más detectar un obstáculo que le impide llegar al objetivo, lanza Voronoi y calcula el camino óptimo con Dijsktra para evitar colisionar con el obstáculo detectado.

En un primer instante se genera un vector de waypoints que determinan el recorrido que se va a realizar en principio, y durante el trayecto se va a ir comprobando si se puede llegar al objetivo directamente. Esto hará que se acorte camino y sea más eficiente el recorrido: Como ya se explicó, para esto el sistema va a ir comprobando si en la dirección de movimiento existe algún obstáculo. Se considera una anchura de comprobación de unos 60 cm, para evitar pasar demasiado cerca de los obstáculos.

En la siguiente imagen podemos observar el recorrido calculado inicialmente:

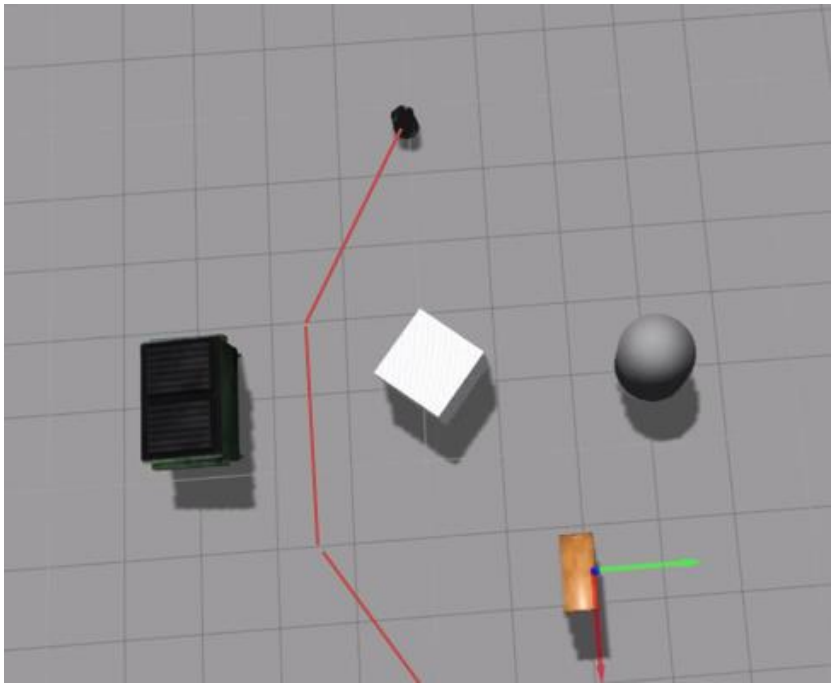

Figura 5-1. Traza del recorrido inicial

El robot va pasando por los waypoints  mientras el camino calculado sea el correcto y no se pueda llegar al objetivo directamente:

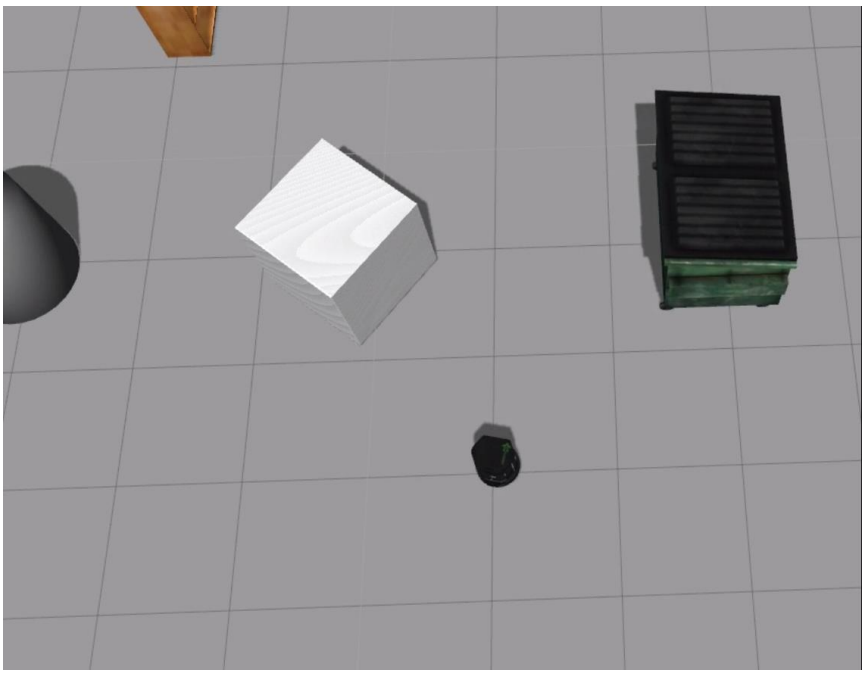

Figura 5-2. Recorrido ejemplo 1.1

En el mismo instante en que el camino al objetivo sea posible, se establece el destino como el siguiente waypoint, como se puede observar en la sFigura 4-3.

