DISEÑO Y SIMULACIÓN DE UN ALGORITMO DE RECONOCIMIENTO DE UN ENTORNO ROBÓTICO BASADO EN DESCOMPOSICIÓN DEL PLANO EN TESELACIONES USANDO FRENTES DE ONDA CON TRIÁNGULOS

OSCAR FERNANDO PENAGOS ESPINEL

FUNDACIÓN UNIVERSITARIA LOS LIBERTADORES
FACULTAD DE INGENIERÍA
INGENIERÍA ELECTRÓNICA
BOGOTÁ DC
2017

DISEÑO Y SIMULACIÓN DE UN ALGORITMO DE RECONOCIMIENTO DE UN ENTORNO ROBÓTICO BASADO EN DESCOMPOSICIÓN DEL PLANO EN TESELACIONES USANDO FRENTES DE ONDA CON TRIÁNGULOS

OSCAR FERNANDO PENAGOS ESPINEL

Trabajo para optar por el título de ingeniero electrónico

Director
Ivan Dario Ladino Vega

FUNDACIÓN UNIVERSITARIA LOS LIBERTADORES
FACULTAD DE INGENIERÍA
INGENIERÍA ELECTRÓNICA
BOGOTÁ DC
2017

Nota de aceptación

_____

_____

_____

_____

_____

Firma

Nombre:

Presidente del jurado

_____

Firma

Nombre:

Jurado

_____

Firma

Nombre:

Jurado

Bogotá DC, Octubre 13 de 2017

Dedico este proyecto a mi familia quienes fueron el principal apoyo que tube durante el desarrollo de mi proyecto de grado y de mi carrera en general.

AGRADECIMIENTOS

Agradezco la realización de mi proyecto de grado a la planta docente de ingeniería electrónica, en especial a los ingenieros Ivan Ladino y Brayan Sáenz quienes me orientaron para el correcto desarrollo del proyecto, además quiero agradecer al ingeniero Johan Agudelo, el cual me brindó las herramientas intelectuales necesarios para poder realizar una gran parte del proyecto.

CONTENIDO

Pág.

LISTA DE FIGURAS

Pág.

RESUMEN

En el siguiente documento se detallará la implementación de un algoritmo para el reconocimiento de un entorno y posteriormente utilizar herramientas de topología para la discretización y descomposición del espacio de configuración por medio de teselaciones. Para observar la efectividad de las teselaciones se realizó un algoritmo de búsqueda básico derivado del algoritmo A*. Por último se comparó los resultados obtenidos variando el frente de onda de las teselaciones con diferentes polígonos regulares tenienedo como puntos de referencia la cantidad de líneas utilizadas, los ciclos necesarios para la teselación del espacio de configuración, el tiempo total de ejecución y la cobertura del algoritmo.

INTRODUCCIÓN

La navegación dirigida en robótica es una problemática que actualmente cuenta con múltiples soluciones, originalmente, la navegación se realizaba a partir de algoritmos que permitían generar rutas sobre entornos continuos, algunos de estos algoritmos son A^*, $Dijkstra$ y D^*, cada uno requería diferente nivel de conocimiento del entorno, sin embargo, no garantizaban que la ruta encontrada fuera la mas eficiente o como en el caso del $Dijkstra$ no garantiza encontrar alguna ruta para cualquier entorno, esta falla se hacia cada vez mas pronunciada a medida que aumentaba la complejidad del entorno. Es por esto que fue necesario crear procedimientos que permitieran un mejor manejo del entorno, a partir de esa necesidad se crearon métodos de planificación que lograran discretizar o descomponer el entorno, algunos de los mas representativos son planificación por diagramas de Voronoi, planificación por grafos de visibilidad y planificación por descomposición en celdas, la ventaja de utilizar estos métodos es que nos permiten seguir utilizando los algoritmos de búsqueda sobre entornos que poseen una cantidad de movimientos limitada.

En [1], el autor propone utilizar herramientas de topología, los algoritmos $Dijkstra$ y A^* y el método de planificación con diagramas de Voronoi para desarrollar algoritmos de búsqueda que garanticen encontrar la ruta mas eficiente para entornos Euclidianos.

En [1], el autor utilizó los diagramas de Voronoi para descomponer el espacio de configuración de un entorno plano, utilizó el algoritmo $Dijkstra$ para generar un frente de onda que en ciertos puntos sobre el espacio de configuración cubra el entorno y limiten los obstáculos, luego utiliza el centroide de cada una de las celdas generadas por cada expansión para trazar la ruta del robot. Para encontrar la ruta mas eficiente hace uso de herramientas de topología, las mas significativas son las trayectorias homotópicas y los conjuntos homológicos.

En su investigación no solo logró encontrar la ruta mas eficientes sobre entornos empíricos abarcando los métodos mencionados y una fuerte fundamentación topológica, logró implementar algoritmos que operen sobre entornos Euclidianos en $\mathbb{R}^3$ y además generar trayectorias para agentes multi-robot.

Los métodos y algoritmos mencionados hacen énfasis en descomponer el entorno en diferentes modos y luego trazar rutas con algoritmos clásicos de búsqueda en grafos, en [1] utilizan bases de esos métodos y conceptos de topología para encontrar la ruta mas eficiente, en este artículo se proponen formas alternas para la descomposición y consideraciones adicionales a partir de este nuevo modelo. También se tendrá en cuenta el uso topología para el trazado de la ruta.

El modelo esta diseñado teniendo en cuenta una descomposición uniforme a lo largo del entorno, de este modo se garantiza que las distancias Euclidianas sobre los centros de las celdas adyacentes sean relativamente parecidas y se puede encontrar de manera mas sencilla una heurística que permita limitar la distancia del movimiento entre puntos, para ello se requiere calcular el mínimo de celdas necesarias para el correcto diseño del algoritmo de búsqueda. Las celdas obtenidas por método de diagramas de Voronoi se realiza con expansiones de frente de onda hexagonal, sin embargo, se realizaron simulaciones modificando la forma del frente de onda con diferentes polígonos regulares para comparar y seleccionar aquella que se adapte mejor sobre el entorno y a las especificaciones del algoritmo de búsqueda.

OBJETIVOS

OBJETIVO GENERAL

Implementar un algoritmo que nos permita reconocer un entorno en dos dimensiones con obstáculos aleatorios, para que en un futuro cercano pueda ser utilizado para el reconocimiento de la ruta de menor coste entre dos puntos.

OBJETIVOS ESPECÍFICOS

- Definir el número de teselaciones de acuerdo al tamaño del entorno, la cantidad de obstáculos y el área total que ocupan.
- Comparar la eficiencia del algoritmo de descomposición utilizando teselaciones con diferentes polígonos regulares.
- Encontrar un punto representativo para cada teselación tal que por cada una exista un solo posible estado del robot.

1. ALGORITMO DE BÚSQUEDA GRÁFICA

En una aproximación discreta sobre una ruta de planificación, una gráfica es construida por la discretización del espacio de configuración y el posicionamiento de un vértice / nodo para cada celda discretizada. Los borden son establecidos sobre la base de acciones disponibles entre vértices vecinos.

Así una gráfica consiste en tres componentes : Un conjunto de vértices $V(G)$, un conjunto de bordes $\varepsilon(G) \subseteq V(G)\ x\ V(G)$, y una función de costo $C_G : \varepsilon(G) \longrightarrow \mathbb{R}^+$. Un elemento en $V(G)$ es llamado un vértice o un nodo. Un elemento en $\varepsilon(G)$ está representado por el par ordenado $[a, b] \in \varepsilon(G)$, el cual implica que ahí exista un borde en G conectando $a \in V(G)$ a $b \in V(G)$.

1.1. ALGORITMO DE DIJKSTRA

El algoritmo de Dijkstra [3] es el mas fundamental en encontrar un costo mínimo (o la distancia mínima) de trayectorias a través de un vértice, $p \in V(G)$, en la gráfica a cualquier otro vértice es accesible desde p. Esto es garantizado para ser óptimo. La intuición detrás del algoritmo de Dijkstra es que empieza desde el inicio de un nodo p, un 'frente de honda' de casi el mismo tamaño que la distancia geodésica (costo del camino mas corto) es propagada a través de la gráfica. La Figura 1.2 ilustra el proceso del algoritmo de Dijkstra. Observe que el frente de onda está marcado por los círculos azules vacíos. Esto es un conjunto de nodos $[u \in Q|g(u)$ es infinito].

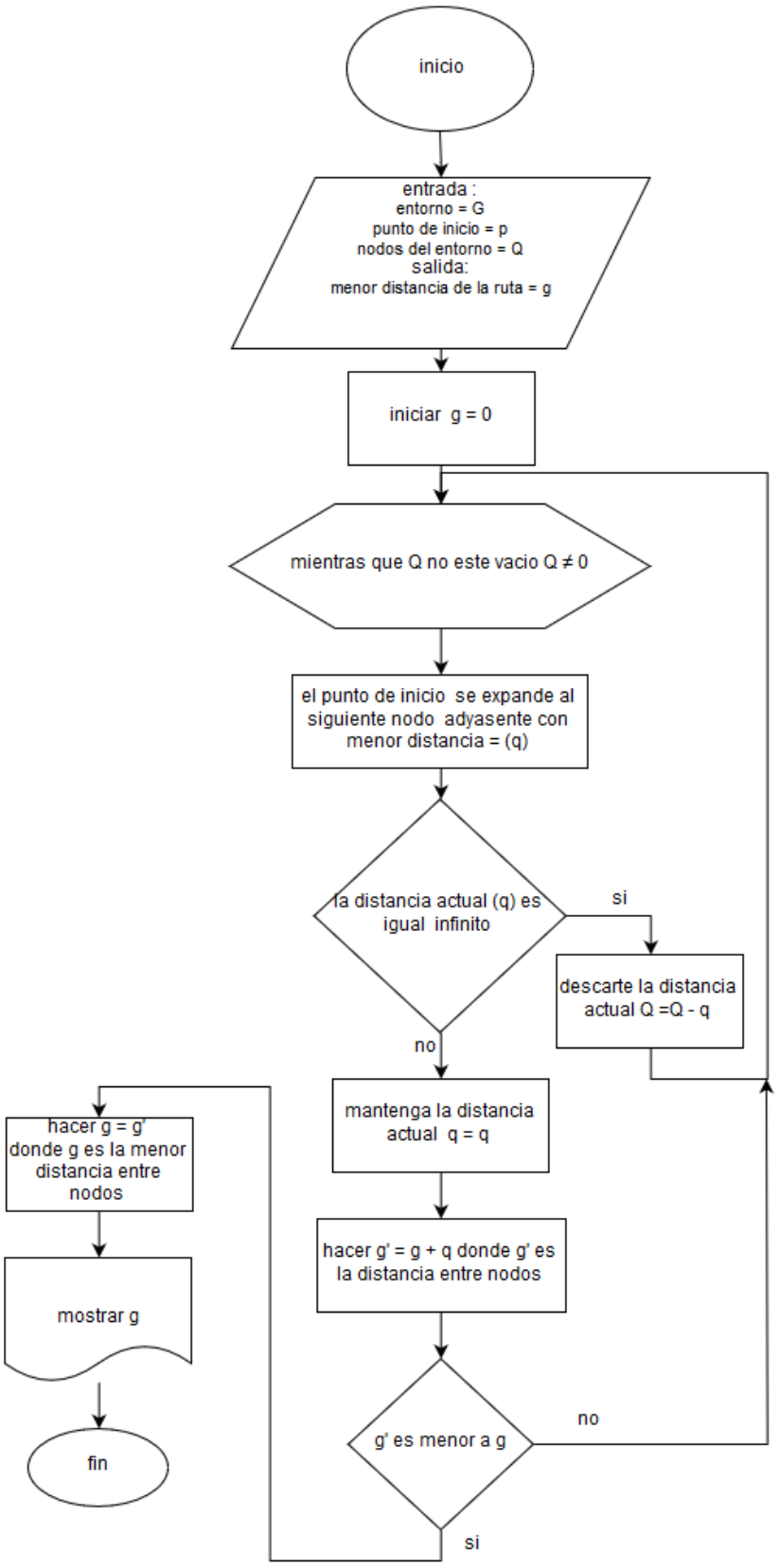

Figura 1.1: Representación en diagramas de flujo del algoritmo de Dijkstra. Basada en [1] pág. 35 y modificada por el autor.