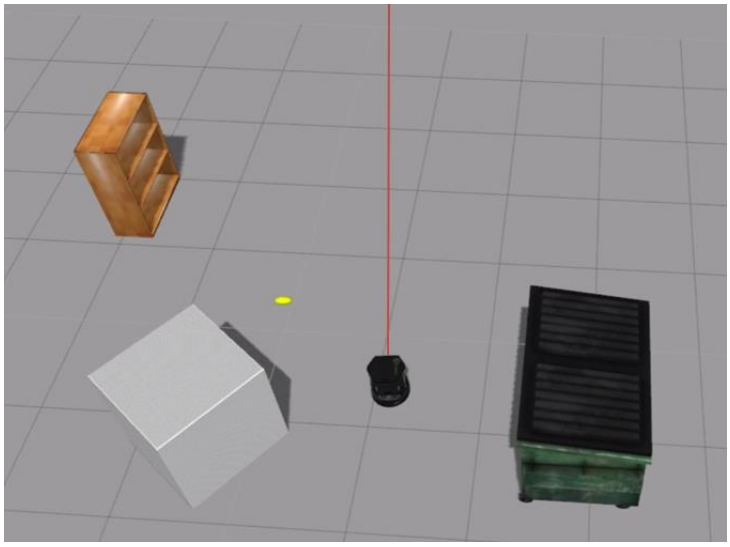

Figura 5-3. Recorrido ejemplo 1.2

Ahora se sitúa en tiempo real un obstáculo justo delante del recorrido, e inmediatamente recalcula el camino y genera una nueva trayectoria:

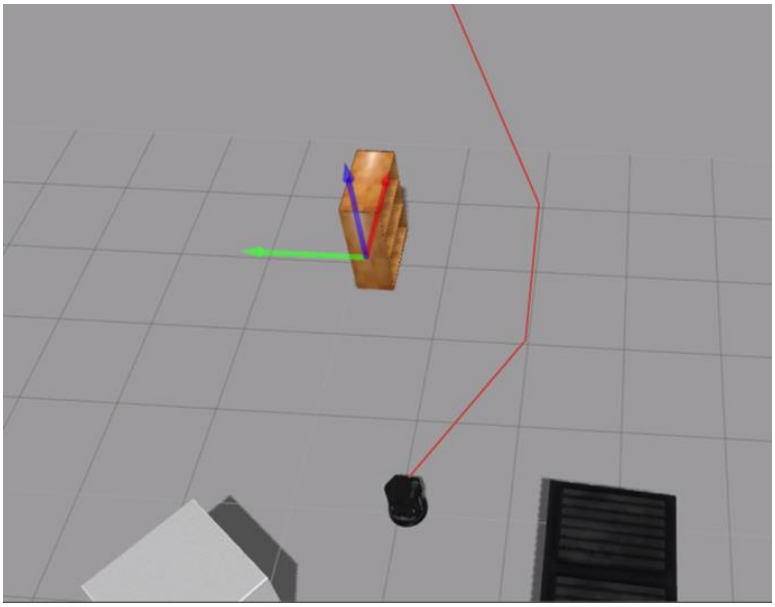

Figura 5-4. Recorrido ejemplo 1.3

Como se observa, el robot va trazando las trayectorias eligiendo el camino más corto y de forma que queda lejos de los obstáculos del camino para generar trayectorias seguras:

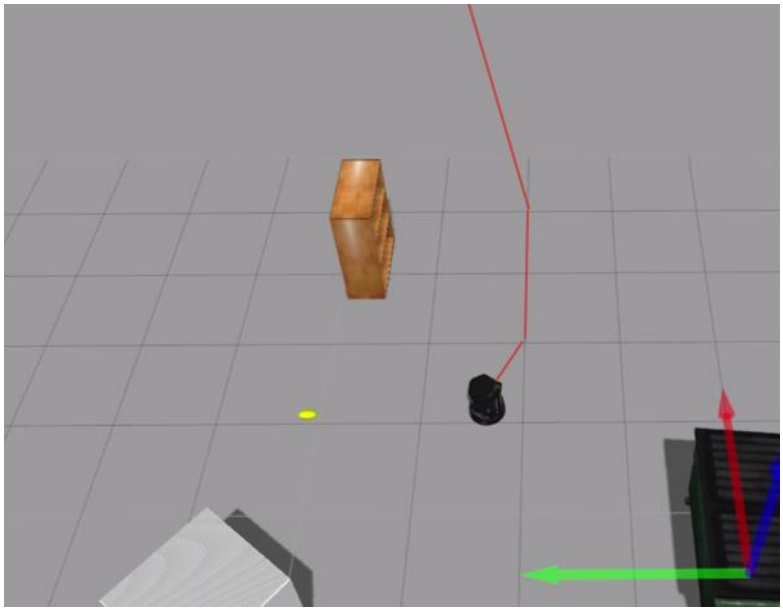

Figura 5-5. Recorrido ejemplo 1.4

De nuevo, una vez que el objetivo es accesible directamente, el vehiculo toma su dirección:

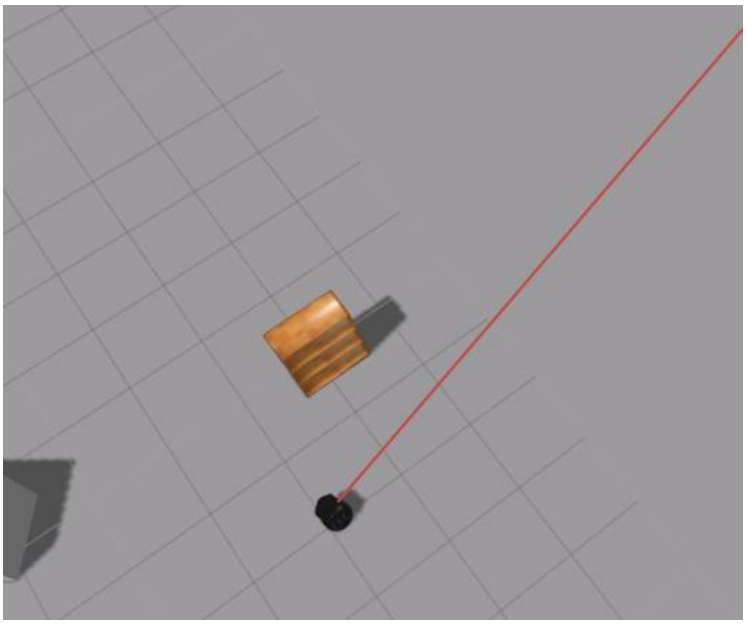

Figura 5-6. Recorrido ejemplo 1.5

Ahora vamos a introducir un obstáculo cercano al anterior justo delante del robot cuando el anterior ha salido del rango del sensor, en ese momento el robot cambia de nuevo de trayectoria y se acerca demasiado al obstáculo anterior, por lo que se puede determinar que el rango angular del sensor podría ser demasiado reducido para ciertos casos, además de que el modelado de las zonas muertas en la matriz *mapa* debe ser más preciso para evitar el acercamiento a zonas desconocidas:

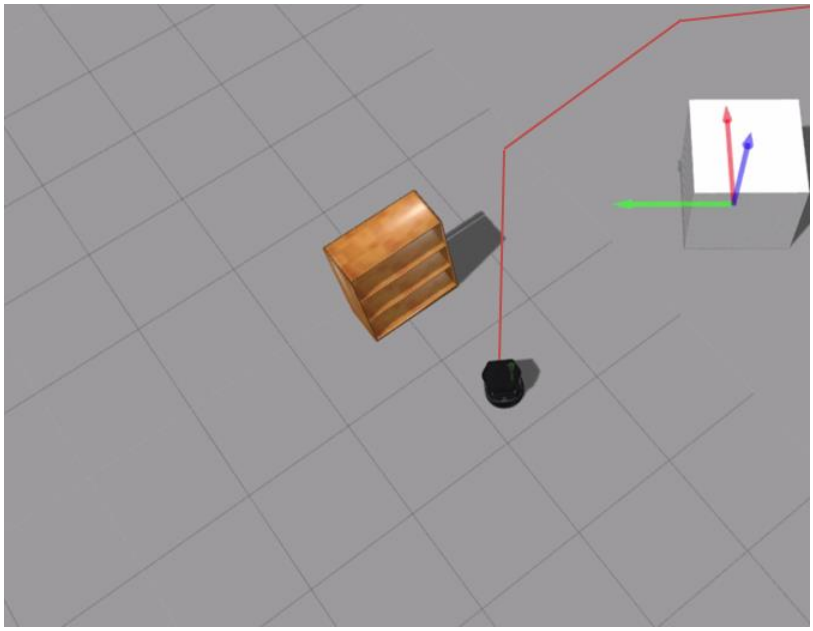

Figura 5-7. Recorrido ejemplo 1.5

Una vez más , el robot toma el camino directo al no detectar obstáculos en la dirección hacía el destino:

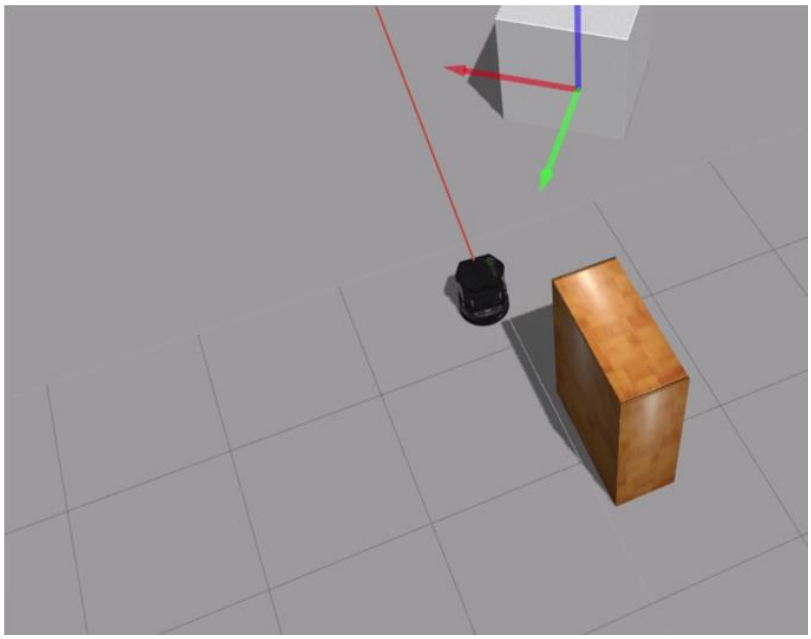

Figura 5-8. Recorrido ejemplo 1.6

Una vez que llega al destino el vehiculo se detiene y el sistema avisa por pantalla de la llegada al destino con el mensaje *goal reached*.

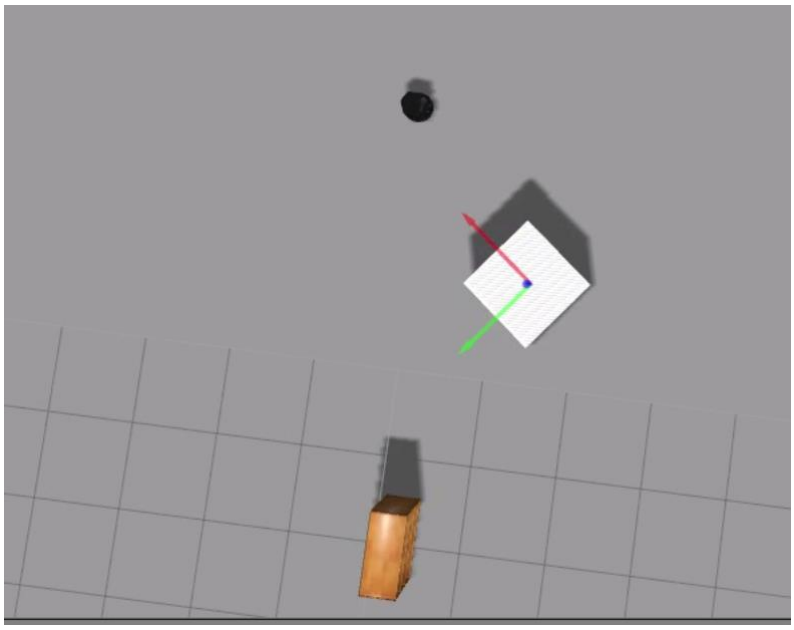

Figura 5-9. Recorrido ejemplo 1.7

## 5.2 Experimento 2

En este caso el recorrido la distribución de los obstáculos y su número es diferente. Podemos observar en la Figura 4-10 que existe un obstáculo de gran longitud que provoca que el sensor detecte una gran cantidad de puntos obstáculos que podrían saturar al sistema de generación de trayectorias, sin embargo se observa que el sistema responde bien ante tal cantidad de obstáculos y no tiene problemas para procesar los datos.