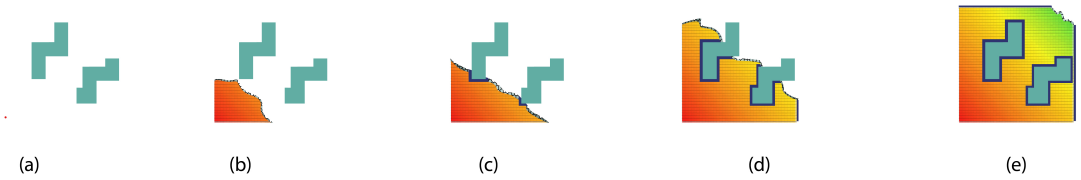

Figura 1.2: Ilustración del proceso del algoritmo de Dijkstra. Nodo de inicio en rojo. Basada en [1] pág. 36 y modificada por el autor.

(a) $iter = 0$.

(b) $iter = 0.25$.

(c) $iter = 0.5$.

(d) $iter = 0.75$.

(e) $iter = 1$.

1.2. ALGORITMO A^*

El algoritmo A^* [4] es, en esencia, muy similar al algoritmo Dijkstra. Mientras que con el algoritmo Dijkstra uno típicamente está interesado en encontrar caminos de costo bajo para cualquier vértice en la gráfica desde un vértice inicial, en A^* se busca un objetivo fijo. Esto nos permite dirigir la búsqueda de una manera más informada. En lugar de expandir el 'frente de onda' (mencionado anteriormente) uniformemente en todas las direcciones, lo expandimos con predilección al vértice seleccionado como el objetivo (Figura 1.3). Esta parcialidad está regida por la función heurística. Una función heurística para una gráfica G es una función $h : V(G)xV(G) \to \mathbb{R}^+$, tal que $h(v_a, v_b)$ nos de algunas estimaciones de la mínima distancia (costo total del camino mas corto) entre los vértices v_a y v_b. Una función heurística admisible es aquella función heurística que siempre subestima el valor del costo actual. Esto puede mostrarse en el algoritmo A^* el cual es un heurística admisible que retorna (menos costo) al camino del objetivo. Desde luego, la función constante $h(v_a, v_b) = 0$ es un heurístico admisible, y en este caso puede mostrarse que el algoritmo A^* se convierte en el equivalente del algoritmo Dijkstra. De hecho, la diferencia mas grande entre del algoritmo A^* al algoritmo Dijkstra (junto a otras pocas diferencias estructurales) es que uno selecciona el vértice q a ser expandido como el unico con valor mas bajo de f en lugar de g, donde el valor f es el valor g además de la heurística desde el vértice hasta el objetivo.

Función Heurística

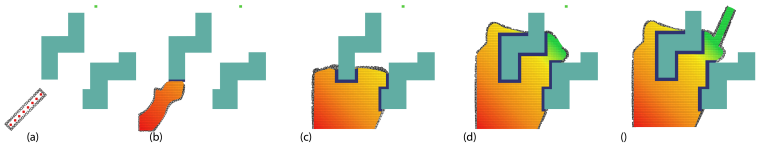

Figura 1.3: Ilustración del proceso del algoritmo A^*. El conjunto abierto esta marcado por los círculos azules vacíos.

(a) $iter = 0$.

(b) $iter = 0.25$.

(c) $iter = 0.5$.

(d) $iter = 0.75$.

(e) $iter = 1$.

La elección de la función heurística, h, es extremadamente crucial en una búsqueda A^*. Una función heurística es una función escalar positiva de los vértices en la búsqueda gráfica. Para que A^* retorne a la solución optima, la función heurística necesita ser admisible - esto se debe, a que esta no debe sobre estimar el valor de costo actual para la búsqueda del objetivo. Ademas hacer que la búsqueda sea mas eficiente y reducir el número de vértices expandidos, la heurística debe ser lo mas cercano posible al costo mínimo actual al objetivo. Para una gráfica construida por la discretización de un espacio Euclidiano, una función heurística obvia y de uso común es la distancia Euclidiana al objetivo, $h_E(u, v) = ||u - v||_2$ (donde, z representa la coordenada del vértice, z, en la configuración espacial original). Pero, para tipos particulares de discretización un puede usar funciones heurísticas con límites mas estrechos. Por ejemplo, par aun gráfico de red de ocho conexiones (uno creado por la discretización cuadrada uniforme de un plano como en la Figura 1.5), uno puede usar mas eficiencia heurística dado por $h_8(u, v) = \sqrt{2}min(\triangle x, \triangle y) + |\triangle x - \triangle y|$, donde $\triangle x = u_x - v_x|$ y $\triangle y = |u_y v_y|$, con z_x y z_y representando respectivamente a las coordenadas x y y de los puntos z en las coordenadas espaciales originales [8].

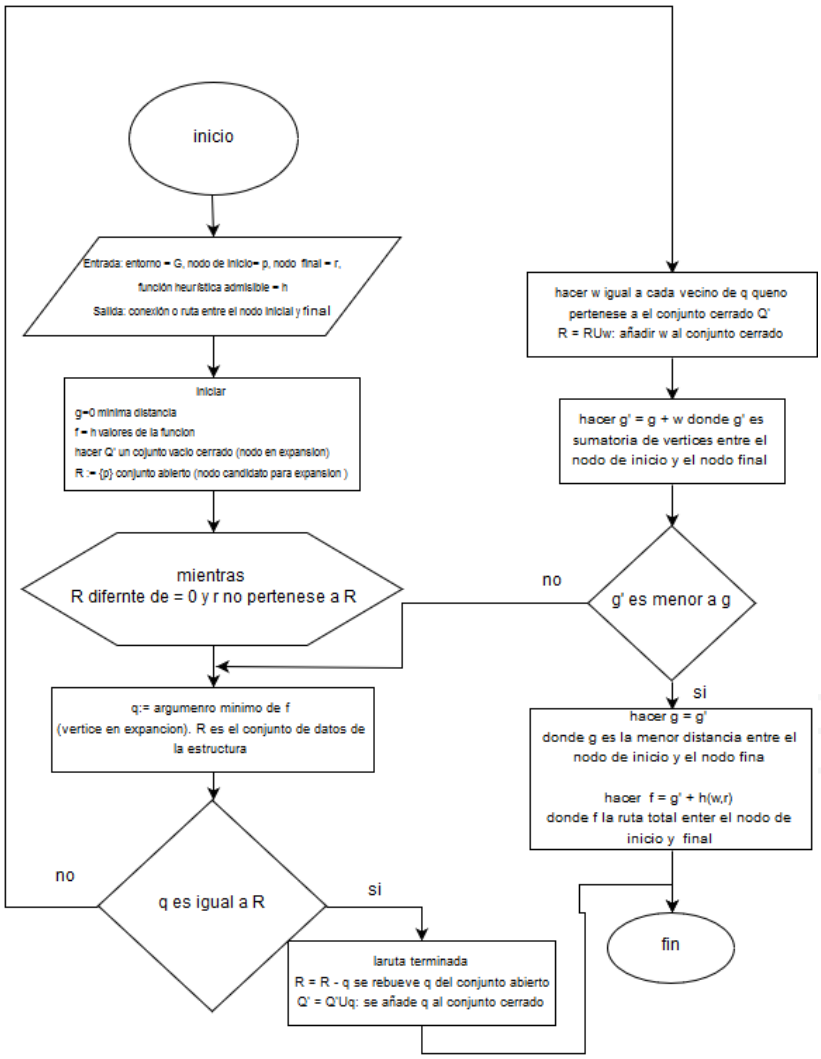

Figura 1.14: Representación en diagramas de flujo del algoritmo de A^*.

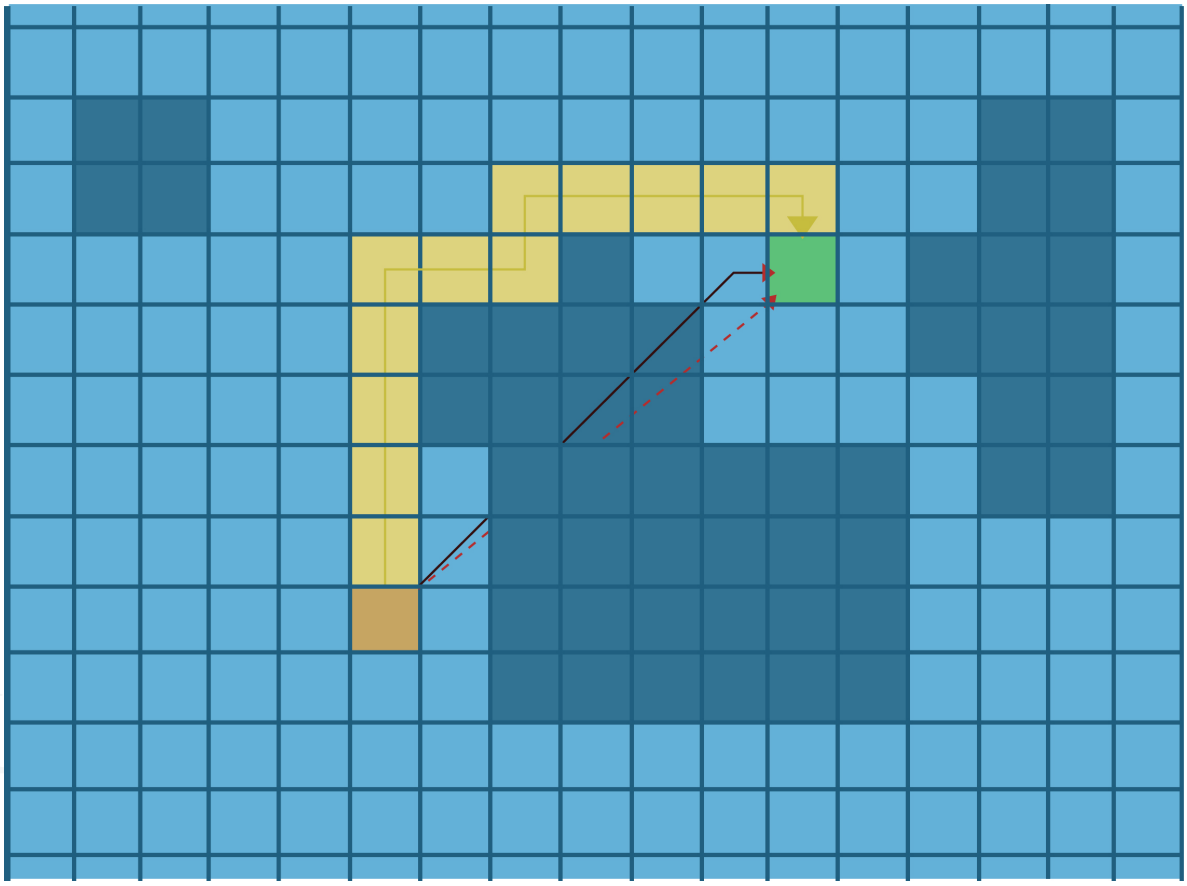

Figura 1.5: Los dos diferentes tipos de heurísticas para un gráfico de una red de ocho conexiones : La parte naranja es el estado de inicio y la verde el vértice objetivo. La línea discontinua representa h_E, la línea negra representa h_s, el verde claro representa la trayectoria real de menor costo. Los obstáculos están representados por el azul oscuro. Basada en [1] pág. 12 y modificada por el autor. Basada en [1] pág. 37 y modificada por el autor.

2. COBERTURA Y EXPLORACIÓN USANDO MÉTODOS BASADOS EN LA BÚSQUEDA

La métrica en un espacio de configuración de un robot puede ser inducido por criterios de diferentes problemas. anteriormente la métrica era de menor importancia, por la utilización de un algoritmo de búsqueda óptima (algoritmo A^*), hicimos uso de la métrica en un espacio de configuración. La mayoría de las veces usamos la restricción de la métrica plana Euclidiana a la gráfica para determinar la longitud de una trayectoria. Además, el concepto de métrica es también fundamental para muchas coberturas y exploración [10, 11] en problemas de robots. Una aproximación hacia la solución de problemas de coberturas con n involucra la partición del espacio de configuración en n *mosaicos* (una partición simple conectada). En particular, esto requiere un mosaico de Voronoi. Si bien tal mosaico es fácil de lograr en un entorno convexo con métrica Euclidiana, se incrementa significativamente su dificultad en entornos con obstáculos y métricas no Euclidianas. Pero la mayoría de configuraciones en robots no son convexas debido a la presencia de obstáculos, y la exploración de problemas, como se discutirá mas adelante, la métrica Euclidiana es central. En este capítulo trataremos de desarrollar ciertas herramientas para hacer frente a estos problemas [9].

El problema de encontrar una buena cobertura de un entorno es fundamental en ls problemas prácticos con multi-robots. Un enfoque común de control de cobertura, que es eficiente y puede ser implementado en una forma distributiva, es a través de las leyes de control de realimentación con respecto a los centroides de las celdas de Voronoi resultantes de los mosaicos de Voronoi de un entorno. En [2], los autores proponen unas leyes para el control el robot basado en un gradiente individual descendiente que garantice una cobertura óptima de un entorno convexo dado por una función de densidad la cual representa la distribución de la cobertura deseada. Los autores de [6] construye sobre esta idea u desarrolla las leyes de control enfocado en la posición de una red de sensores óptimos con respecto a una distribución de probabilidad conocida. En [7], está aproximación se extiende a los controladores estrechamente óptimo que no requieren conocimiento previo de la distribución de cobertura deseada. Para trabajar bajo esta condición de requerir un ambiente convexo, los autores de [5] proponen el uso de *mosaicos geodésicos de Voronoi* determinados por la distancia geodésica en lugar de la distancia Euclidiana. Por lo tanto tal método involucra una gran cantidad de cálculos geométricos y trabajo

únicamente para los entornos con obstáculos poligonales. Además, también nos gustaría ser capaces de resolver el problema de cobertura con métrica no Euclidiana intrínseca al espacio de configuración. Nosotros usaremos las aproximaciones basadas en búsqueda gráfica para desarrollar herramientas para resolver el problema de cobertura en entornos no convexos con métrica no Euclidiana.

Equipados con esto, por el final del capítulo consideraremos el siguiente escenario: Un equipo de robots entran a un entorno no convexo desconocido. Los robots se deben controlar para explorar el entorno para la construcción del mapa y converger en una formación en el mapa que disperse los robots a ciertas localizaciones que les permitan ocupar actividades como la vigilancia persistente. Esta descripción se presta para una amplia clase de aplicaciones robóticas. Por lo tanto nos enfocaremos esencialmente en componentes hacia este escenario: el desarrollo de robot individuales descentralizada leyes de control basado en estimaciones inciertas del entorno que dirijan el equipo de robots para explorar el entorno y su cobertura. Usaremos un algoritmo de implementación distribuida para el cálculo de mosaicos de Voronoi generalizados de entornos no convexos (usando una representación discreta) en tiempo real para leyes de control con realimentación. Por lo tanto usaremos una medición basa en la entropía que le permite operar al control de cobertura en entornos desconocidos no convexos y al mismo tiempo lograr la exploración a través de ganancia de información.

2.1. COBERTURA FUNCIONAL, MOSAICO DE VORONOI EN TIEMPO CONTINUO

Suponga n robots móviles en el entorno, y en la posición particular del i-ésimo robot está representada por $\mathbf{p}_i \in \Omega$ y el mosaico asociado con el por $W_i, \forall\, i = 1, 2, ..., n$. Por definición, los mosaicos son tales que $Int(W_i) \cap Int(W_j) = 0$, $\forall\, i \neq j$, y $\cup_{i=1}^{n} W_i = \Omega$. Dado un conjunto de posiciones de un robot $P = \{\mathbf{p}_1, \mathbf{p}_2, ..., \mathbf{p}_n\}$ y los mosaicos $W = \{W_1, W_2, ..., W_n\}$ tal que $\mathbf{p}_i \in W_i, \forall\, i = 1, 3, ..., n$, la *cobertura funcional* está definida como:

$$\mathcal{H}(P, W) = \sum_{i=1}^{n} \mathcal{H}(\mathbf{p}_i, W_i) = \sum_{i=1}^{n} \int_{W_i} f_i(\mathrm{d}(\mathbf{q}, \mathbf{p}_i))\phi(\mathbf{q})\mathrm{d}\mathbf{q}$$

donde $f_i : \mathbb{R} \to \mathbb{R}$ son funciones suaves y estrictamente crecientes, $\phi : \Omega \to \mathbb{R}$ es una función de peso o densidad, y d$\mathbf{q}$ representa el volumen o el área de un elemento infinitesimal.

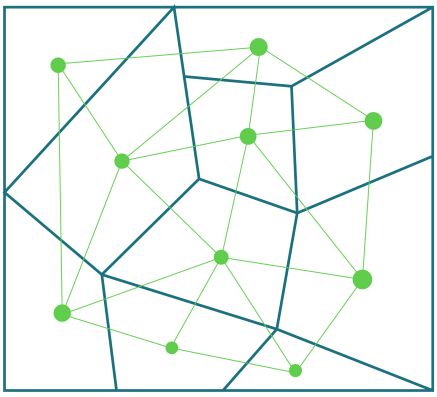

Figura 2.1: Mosaico de Voronoi de un convexo Ω (región rectangular) con $n = 10$ robots (círculos verdes). Los segmentos de línea rojos muestran los límites del mosaico. Note como un segmento del límite es el bisector perpendicular de la línea verde uniendo las divisiones de los segmentos de los límites. Basada en [1] pág. 37 y modificada por el autor. Basada en [1] pág. 97 y modificada por el autor.

El nombre *cobertura funcional* está indicado por el hecho que $\mathcal{H}$ mide que tan *mala* es la cobertura. Por lo cual, sea P el conjunto de todas las posiciones iniciales del robot, el objetivo final del algoritmo es idear una ley de control que minimice la función $\widetilde{\mathcal{H}}(P) := \overset{\min}{W}$ $\mathcal{H}(P, W)$ (el mejor valor de $\mathcal{H}(P, W)$ para un P determinado). Es fácil de mostrar [5, 12] que $\widetilde{\mathcal{H}}(P) = \mathcal{H}(P, V)$, donde $V = \{V_1, V_2, ..., V_n\}$ es el mosaico de Voronoi dado por

$$V_i = \{\mathbf{q} \in \Omega \mid f_i(d(\mathbf{q}, \mathbf{p}_i)) \leq f_j(d(\mathbf{q}, \mathbf{p}_j)), \forall\, j \neq i\}$$

Siempre que $\mathbf{p}_i \in Int(\Omega)$, la ley de control para minimizar $\widetilde{\mathcal{H}}(P) = \sum_{i=1}^{n} \int_{V_i} f_i(d(\mathbf{q}, \mathbf{p}_i))\phi(\mathbf{q})\mathrm{d}\mathbf{q}$ puede ser reducido en un problema de seguir su gradiente. A pesar de que V_i son funciones de P, puede ser mostrado usando métodos de diferenciación usando integración bajo que

$$\frac{\partial \widetilde{\mathcal{H}}(P)}{\partial \mathbf{p}_i} = \int_{V_i} \frac{\partial}{\partial \mathbf{p}_i} f_i(d(\mathbf{q}, \mathbf{p}_i))\phi(\mathbf{q}\mathrm{d}\mathbf{q})$$

Generalmente uno escoge un $f_i(x) = x^2$ para implementaciones mas prácticas. Sin embargo, una variación del problema para tener en cuenta son los sensores de huellas finitos del robots, construyendo un *mosaico de poder de Voronoi*, en el cual uno selecciona $f_i(x) = x^2 - R_j^2$, donde R_i es el radio de la huella del sensor del i-ésimo robot.

Hasta ahora no hemos hecho una suposición principal sobre la función de distancia d. De este modo si el espacio Ω es convexo, y la métrica η es plano (Euclidiano), entonces d es una distancia Euclidiana dada por $d(\mathbf{x}, \mathbf{y}) = ||\mathbf{x} - \mathbf{y}||_2$. Bajo esta consideración y usando la forma de f_i discutida, la ecuación anterior puede ser simplificada para obtener

$$\frac{\partial \widetilde{\mathcal{H}}(P)}{\partial \mathbf{p}_i} = 2(\mathbf{p}_i - \mathbf{p}_i^*)$$

donde, $\mathbf{p}_i^* = \frac{\int_{V_i} \mathbf{q}\phi(\mathbf{q})\mathrm{d}\mathbf{q}}{\int_{V_i} \phi(\mathbf{q})\mathrm{d}\mathbf{q}}$, el centroide ponderado de V_i. De este modo, la función de distancia Euclidiana realiza el cálculo del mosaico de Voronoi -V- muy sencillo, puede ser construida a partir de las mediatrices de los segmentos de línea $\overline{\mathbf{p}_i\mathbf{p}_j}$, $\forall\, i \neq j$, con lo que cada V_o un polígono convexo (Figura 2.1). Esto también permite de forma aproximada el cálculo del centroide, $\mathbf{p}_i^*$ cuando la función de peso ϕ es uniforme.

2.2. BÚSQUEDA GRÁFICA BASADA EN EL MOSAICO DE VORONOI

La idea principal es hacer una modificación básica al algoritmo de Dijkstra. Para la creación de los mosaicos de Voronoi iniciamos el *conjunto abierto* con múltiples nodos de salida de los cuales iniciamos la propagación de los frentes de onda. De este modo los frentes de onda se emanan de múltiples fuentes. Los lugares donde los frentes de onda chocan representarán los límites de los mosaicos de Voronoi. En adición, podemos convenientemente alterar la función de distancia, el nivel del conjunto el cual representa los límites de los mosaicos. Esto nos permite crear un *mosaico de poder geodésico de Voronoi*. La Figura 2.2 ilustra el proceso del algoritmo. Mas ejemplos de los mosaicos creados usando el algoritmo son ilustrados en la Figura 2.2 [9].