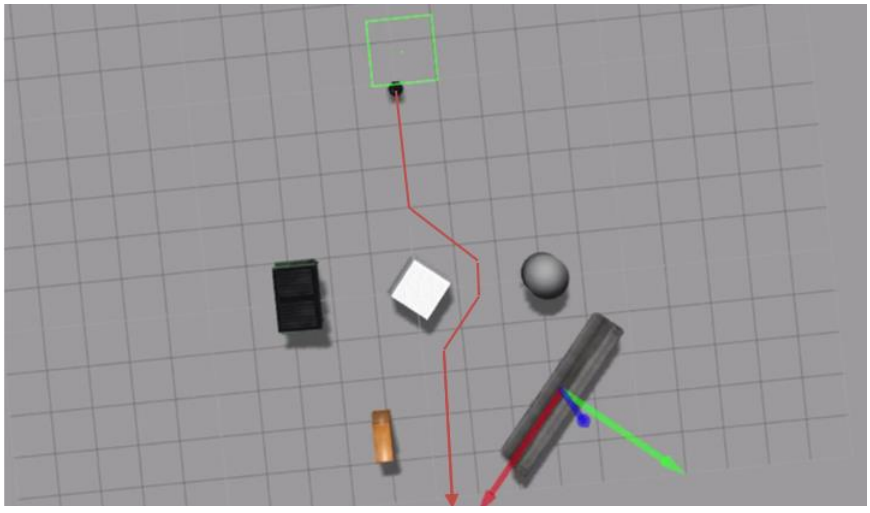

Figura 5-10. Recorrido ejemplo 2.1

Al igual que en el anterior experimento, el vehiculo comienza esquivando el primer obstáculo sin problema, manteniendose en todo momento a una distancia prudente y realizando una trayectoria suave.

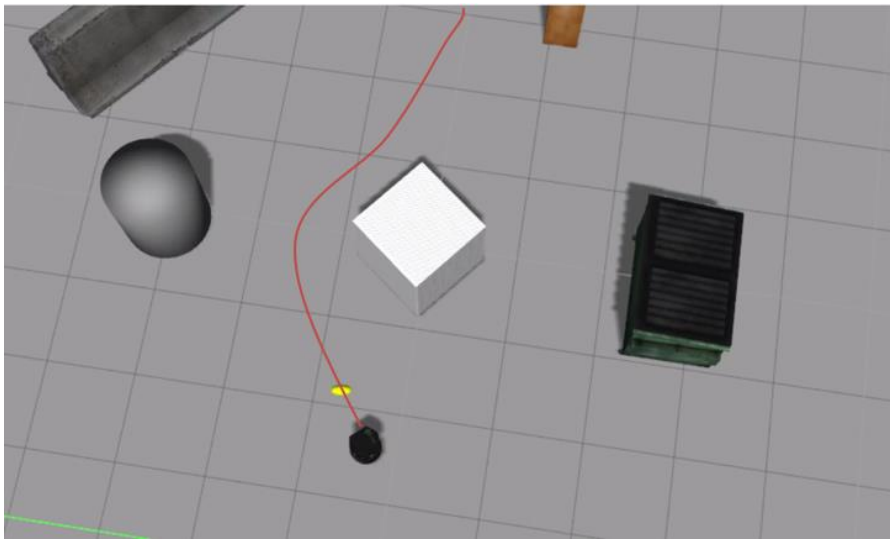

Figura 5-11. Recorrido ejemplo 2.2

En el momento en que detecta el obstáculo en forma de barrera alargada decide tomar el único camino posible, cambiando la trayectoria, lo que provoca un acercamiento excesivo al primer obstáculo detectado, pero respetando la distancia de seguridad.

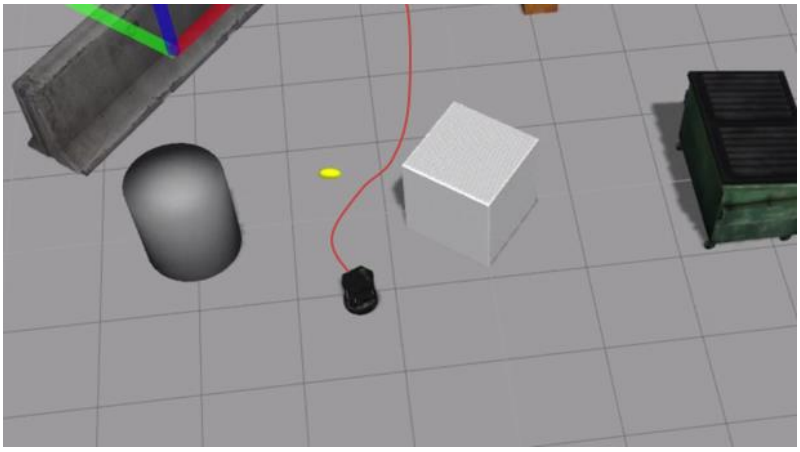

Figura 5-12. Recorrido ejemplo 2.3

Se puede observar a continuación como al tener el *goal* en el rango de visión del sensor y no detectar ningún obstáculo en su camino el robot cambia su dirección y se dirige de forma directa al objetivo. Otro dato que se ha tenido en cuenta es que es posible que detecte un obstáculo en ese rango angular, pero que siempre y cuando esté situado a una distancia mayor que el objetivo, el robot seguirá dirigiéndose al objetivo, ya que el obstáculo se encuentra después de este.