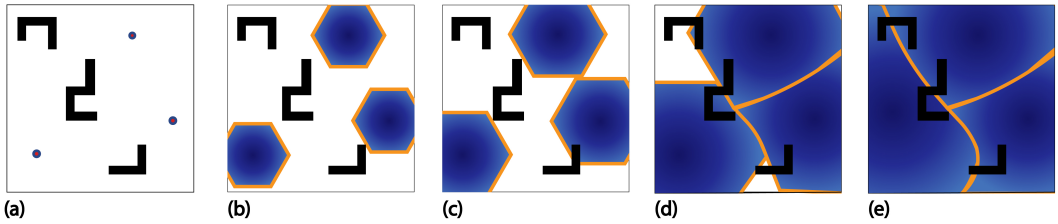

Figura 2.2: Ilustración del progreso del algoritmo de de mosaicos básico en un entorno con varios obstáculos. El área rellena indica el conjunto de los vértices expandidos (complemento de Q). Los límites de los mosaicos son de color naranja. La gráfica está construida por un la discretización de un entorno cuadrado.

a. iter = 0
b. iter = 0.25
c. iter = 0.5
d. iter = 0.75
e. iter = 1

3. IMPLEMENTACIÓN

A continuación se detallará la descripción del algoritmo por etapas a través del software Matlab, cada una de estas etapas esta acompañada de la descripción matemáticas, el pseudocódigo del algoritmo implementado, y una imagen del proceso realizado sobre el entorno.

La implementación del algoritmo que nos permita la teselación de un entorno aleatorio se realizo a través de cinco etapas:

- Creación de un entorno aleatorio
- Reconocimiento del número óptimo de teselaciones
- Expansión en frente de onda con polígonos regulares
- Cálculo del centroide de cada teselación
- Prueba con un algoritmo de búsqueda

El primer paso es la creación del entorno. Se definió las dimensiones del entorno y a partir de geometría Euclidiana se generó de manera aleatoria la cantidad de obstáculos, el tamaño de cada uno, la posición en el entorno y la forma. En la Figura 3.1 se puede observar un entorno generado por el algoritmo, donde las áreas blancas equivalen a los obstáculos y el área negra al espacio de configuración. En el Algoritmo 1 se encuentra el proceso utilizado.

Algoritmo 1 - Creación del entorno.

REQUIRE ENTRADA Q = Número de obstáculos
REQUIRE ENTRADA P(x,y) = Posición del obstáculo
REQUIRE ENTRADA T(1,Q) = Tamaño del obstáculo
REQUIRE ENTRADA O = Contador actual de obstáculos
REQUIRE SALIDA I = Imagen de salida

WHILE $O \neq Q$
STATE Posición = P

STATE Definir forma aleatoria = F
FOR $i \leftarrow 1$ to $T(1, Q)$
STATE Definir Contorno según F
STATE $Contorno = 255$
ENDFOR
STATE O = O + 1;
ENDWHILE
RETURN TRUE

El segundo paso corresponde al análisis y reconocimiento del número óptimo de teselaciones, el objetivo de esta etapa es determinar el menor número posible de teselaciones para la correcta discretización del entono; para ello se utilizó dos criterios a partir de la lectura del entorno, los cuales son la cantidad total de obstáculos y el área total que ocupan sobre el entorno. Teniendo en cuenta de estos dos datos se puede calcular la media y la covarianza, luego por medio de un promedio entre estos datos encontramos el valor mínimo necesario. El número encontrado según el cálculo posee una raíz cuadrada entera, esto se debe a que la posición de los 0-simpliciales por los cuales se expandirá el frente de onda corresponde a una cuadrícula sobre el entorno con el mismo número de 0-simpliciales por filas y por columnas. En el algoritmo 2 se encuentran los pasos realizados mediante un pseudocódigo.

Algoritmo 2 - Número de teselaciones.

REQUIRE ENTRADA I = Imagen de entrada
REQUIRE U = Vector para definir el área de cada obstáculo
REQUIRE G = Área total ocupada por los obstáculos
REQUIRE E = Número total de pixeles de la imagen del entorno
REQUIRE SALIDA D = Cantidad de obstáculos
REQUIRE SALIDA X = Número de teselaciones

STATE Conteo y etiquetado de obstáculos a través de la función bwlabel(I)
STATE D = bwlabel(I)
FOR $i \leftarrow 1$ to E
IF $I(i) = E$
STATE $U(i) = U(i) + 1$
ENDIF

ENDFOR
STATE Cálculo de la media y la covarianza a partir de U y D
STATE $X = (covarianza + media + D)/3$
RETURN TRUE

En el tercer paso se realizó la teselación del espacio de configuración. Primero se determinó la posición exacta de los 0-simpliciales, de los cuales se expandirá los frente de onda con forma de triángulo. Cada frente de onda es una variedad en expansión que se detendrá hasta que el espacio de configuración se encuentre totalmente teselado. Adicionalmente hay que tener en cuenta que la teselación se expandirá por frente de onda si el punto de origen se encuentra sobre el espacio de configuración. En la Figura 3.2, 3.3 y 3.4 se puede observar la teselación de un entorno para diferentes número de iteraciones (donde iter = 0 indica el origen de las iteraciones e iter = 1 indica que el plano se encuentra totalmente teselado). El Algoritmo 3 describe el proceso para la teselación del entorno.

Algoritmo 3 - Teselado del entorno.

REQUIRE ENTRADA X = Número de teselaciones
REQUIRE ENTRADA Y = Posición de cada teselación
REQUIRE ENTRADA G(1,X) = Tonalidad de cada teselación (Escala de grises)
REQUIRE ENTRADA Z = Número de puntos por expandir
REQUIRE ENTRADA I = Imagen de entrada
REQUIRE V = Número de puntos expandidos
REQUIRE W = Contador actual de teselaciones
REQUIRE SALIDA J = Imagen de salida

WHILE $Z \neq V$
WHILE $X \neq W$
STATE Expandir frente de onda en una posición
IF $I(Y) = 0$ AND $J(Y) = 0$
STATE $J(Y) = G(1, W)$
STATE $V = V + 1$;
ENDIF
STATE $W = W + 1$
ENDWHILE

ENDWHILE
RETURN TRUE

El cuarto paso se basa en hallar los centroides de cada teselación, para realizar este paso elegimos una teselación y modificamos la imagen para eliminar los puntos conexos a la variedad sobre el espacio de configuración, de este modo con un producto punto se calcula el centro de masa para una teselación a la vez. En la Figura 3.5 se encuentra un entorno teselado con sus respectivos centroides. En el Algoritmo 4 se encuentra el proceso utilizado.

Algoritmo 1 - Cálculo de centroides.

REQUIRE ENTRADA X = Número de teselaciones
REQUIRE ENTRADA J = Imagen de entrada
REQUIRE ENTRADA F = Cantidad de filas de J
REQUIRE ENTRADA C = Cantidad de columnas de J
REQUIRE ENTRADA G(1,X) = Tonalidad de cada teselación (Escala de grises)
REQUIRE H = Imagen de transición
REQUIRE W = Contador actual de teselaciones
REQUIRE S = Matriz de unos para el cálculo del centroide
REQUIRE SALIDA K = Imagen de salida
REQUIRE SALIDA C(x,y) = Posición del centroide de cada teselación

WHILE $W \neq X$
FOR$i \leftarrow 1$ to F
FOR$j \leftarrow 1$ to C
IF $J(Y) = G(1, W)$
STATE$H(i, j) = 255$
ENDIF
ENDFOR
ENDFOR
FOR$i \leftarrow 1$ to F
STATE $M1 = i \times (H \bullet S) + M1$
STATE $M2 = (H \bullet S) + M2$
ENDFOR
FOR$j \leftarrow 1$ to C

STATE $M3 = j \times (H \bullet S) + M3$
STATE $M4 = (H \bullet S) + M4$
ENDFOR
STATE $C(x) = M1/M2$
STATE $C(y) = M3/M4$
STATE $W = W + 1$
ENDWHILE
RETURN TRUE

El quinto y último paso es la implementación de un algoritmo de búsqueda para observar el desempeño de las teselaciones. Los puntos sobre los cuales el algoritmo se podrá desplazar son los centroides. Teniendo en cuenta que cualquier trayectoria homotópica también es homológica y ademas que la trayectoria homológica me define los límites de las variedades que puedo recorrer, el primer paso fue la construcción de los grupos de trayectorias homológicas. Una vez obtenido los grupos, observamos cual recorre una distancia Euclidiana menor en el total de su trayectoria y adicionalmente cual es la pendiente realizada a lo largo de su desplazamiento. Luego de reconocer el grupo homológico que mas nos conviene, construimos un grupo de trayectorias homotópicas sobre esa variedad, y realizamos búsqueda para encontrar la trayectoria homotópica mas conveniente. En el Algoritmo 5 se encuentra la implementación del algoritmo de búsqueda. En las Figuras 3.6 y 3.7 se observa el algoritmo de búsqueda entre dos puntos, la línea verde representa el recorrido del robot, la Figura 3.6 se encuentra sobre un entorno con 13 teselaciones y la Figura 3.7 sobre un entorno con 1400 teselaciones.

Nota: El algoritmo de búsqueda implementado no es totalmente funcional como lo descrito anteriormente, únicamente está calculando una ruta homotópica eficiente dentro de la variedad con menor distancia Euclidiana al objetivo. En la Figura 3.7 se observa el algoritmo de búsqueda implementado.

Algoritmo 5 - Algoritmo de búsqueda.

ENTRADA K = Imagen de entrada
ENTRADA C = Posición de los centroides
ENTRADA P = Posición del punto de origen
ENTRADA D = Posición del punto de destino

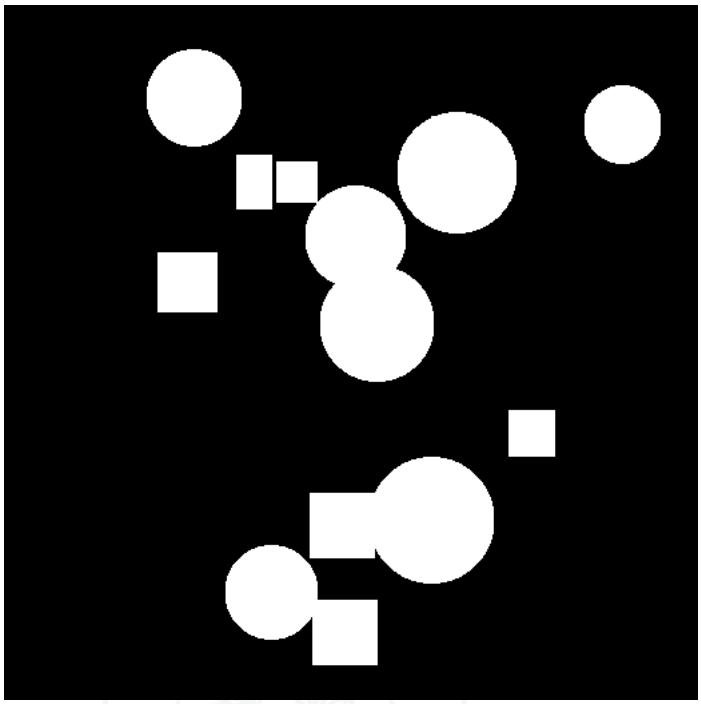

Figura 3.1: Entorno aleatorio creado por el algoritmo 1.

SALIDA L = Imagen de salida

WHILE $W \neq X$
FOR $i \leftarrow 1$ to F
FOR$j \leftarrow 1$ to C
IF $J(Y) = G(1, W)$
STATE $H(i, j) = 255$
ENDIF
ENDFOR
ENDFOR
STATE $W = W + 1$
ENDWHILE
RETURN TRUE

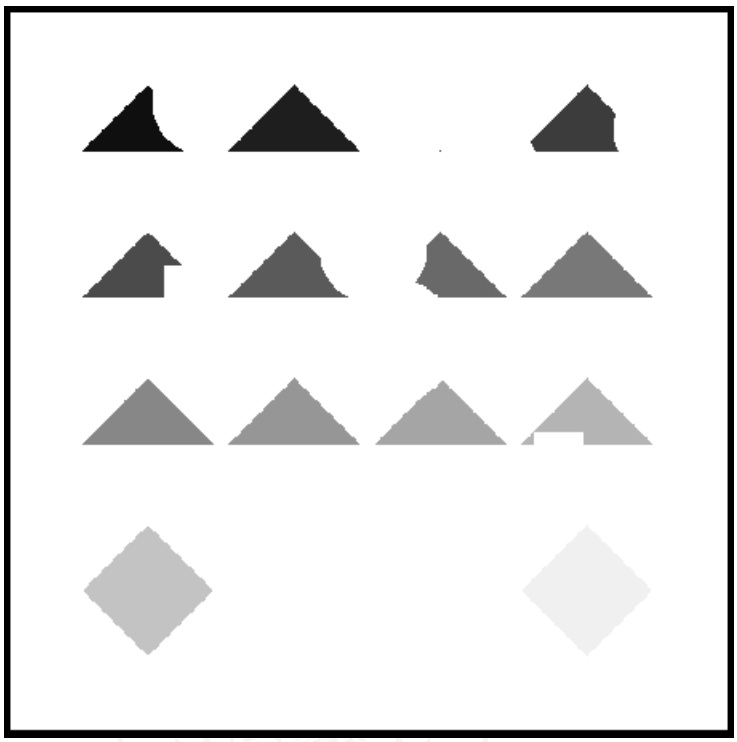

Figura 3.2: Teselaciones a partir de triángulos, iter = 0.13.

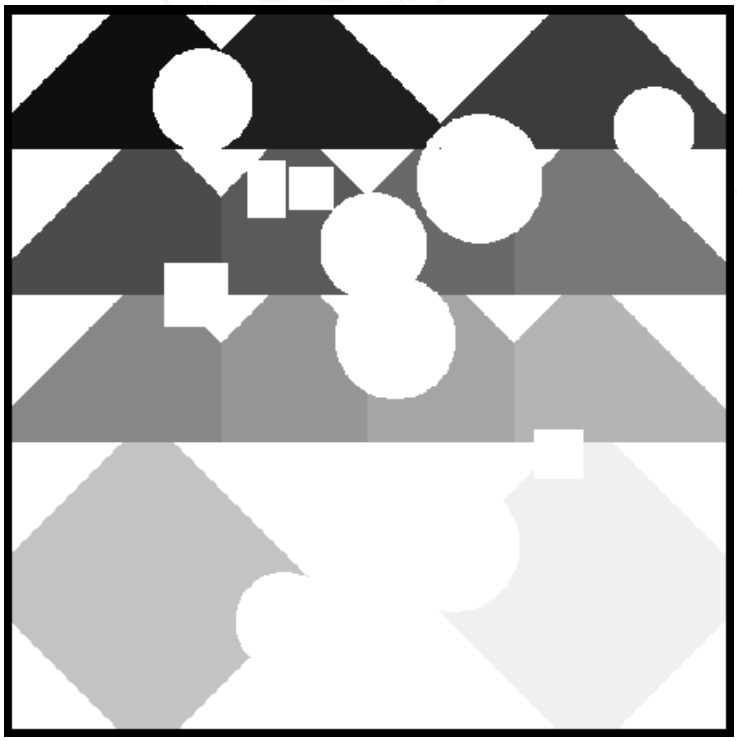

Figura 3.3: Teselaciones a partir de triángulos, iter = 0.67.

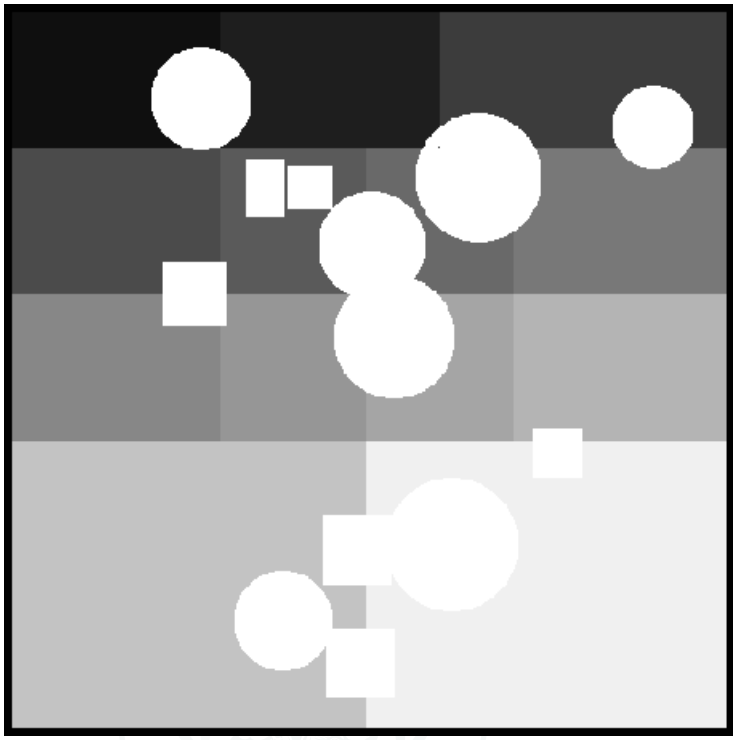

Figura 3.4: Teselaciones a partir de triángulos, iter = 1.

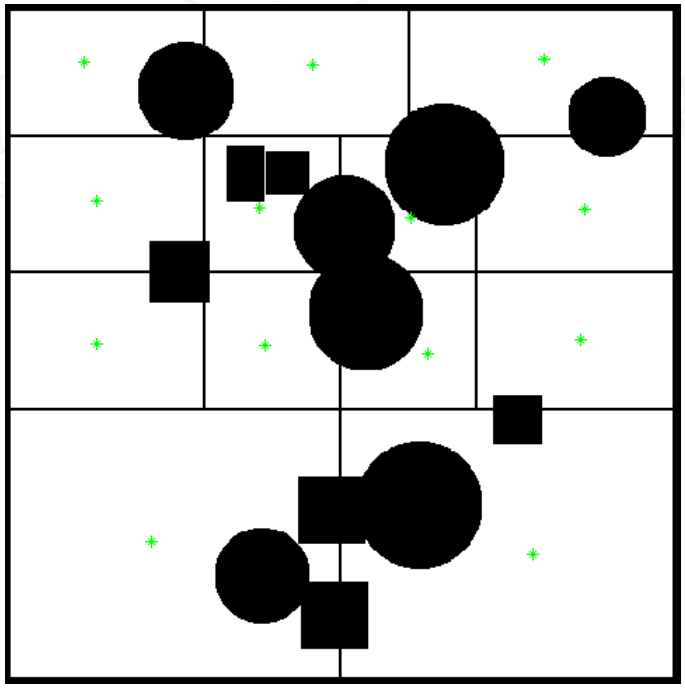

Figura 3.5: Centroides para las teselaciones con triángulos.

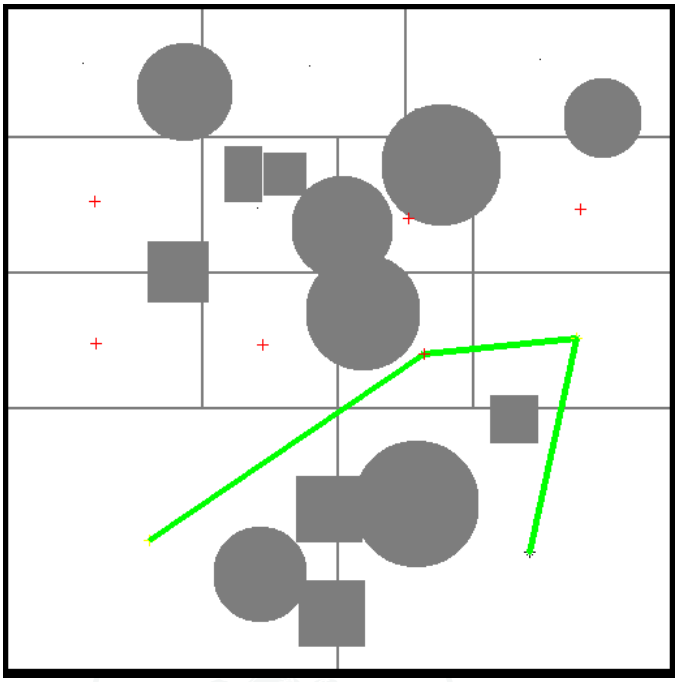

Figura 3.6: Búsqueda de la ruta a través de un entorno con 13 teselaciones.

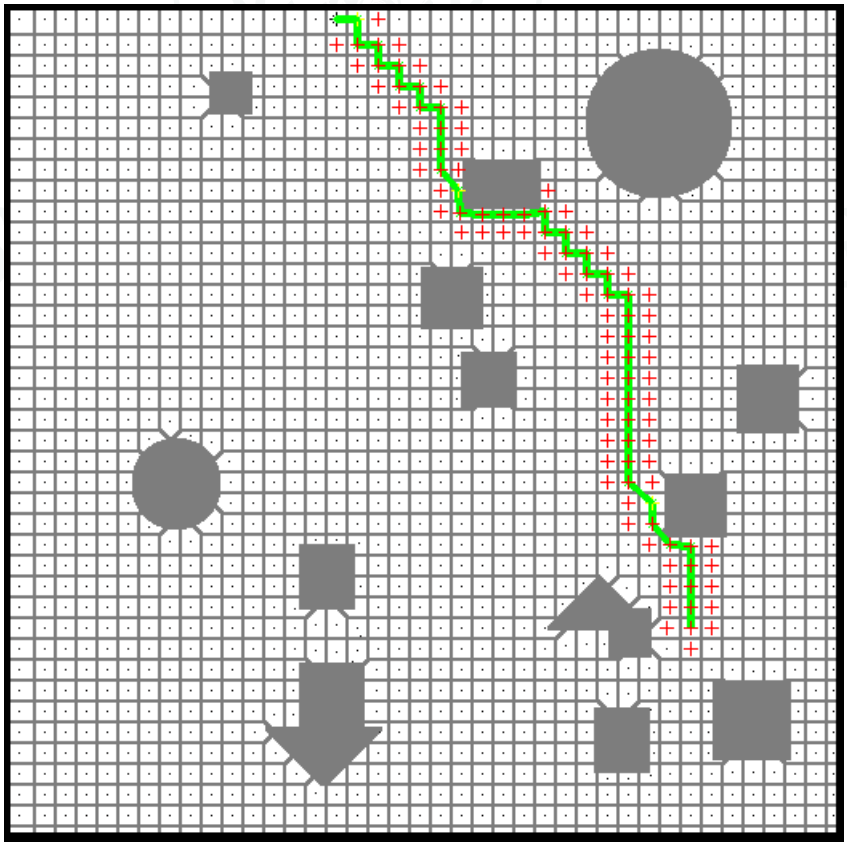

Figura 3.7: Búsqueda de la ruta a través de un etorno con 1400 teselaciones.