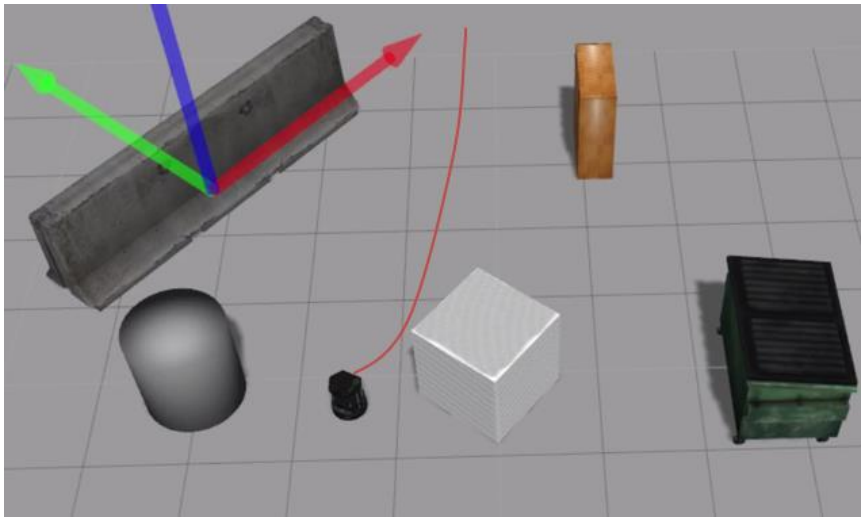

Figura 5-13. Recorrido ejemplo 2.4

Al pasar todo el conjunto de obstáculos, se sitúan dos obstáculos más en los laterales del camino, y como era de esperar, sigue respondiendo perfectamente, ya que estos no interrumpen el movimiento hacía el objetivo.

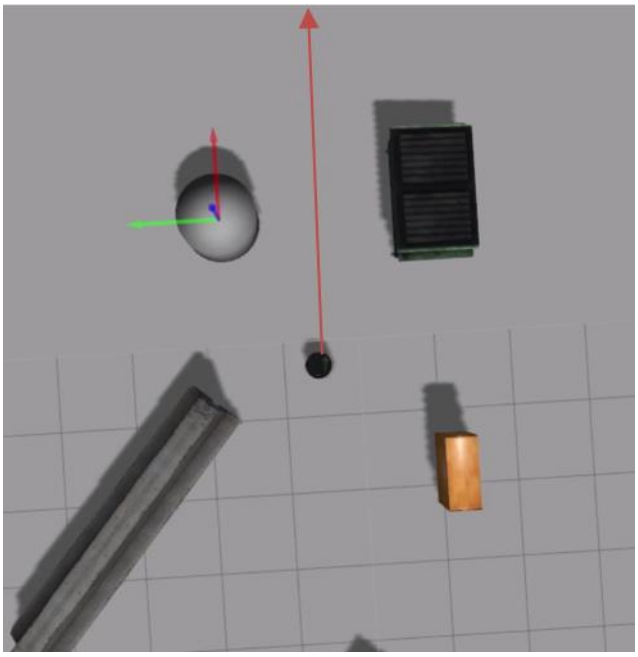

Figura 5-14. Recorrido ejemplo 2.6

Por último se coloca un obstáculo a una distancia proxima al robot para comprobar su capacidad de reacción a poca distancia del obstáculo, y tambien responde con velocidad, alejándose de este de forma adecuada y suave. Esto es permitido, en gran parte, a la reducida velocidad lineal establecida, y a la rápida velocidad angular de este que le permite maniobrar en espacios reducidos. La trayectoria óptima desde el punto de vista de la distancia a los obstáculos sería la trayectoria rectilineas, haciendo su velocidad lineal cero cada vez que alcanze un waypoint, y volviendo a aumentarla cuando el ángulo del robot esté orientado al siguiente waypoint. Sin embargo esto provoca trayectorias bruscas y ralentiza mucho el recorrido, por lo que hay que llegar a un

equilibrio entre eficiencia y seguridad que dependerá en gran medida de la dinámica del robot.