4. REQUERIMIENTOS Y EJECUCIÓN DEL ALGORITMO

Para la ejecución del algoritmo se requieren cinco algoritmos, todos corresponden a archivos ".m" de Matlab, el orden de ejecución corresponde al mismo orden de implementación mencionado, el primero es el encargado de generar el entorno ("Entorno.m"), el segundo calcula la cantidad de teselaciones dependiendo de la características del entorno ("Cantidad_Teselaciones.m"), en el tercero se realiza el teselado del entorno, este se puede realizar a partir de rombos, círculos o triángulos, el nombre de cada uno es ("Teselaciones_Rombos.m"), ("Teselaciones_Círculos.m") y ("Teselaciones_Triangulos.m") respectivamente, si se ejecutan los tres la imagen obtenida se sobre escribirá, por lo cual solo se tendría en cuenta el último en ejecutarse, el cuarto halla los centroides de cada teselación ("Limites_Teselaciones.m") y el quinto es el encargado de realizar la búsqueda ("Busqueda.m").

Para la correcta ejecución de los algoritmo, todos los archivos "$.m$" deben estar contenidos en la misma carpeta, además es recomendable usar una versión de Matlab igual o superior a la 2013a. Si el usuario desea cargar su propia imagen del entorno debe incluir la imagen dentro de la misma carpeta donde se encuentran los algoritmos, adicionalmente debe colocar los obsáculos de un color blanco con un valor de 255 en la escala de grises y el área de trabajo del robot con un color negro, 0 en escala de grises, además el formato de la imagen debe ser "$.PNG$" y debe llamarse "I".

5. RESULTADOS

Para la comparación de resultados para los diferentes frentes de ondas implementados en la teselación del entorno se mantuvieron dos criterios sobre los cuales se determinará un grado de selectividad sobre las teselaciones realizadas con diferentes frentes de onda, la forma y el tiempo de ejecución de cada algoritmo.

A continuación se mostrarán resultados obtenidos para la expansión con lo polígonos anteriormente mencionados, sin embargo los datos expuestos son realizados para una imagen con 49 teselaciones propuestas.

5.1. TIEMPOS DE EJECUCIÓN Y EFICIENCIA DE LOS ALGORITMOS

Al comparar los resultados las Tablas 5.2, 5.5 y 5.8 obtenidos por medio de la simulación de Matlab para las teselaciones con diferentes frente de onda se observó que la realizada por rombos tiene un menor tiempo de ejecución, sin embargo requiere mas ciclos para completar la teselación del entorno. De las Tablas 5.1, 5.4 y 5.7 la cobertura para las tres diferentes teselaciones fue muy parecida, sin embargo la teselación a partir de rombos me generó un valor de 100%. El algoritmo para la generación de teselaciones a partir de círculos genera un tiempo de respuesta mucho mas grande debido a la función diseñada para la expansión, puesto que al trabajar pixel a pixel es mucho mas sencillo algorítmicamente desarrollar cualquiera de los otros polígonos regulares propuestos. De las Tabla 5.3, 5.6 y 5.9 se puede afirmar que las instrucciones que requieren una mayor cantidad de tiempo para su ejecución son la lectura y escritura de la imagen del entorno, sin embargo al realizar teselaciones a partir de círculos la cantidad de comparaciones y condicionales para definir su contorno provocaron que sea la instrucción con mayor porcentaje de tiempo dentro del algoritmo.

Cabe aclarar que el tiempo de ejecución de estos algoritmos no depende del número de teselaciones tomadas, debido a que para una imagen determinada, el número de puntos por expandir

es constante por lo cual necesitará de un número constante de ciclos para teselar el entorno completamente.

5.1.1. Tablas de tiempos de ejecución de algoritmos e instrucciones para teselaciones a partir de triángulos

Código	Llamadas	Tiempo total	Porcentade de tiempo
imwrite	1	0.533 s	17.6
Figura 1.1	1	0.312 s	10.3
Plot	179	0.23 s	7.6
if (condicional frontera)	1251900	0.156 s	5.1
if (condicional entorno)	995956	0.14 s	4.6
Otros comandos		1.659 s	54.8
Total		3.03 s	100

Tabla 5.1: Líneas del algoritmo con mayor tiempo de ejecución.

Código	Llamadas	Tiempo total	Porcentaje de tiempo
imwrite	1	0.527 s	17.4
imshow	2	0.218 s	7.2
newplot	255	0.1 s	3.3
subplot	2	0.085 s	2.8
ishold	255	0.043 s	1.4
imread	1	0.043 s	1.4
hold	1	0.007 s	0.2
Autotiempo		2.007 s	66.2
Total		3.03 s	100

Tabla 5.2: Instrucciones del algoritmo con mayor tiempo de ejecución.

Total de líneas en la función	102
Lineas sin uso (en blanco - comentarios)	15
Lineas de código (que pueden ejecutarse)	87
Lineas de código (que se ejecutan)	87
Líneas de código que no se ejecutan	0
Porcentaje (funcionamiento)	100

Tabla 5.3: Tabla de cobertura.

5.1.2. *Tablas de tiempos de ejecución de algoritmos e instrucciones para teselaciones a partir de rombos*

Código	Llamadas	Tiempo total	Porcentade de tiempo
Figura 1.1	1	0.150 s	10.3
Figura 1.2	1	0.055 s	3.8
if (condicion de posición)	347760	0.048 s	3.3
if (condicional frontera - filas)	347760	0.047 s	3.2
if (condicional frontera - columnas)	347760	0.047 s	3.2
Otros comandos		1.116 s	76.3
Total		1.463 s	100

Tabla 5.4: Líneas del algoritmo con mayor tiempo de ejecución.

Código	Llamadas	Tiempo total	Porcentade de tiempo
imshow	2	0.112 s	7.7
imwrite	1	0.031 s	2.1
subplot	2	0.03 s	2.1
imread	1	0.014 s	1.0
newplot	16	0.01 s	0.7
ishold	16	0.005 s	0.3
hold	1	0.003 s	0.2
Autotiempo		1.258 s	86
Total		1.463 s	100

Tabla 5.5: Instrucciones del algoritmo con mayor tiempo de ejecución.

Total de líneas en la función	127
Lineas sin uso (en blanco - comentarios)	18
Lineas de código (que pueden ejecutarse)	109
Lineas de código (que se ejecutan)	104
Líneas de código que no se ejecutan	5
Porcentaje (funcionamiento)	95.41 %

Tabla 5.6: Tabla de cobertura.

5.1.3. Tablas de tiempos de ejecución de algoritmos e instrucciones para teselaciones a partir de círculos

Código	Llamadas	Tiempo total	Porcentaje de tiempo
if (condicional de frontera)	54091600	6.225 s	12.2
if (condicional de obstaculo)	54091600	5.680 s	11.1
Igualdad	54091600	5.508 s	10.7
end (if)	54091600	5.501 s	10.7
if (condicional contorno de circulos)	54091600	5.480 s	10.7
Otros comandos	54091600	22.978 s	44.7
Total		51.402 s	100

Tabla 5.7: Líneas del algoritmo con mayor tiempo de ejecución.

Código	Llamadas	Tiempo total	Porcentade de tiempo
imshow	2	0.127 s	0.2
subplot	2	0.03 s	0.1
imwrite	1	0.027 s	0.1
imread	1	0.013 s	0.0
newplot	16	0.009 s	0.0
ishold	16	0.003 s	0.0
hold	1	0.003 s	0.0
Autotiempo		51.190 s	99.6
Total		51.402 s	100

Tabla 5.8: Instrucciones del algoritmo con mayor tiempo de ejecución.

Total de líneas en la función	96
Lineas sin uso (en blanco - comentarios)	8
Lineas de código (que pueden ejecutarse)	88
Lineas de código (que se ejecutan)	83
Líneas de código que no se ejecutan	5
Porcentaje (funcionamiento)	94.32 %

Tabla 5.9: Tabla de cobertura.

5.2. FORMA DE LAS TESELACIONES

De las Figuras 5.1, 5.2 y 5.3, las cuales corresponden al mismo plano teselado por diferentes polígonos (triángulos, rombos y círculos respectivamente), se observó que las teselaciones por triángulos forman ángulos recto en sus intersecciones y no genera alguna inclinación o curva al momento de limitar con un obstáculo. Las teselaciones realizadas a partir de rombos y círculos equivalentes, ambas presentan curvas o límites inclinados cuando tienen fronteras con obstáculos y forman ángulos rectos entre sus intersecciones, la forma final obtenida para ambos casos es la misma, pero el tiempo de ejecución de las teselaciones a partir de círculos es alrededor de 30 a 35 veces mas grande que las realizadas por rombos. La forma de la teselación cuando limita con un obstáculo nos ayuda a determinar un centro de masa mas efectivo, pues al tener curvas entre la frontera de una teselación y un obstáculo permite que el centro de masa esté mas alejado de estos, lo que nos permite utilizar menos teselaciones para trazar la trayectoria de la ruta de menor coste sin atravesar obstáculos.

Teniendo en cuenta los dos aspectos anteriores los polígonos mas eficientes para realizar las teselaciones son triángulos y rombos, sin embargo el tiempo de ejecución por rombos es la mitad que el requerido por la expansión en frente de onda por triángulos, y ademas, la forma de las teselaciones cuando limitan con un obstáculo hacen que la expansión en frente de onda a partir de rombos sea la mas efectiva entre las tres.