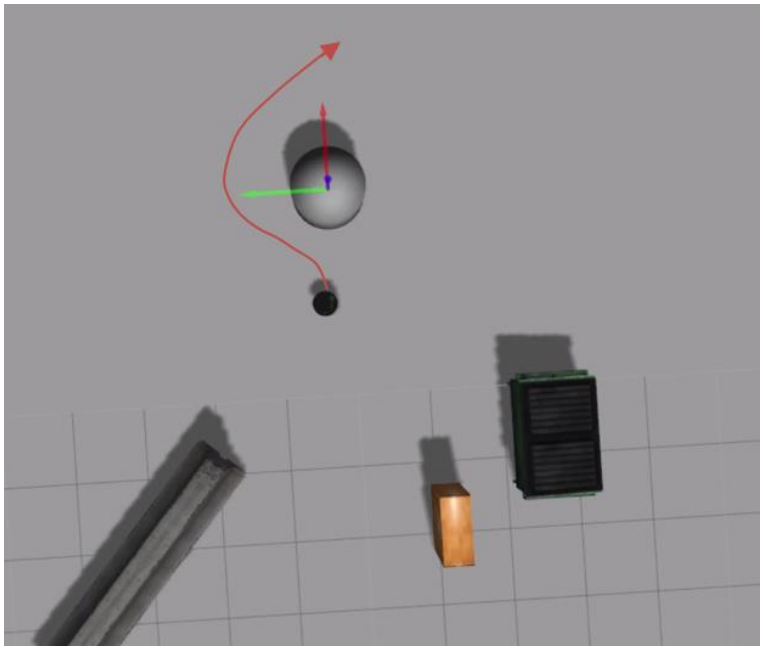

Figura 5-15. Recorrido ejemplo 2.7

Finalmente el robot llega al objetivo sin haber tenido problemas de colisiones ni de saturación del sistema.El camino presentaba una gran cantidad de obstáculos situados a una distancia moderada entre sí, que permiten al sistema reaccionar con tiempo suficiente, sin tener que actualizar la ruta a seguir demasiadas veces como para que suponga un problema computacional.

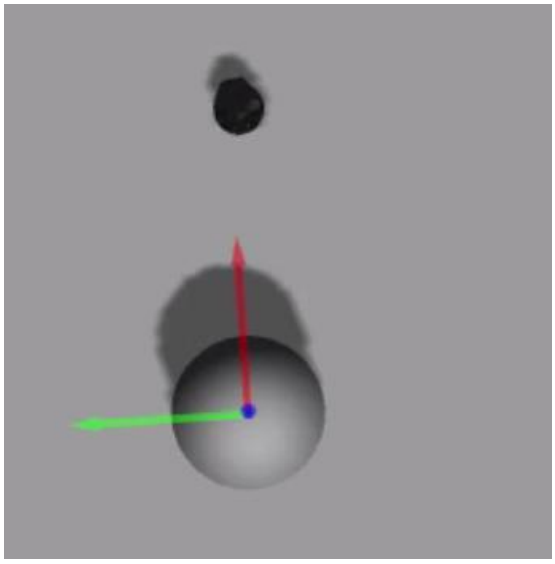

Figura 5-16. Recorrido ejemplo 2.8

# 6 CONCLUSIONES Y MÁRGENES DE AMPLIACIÓN

*Todos somos genios, pero si juzgas a un pez por su capacidad para trepar a los árboles, vivirá toda su vida pensando que es un inútil.*

*- Albert Einstein -*

A forma de resumen, se ha observado que el uso del diagrama de Voronoi para para generar trayectorias a tiempo real que eviten obstáculos presenta unos resultados muy eficientes y con un coste computacional reducido. Optimizando el cálculo del diagrama, reduciendo su uso todo lo posible y modelando los obstáculos de forma correcta se consiguen minimizar los tiempos de cálculo.

El modelado utilizado para los obstáculos muestra ciertas carencias en entornos de operación más complejos, lo cual podría mejorarse mejorando la resolución de la matriz de obstáculos, aumentando el número de puntos de clave, o ampliando el rango angular de detección del sensor, permitiendo un mayor conocimiento del entorno.

En cuando al guiado, se podrían tener en cuenta más casos de actuación para el sistema, como pudiera ser el no encontrar un camino que seguir, en cuyo caso el vehículo debería parar inmediatamente y realizar un giro constante en una determinada dirección para intentar buscar una salida.

La conexión Matlab-ROS a través del toolbox de Matlab ofrece unas amplias posibilidades para trabajar con sistemas en tiempo real, siendo muy sencilla de implementar y con muy buenos resultados. El toolbox también permite un conexionado con ROS a través de *servicios* en lugar del uso de *topics,* que aunque su desarrollo requiere un nivel avanzado de programación y un amplio conocimiento de tanto Matlab como ROS, podría ser una gran mejora en la sencillez y simplicidad del conexionado, permitiendo una sincronización total entre ambos entornos, ya que el uso de *topics* está diseñado para el envío continuo de información.Para casos como este en los que el conexionado se produce únicamente en el momento necesario y ROS debe esperar la respuesta de Matlab se hace muy enrevesado el uso de *topics*, y el conexionado con *servicios* se perfila como ideal.

El uso del algoritmo Dijsktra ha mostrado unos tiempos de computación reducidos, por lo que en este aspecto no habría necesidad de utilizar otro algoritmo de búsqueda del camino más corto. A pesar de esto, en una futura mejora del sistema, si se amplían el número de puntos del grafo al cambiar las características del sensor o del modelado de obstáculos podría mostrar una excesiva latencia en el cálculo y sería conveniente utilizar algoritmos con costes computacionales más reducidos como A* o Theta*.