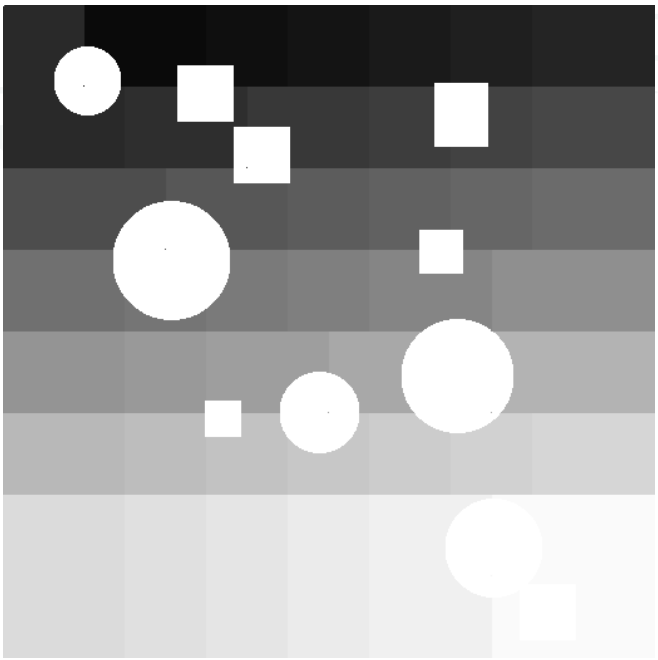

Figura 5.1: Teselaciones a partir de triángulos

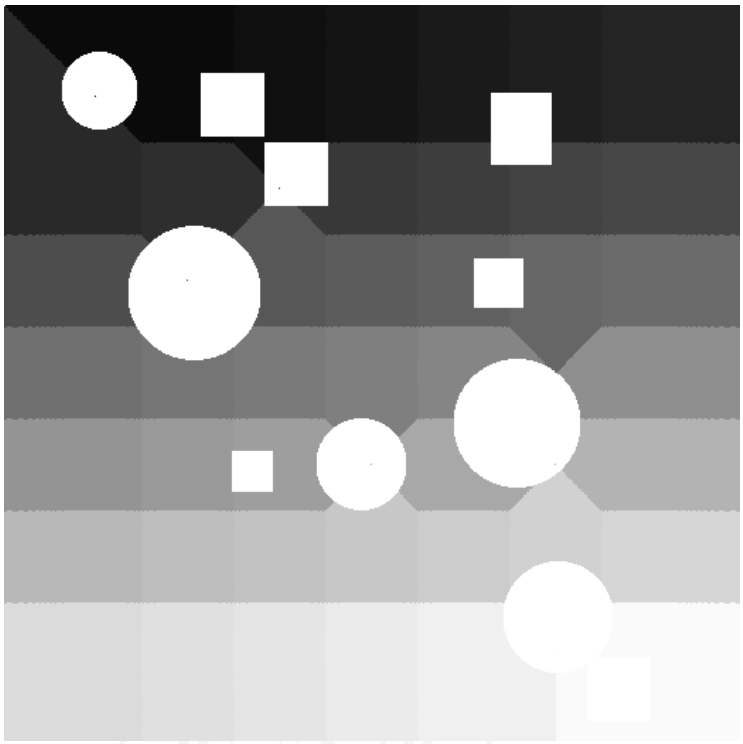

Figura 5.2: Teselaciones a partir de círculos

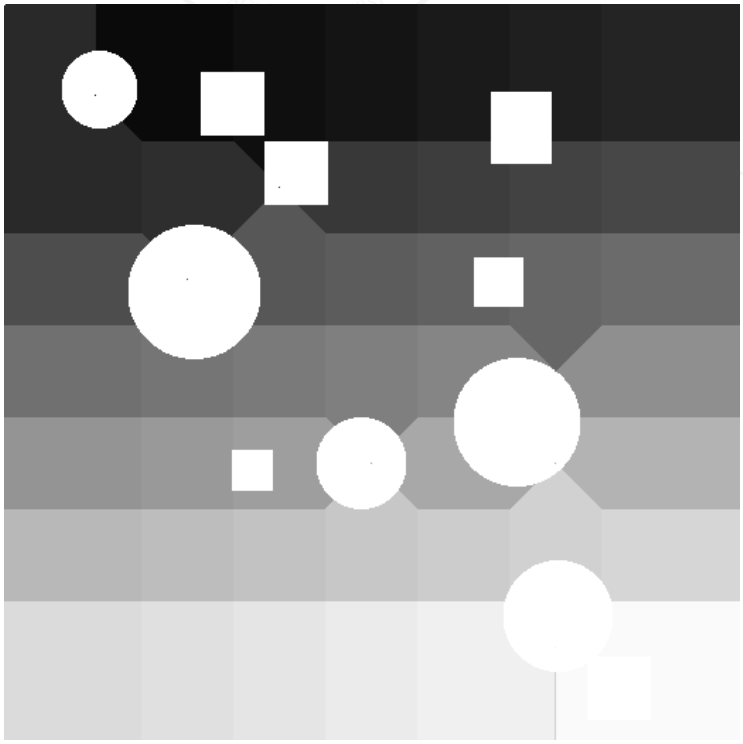

Figura 5.3: Teselaciones a partir de rombos

5.3. CENTROIDE

De la tabla 5.4 podemos obtener el tiempo de ejecución del algoritmo implementado para el cálculo de los centroides de cada teselación. El tiempo total de ejecución es de más de 25 segundos, al compararlo con el tiempo necesario para teselar el espacio a partir de rombos o triángulo es muy alto, sin embargo las instrucciones que requieren una mayor cantidad de tiempo son los condicionales y productos punto, la cantidad de condicionales para este algoritmo es muy alta debido a que para calcular el centroide de una teselación tiene que separarla del resto de la imagen y luego realizar producto punto, este proceso debe ser realizado tantas veces como teselaciones se hayan expandido, por lo cual el tiempo de ejecución para este algoritmo tiene una relación directamente proporcional con la cantidad de teselaciones del espacio de configuración.

código	Usos	Tiempo total	% Tiempo
if	49483200	5.773 s	22.7 %
end (if)	49483200	5.059 s	19.8 %
Producto punto	93080	3.496 s	13.7%
Producto punto F	93080	3.434 s	13.5%
Producto punto C	93080	2.746 s	10.8%
Otras instrucciones	N/A	4.977 s	19.5 %
Total	Total	25.485 s	100 %

Tabla 5.4: Tiempos de ejecución por instrucciones para el cálculo del centroide

6. CONCLUSIONES

En el presente documento se establecieron métodos convencionales para la descomposición de un espacio $\mathbb{R}^2$ en teselaciones. Luego se diseñó un método alterno en el cual se ubicaron los puntos de origen de las teslaciones uniformemente sobre el entorno y se expandieron con frentes de onda de diferetes formas. Posteriormente se evaluó cada uno de los entornos teselados para diferentes frentes de onda con el fin de determinar cual de estos era mas adecuada para la implementación de un algoritmo de búsqueda.

De acuerdo a lo observado en las simulaciones para diferentes entornos y diferentes tipos de teselaciones implementadas se logró determinar las mejores formas de expansión de frente de onda en las teselaciones utilizadas en base a dos criterios, el tiempo que le toma al algoritmo expandir el frente de onda sobre todo el entorno y la forma obtenida del teselado.

- Con respecto a los resultados obtenidos teniendo en cuenta las formas de frente de onda mas adecuadas para la posición y forma de expansión de las teselaciones implementadas son el rombo y el círculo, debido a que los centroides de las teselaciones se encuentran mas alejados de los obstáculos provocando que aumente el número de posibles movimientos entre dos puntos para el mismo entorno con la misma cantidad de teselaciones, sin embargo dado el alto tiempo de ejecución que tiene la expansión a partir de círculos, el rombo es una mejor opción.

- Ahora teniendo en cuenta los tiempos que le toma expandirse sobre el entorno para cada tipo de frente de onda implementado el triángulo es el más versátil, el tiempo de ejecución de este algoritmo para el mismo entorno con igual número de puntos a expandir es mucho mas corto que los demás, siendo hasta un 80% más rápido que la expansión a partir de rombos, la cual es la segunda con menor tiempo.

De lo anterior se puede concluir que las mejores formas de expansión son el rombo y el triángulo, sin embargo cada uno de ellos se desarrolla de una mejo forma para diferentes tipos de entornos o aplicaciones, para aplicaciones en tiempo real donde el criterio fundamental es

el bajo tiempo de ejecución total del algoritmo, la mejor opción es la expansión de teselaciones a partir de triángulos, por otro lado si no se requiere que los tiempos de ejecución sean estrictamente cortos pero se requiere garantizar una mayor cantidad de rutas para un número determinado de puntos, es mejor utilizar la expansión a partir de rombos. Por otro lado, a pesar de obtener diferentes resultados para las tres formas de expansión obtenidas, ambas lograron limitar el número de estados y posibles trayectorias del robot cumpliendo con el objetivo principal del documento el cual fue desarrollar un nuevo modelo que permita el reconocimiento del entorno que permitan desarrollar algoritmos de búsqueda los cuales garanticen encontrar la ruta más eficiente en un tiempo de ejecución mas corto que algoritmos tradicionales.

7. ANEXOS

A continuación se mostrarán los algoritmos implementados.

7.1. CREACIÓN DEL ENTORNO

% **Inicialización de variables**

f = 520; % Definir el tamaño del entorno de 520x520 pixeles.
c = 520; % Definir el tamaño del entorno de 520x520 pixeles.
I = $zeros(f, c)$;
% número de obstaculos max = 20, min = 10
% cantidad de formas = 3
% tamaño del obstaculo max = 10% de la altura del entorno
% forma = 1 =¿ cuadrado
% forna = 2 =¿ círculo
% forma = 3 =¿ rectángulo
o = $round(rand * 10) + 10$; % Cantidad de obstáculos
posx = $zeros(1, o)$; % Posición de los obstáculos en x
posy = $zeros(1, o)$; % Posición de los obstáculos en y
forma = 0; % Forma
tamano = $(f/20)$; % Tamaño adicional al mínimo (Filas)
mintam = $(f/20)$; % Minimo tamaño (Filas)
tamanoc = $(c/20)$; % Tamaño adicional al mínimo (Columnas)
mintamc = $(c/20)$; % Máximo tamaño (Columnas)

% **Generador de obstáculos**

for $i = 1 : o$ $forma = round(rand * 2) + 1$;
if $forma == 1$ % Cuadrado
$lado = round(rand * (tamano)) + round(mintam)$; % Define la longitud del lado del cuadrado
$posx(1, i) = round(rand * (f - (lado))) + round(lado/2)$; % Posición del cuadrado en X
$posy(1, i) = round(rand * (c - (lado))) + round(lado/2)$; % Posición del cuadrado en Y
for n = 1 : f
for m = 1 : c
if $n > ((posx(1, i) - (lado/2)))\&\&n < ((posx(1, i) + (lado/2)))\&\&m > ((posy(1, i) - (lado/2)))\&\&m < ((posy(1, i) + (lado/2)))$
$I(n, m) = 1$; % Gráfica el cuadrado de blanco
end
end
end
end
if $forma == 2$ % Circulo
$radio = round(rand * (tamano)) + round(mintam)$; % Define la longitud del radio de la circunferencia
$posx(1, i) = (round(rand * (f - (2 * radio))) + radio)$; % Posición en X del circulo
$posy(1, i) = (round(rand * (c - (2 * radio))) + radio)$; % Posición en Y del circulo
for $rcc = -radio : radio$
$yy = sqrt((radio)^2 - (rcc)^2)$; %Define el contorno de la circunferencia sobre el eje Y
$yy = round(yy)$;
if $yy = 0$
$I((posx(1, i) - yy) : (posx(1, i) + yy), (posy(1, i) + rcc)) = 1$; % Grafica contorno del circulo sobre el eje Y
end
end
end
if $forma == 3$ % Rectángulo
$base = round(rand * (tamano)) + round(mintam)$;
$altura = round(rand * (tamanoc)) + round(mintamc)$;
$posx(1, i) = round(rand * (f - (base))) + round(base/2)$;
$posy(1, i) = (round(rand * (c - altura))) + round(altura/2)$;
for $k = 1 : f$
for $l = 1 : c$
if $k > ((posx(1, i) - (base/2)))\&\&k < ((posx(1, i) + (base/2)))\&\&l > ((posy(1, i) - (altura/2)))\&\&l < ((posy(1, i) + (altura/2)))$
$I(k, l) = 1$;

end
end
end
end
end

% **Graficar la imagen de salida** %

$figure$
$imshow(I)$
$imwrite(I,'I.png')$

7.2. ALGORITMO PARA CALCULAR EL NÚMERO DE TESELACIONES

global tesel;

% **Inicializar variables** %

I = $imread('I.png')$;
$[f, c] = size(I)$;% Cargar la imagen del entorno.
contador = 0; % Variable para guardar el área de todos los obstáculos.
L = $bwlabel(I)$; % Función para etiquetar los obstáculos.
cantidad = 0; % Variable para guardar la cantidad de obstáculos.

% Ciclo para encontrar la cantidad de obstáculos y el área que ocupan en total.
for $i = 1 : f$ % Recorrido por filas.
for $j = 1 : c$ % Recorrido por columnas.
if $I(i, j) == 255$ % Condicional para reconocer los puntos blancos en el entorno (obstáculos).
$contador = contador + 1$; % Variable para guardar el área de los obstáculos.
end
if $cantidad < L(i, j)$ % Condicional para reconocer el número de obstáculos.
$cantidad = L(i, j)$; % Almacenar la etiqueta de los obsstáculos con mayor valor.
end
end
end

$x = zeros(cantidad, 1)$; % Vector para guardar el área de cada obstáculo.