La herramienta Gazebo es muy útil evaluar los algoritmos implementados y permite reducir el tiempo de desarrollo de nuevos sistemas sin la necesidad de probar los algoritmos en sistemas reales. Sin embargo para observar estados intermedios del algoritmo, como puede ser el modelado de obstáculos o la trayectoria calculada, el uso de la herramienta Rviz permitiría estas opciones ya través del conexionado de los datos del sistema con Rviz a través de *topics*.

Con las mejoras sugeridas anteriormente, el sistema estaría listo para unas posibles pruebas con un Turtlebot real, ya que todo el sistema está preparado para la implementación directa en una plataforma real, teniendo en cuenta la necesidad de un sistema de posicionamiento del vehículo en el entorno, para lo que podría utilizarse un GPS o GPS diferencial para entornos al aire libre, o sistemas de posicionamiento por balizas de posición, cámaras, o emisores y receptores de luz infrarroja para interiores.

# REFERENCIAS

[1] C. Goerzen, Z. Kong, and B. Mettler, "A survey of motion planning algorithms from the perspective of autonomous uav guidance," Journal of Intelligent and Robotic Systems, vol. 57, no. 1, pp. 65-100, 2010.

[2] Conde, R., Alejo, D., Cobano, J.A., Viguria, A., and Ollero, A. "Conflict detection and resolution method for cooperating unmanned aerial vehicles", Journal of Intelligent & Robotic Systems, 2012.

[3] Prasanna, H.M., Ghosey, D., Bhat, M.S., Bhattacharyya, C., and Umakant, J., "Interpolation-aware trajectory optimization for a hypersonic vehicle using nonlinear programming", In AIAA Guidance, Navigation, and Control Conference and Exhibit, San Francisco, USA, 2005.

[4] Vela, A., Solak, S., Singhose, W., and Clarke, J.P., "A mixed integer program for flight-level assignment and speed control for conflict resolution", In Proceedings of the 48th IEEE Conference on Decision and Control, 5219 –5226, 2009.

[5] Pallottino, L., Feron, E., and Bicchi, A., "Conflict resolution problems for air traffic management systems solved with mixed integer programming", Intelligent Transportation Systems, IEEE Transactions on, 3(1), 3– 11, 2002.

[6] Geiger, B., "Unmanned Aerial Vehicle Trajectory Planning with direct methods", Ph.D. thesis, The Pennsylvania State University, Pennsylvania, USA, 2009.

[7] Alejo, D., Cobano, J. A., Heredia, G., and Ollero, A., "Conflict-free 4D Trajectory Planning in Unmanned Aerial Vehicles for Assembly and Structure Construction", Journal Intelligent and Robotic Systems, Vol. 73, pp. 783-795, 2014.

[8] Lavalle, "S.M., Rapidly-exploring random trees: A new tool for path planning", In Computer Science Dept, Iowa State University, Tech. Rep. TR: 9811, 1998.

[9] Xue, M. y Atkins, E.M., "Terminal area trajectory optimization using simulated annealing", In 44th AIAA Aerospace Sciences Meeting and Exhibit. Reno, Nevada, USA, 2006.

[10] Durand, N. and Alliot, J., "Ant colony optimization for air traffic conflict resolution", In Proceedings of the Eighth USA/Europe Air Traffic Management Research and Development Seminar (ATM2009). Napa, (CA, USA), 2009.

[11] Foskey, .M, Garber, M., Lin, M., and Manocha, D., "A voronoi-based hybrid motion planner", In Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS), 2001.

[12] Rao, N., Stoltzfus, N., and Iyengar, S. S., "A retraction method for learned navigation in unknown terrains for a circular robot", IEEE Transactions on Robotics and Automation, 7(5), October 1991.

[13] S. Zheng, A. Janecek, J. Li, and T. Ying, "Dynamic search in fireworks algorithm," in Proc. 2014 IEEE Congress on Evolutionary computation, Beijing, China, 6-11 July 2014, pp. 3222–3229.

[14] S. Karaman and E. Frazzoli, "Sampling-based algorithms for optimal motion planning," International Journal of Robotics Research, vol. 30, 2011.

[15] J. K. Kuchar and L. C. Yang, "A review of conflict detection and resolution modeling methods," IEEE Transactions on Intelligent Transportation Systems, vol. 1, pp. 179–189, 2000.

[16] Wikipedia. https://es.wikipedia.org/wiki/Grafo. [En línea]

[17] Ferran Hurtado, Belen Palop y Vera Sacristian, "Diagramas de Voronoi con funciones temporales," Universitat Politecnica de Catalunya, Barcelona, 1996.