% Ciclo para calcular el área de cada teselación.

for $k = 1 : cantidad$ % Recorrer todos los obsstáculos.
for $i = 1 : f$ % Recorrido por filas
for $j = 1 : c$ % Recorrido por columnas
if $L(i, j) == k$ % Condicional para evaluar si el punto actual corresponde con la alguna etiqueta de un obstáculo. $x(k, 1) = x(k, 1) + 1$; % Vector de áreas de las teselaciones.

end
end
end
end

$ccov = 2 * sqrt(cov(x)/1000000)$; % Normalizar la covarianza a un valor entre 0 y 20.
$media = sqrt((contador/cantidad)/100)$; % Normalizar la media a un valor entre 0 y 20.
$porc = 50 * (contador/(f * c))$; % Normalizar el porcentaje a un valor entre y 20
.

$tesel = round((ccov + cmedia + cporc)/3) - 1$; % Promedio de todas las

7.3. ALGORITMO DE EXPANSIÓN DE TESELACIONES A PARTIR DE TRIÁNGULOS

% **Inicialización de variables y lectura del entorno** %

global tesel;
I = imread('I.png'); % Entorno
$[f, c] = size(I)$; % Lectura del tamaño del entorno
x = tesel; % Número de puntos por eje (teselaciones)
x = $x + 1$; % Se le suma 1 a X para que todas las teselaciones sean equidistantes al dividir el entorno
tf = round(f/x); % División de filas
tc = round(c/x); % División de columnas
figure(1), subplot(1,2,1), imshow(I)
tes = ones(f,c); % Matriz de unos que corresponderá al entorno de salida
tes = $255. * tes$; % Se vuelve blanca la matriz en formato uint8
centx = zeros$(x - 1, x - 1)$;
centy = zeros$(x - 1, x - 1)$;
global tesv; % Matriz de teselaciones válidas como variable global
tesv = zeros$(x - 1, x - 1)$; % Tamaño de la matriz correspondiente al número de teselaciones
count = 0; % Contador de pixeles del espacio de configuración
coltes = zeros$(x - 1, x - 1)$; % Matriz para definir la tonalidad de cada teselación
cont = 1; % Contador para definir el tamaño de cada teselación
m = 1; % Codicional para conocer si el entorno se encuentra teselado en su totalidad
hold on

% **Ciclo para reconocer el espacio de configuración** %

for $i = 1 : f$ % Recorrido por filas del entorno
for $j = 1 : c$ % Recorrido por columnas del entorno
if $I(i, j) == 0$ % Compara un picel si se encuentra sobre el espacio de configuración
$count = count + 1$; % count aumenta en uno por cada pixel
end
end
end

% **Ciclo para definir la tonalidad de cada teselación y la separación entre cada punto de origen** %

for $i = 1 : x - 1$ % Ciclo para cantidad de teselaciones por filas
for $j = 1 : x - 1$ % Ciclo para cantidad de teselaciones por columnas
$px = i * tf$; % Define la posición en X de cada teselación
$py = j * tc$; % Define la posición en Y de cada teselación
if $(x - 1) >= 15$ % Define un rango máximo de teselaciones para diferenciar la tonalidad
kk = 15; % Limita el número de tonos (entre 1 y 254, 255 y 0 no cuentan)
else
$kk = (x - 1)$; % En caso contrario la tonalidad depende del número de teselaciones
end
$coltes(i, j) = (255/(((kk)^2) + 1)) * (((i - 1) * (kk)) + j)$; % Define la tonalidad dividiendo 255 en la posición de cada teselación
while $coltes(i, j) > 255$ % Condicional para tonalidades superiores a 255
$coltes(i, j) = coltes(i, j) - 255$; % Se resta 255 hasta que sea menor a 255
end
if $coltes(i, j) == 255$ % Si alguna teselación es igual a 255
$coltes(i, j) = 1$; % Se le impartirá una tonalidad de uno a esa teselación
end
if $I(px, py) == 0$ % Si la teselación se encuentra sobre el espacio de configuración
$plot(py, px, '*r')$ % Graficará un asterisco rojo sobre el centro de una teselación válida
$tesv(i, j) = 1$; % El punto sobre la matriz de teselaciones será uno, indicando que es válida
else if $I(px, py) == 255$ % Si la teselación se encuentra sobre un obstáculo
$plot(py, px, '*b')$ % Graficará un asterisco azul en ese punto
$tesv(i, j) = 0$; % El punto sobre la matriz de teselaciones será cero, indicando que no es válida
end
end
end
end

% **Ciclo para la expansión en frente de onda de cada teselación** %

while $m < count$ % Ciclo para confirmar la teselación del espacio de configuración

for $k = 1 : x - 1$ % Ciclo por columnas de las teselaciones
for $l = 1 : x - 1$ % Ciclo por filas de las teselaciones
for $j = 0 : cont$ % Contador j para definir el tamaño de las teselaciones
if $tesv(l, k) == 1$ % Condicional si es una teselación válida
% Ciclo para triangulos que se encuentren sobre la última fila %
if $l == (x - 1)$
% Primer cuadrante
$yc = (k * tc) + j$; % Incremento en Y igual la posición en Y mas el conteo de j
$xc = (l * tf) + cont - j$; % Incremento en X igual a la posición en X mas el decremento de cont
if $yc >= c$ % Limitación en Y si sobrepasa el entorno
$yc = c$; % Valor máximo de Y igual a la última columna
end
if $xc >= f$ % Limitación en X si sobrepasa el entorno
$xc = f$; % Valor máximo de X igual a la última fila
end
% Condicional para poder expandir el frente de onda Si la imagen original el pixel se encuentra sobre el espacio de configuración y en la imagen de salida el pixel aum no ha sido expandido por alguna teselación
if $I(xc, yc) == 0 \&\& tes(xc, yc) == 255$
$tes(xc, yc) = coltes(l, k)$; % Define la tonalidad del pixel dependiendo de la teselación correspondiente
$m = m + 1$; % Aumenta el contador de pixeles expandidos
end
% Segundo cuadrante
$yc = (k * tc) + j$; % Incremento en Y igual la posición en Y mas el conteo de j
$xc = (l * tf) - (cont - j)$; % Incremento en X igual a la posición en X mas el incremento de cont
if $yc >= c$ % Limite en Y si sobrepasa la altura del entorno
$yc = c$; % Valor máximo de Y igual a la última columna
end
if $xc <= 1$ % Limite en X si sobrepasa el entorno
$xc = 1$; % Valir mínimo de X igual a la primera fila
end
% Condicional para poder expandir el frente de onda si la imagen original el pixel se encuentra sobre el espacio de configuración y en la imagen de salida el pixel aum no ha sido expandido por alguna teselación
if $I(xc, yc) == 0 \&\& tes(xc, yc) == 255$

$tes(xc, yc) = coltes(l, k)$; % Define la tonalidad del pixel dependiendo de la teselación correspondiente
$m = m + 1$; % Aumenta el contador de pixeles expandidos
end
% Tercer cuadrante
$yc = (k * tc) - j$; % Incremento en Y igual la posición en Y menos el conteo de j
$xc = (l * tf) + cont - j$; % Incremento en X igual a la posición en X mas el decremento de cont
if $yc <= 1$ % Limitación en Y si sobrepasa el entorno
$yc = 1$; % Valor mínimo de Y igual a la primera columna
end
if $xc >= f$ % Limitación en X si sobrepasa el entorno
$xc = f$; % Valor máximo de X igual a la última fila
end
% Condicional para poder expandir el frente de onda si la imagen original el pixel se encuentra sobre el espacio de configuración y en la imagen de salida el pixel aum no ha sido expandido por alguna teselación
if $I(xc, yc) == 0 \&\& tes(xc, yc) == 255$
$tes(xc, yc) = coltes(l, k)$; % Define la tonalidad del pixel dependiendo de la teselación correspondiente
$m = m + 1$; % Aumenta el contador de pixeles expandidos
end
% Cuarto cuadrante
$yc = (k * tc) - j$; % Incremento en Y igual la posición en Y menos el conteo de j
$xc = (l * tf) - (cont - j)$; % Incremento en X igual a la posición en X mas el incremento de cont
if $yc <= 1$ % Limitación en Y si sobrepasa el entorno
$yc = 1$; % Valor mínimo de Y igual a la primera columna
end
if $xc <= 1$ % Limitación en X si sobrepasa el entorno
xc = 1; % Valir mínimo de X igual a la primera fila
end
% Condicional para poder expandir el frente de onda si la imagen original el pixel se encuentra sobre el espacio de configuración y en la imagen de salida el pixel aum no ha sido expandido por alguna teselación
if $I(xc, yc) == 0 \&\& tes(xc, yc) == 255$
$tes(xc, yc) = coltes(l, k)$; % Define la tonalidad del pixel dependiendo de la teselación correspondiente

$m = m + 1$; % Aumenta el contador de pixeles expandidos
end
% Ciclo para triangulos que no se encuentren sobre la última fila %
else
% Cuarto cuadrante
$yc = (k * tc) - j$; % Incremento en Y igual la posición en Y menos el conteo de j
$xc = (l * tf) - (cont - j)$; % Incremento en X igual a la posición en X mas el incremento de cont
if $yc <= 1$ % Limitación en Y si sobrepasa el entorno
$yc = 1$; % Valor mínimo de Y igual a la primera columna
end
if $xc <= 1$ % Limitación en X si sobrepasa el entorno
$xc = 1$; % Valir mínimo de X igual a la primera fila
end
% Condicional para poder expandir el frente de onda si la imagen original el pixel se encuentra sobre el espacio de configuración y en la imagen de salida el pixel aum no ha sido expandido por alguna teselación
if $I(xc, yc) == 0 \&\& tes(xc, yc) == 255$
$tes(xc, yc) = coltes(l, k)$; % Define la tonalidad del pixel dependiendo de la teselación correspondiente
$m = m + 1$; % Aumenta el contador de pixeles expandidos
end
% Segundo cuadrante
$yc = (k * tc) + j$; % Incremento en Y igual la posición en Y mas el conteo de j
$xc = (l * tf) - (cont - j)$; % Incremento en X igual a la posición en X mas el incremento de cont
if $yc >= c$ % Limitación en Y si sobrepasa la altura del entorno
$yc = c$; % Valor máximo de Y igual a la última columna
end
if $xc <= 1$ % Limitación en X si sobrepasa el entorno
$xc = 1$; % Valir mínimo de X igual a la primera fila
end
% Condicional para poder expandir el frente de onda si la imagen original el pixel se encuentra sobre el espacio de configuración y en la imagen de salida el pixel aum no ha sido expandido por alguna teselación
if $I(xc, yc) == 0 \&\& tes(xc, yc) == 255$
$tes(xc, yc) = coltes(l, k)$; % Define la tonalidad del pixel dependiendo de la teselación correspondiente

$m = m + 1$; % Aumenta el contador de pixeles expandidos
end
end
end
end
end
end
$cont = cont + 1$; % Aumento del tamaño del frente de onda para las teselaciones **end**

% **Ciclo para graficar los puntos de origen de las teselaciones** %

for $r = 1 : x - 1$ % Recorrido por filas
for $t = 1 : x - 1$ % Recorrido por columnas
$yc = (t * tc)$; % Posición en Y de cada teselación
$xc = (r * tf)$; % Posición en X de cada teselación
$tes(xc, yc) = coltes(r, t)$; % Graficar el punto de la teselación con su valor correspondiente
end
end
for $i = 1 : x - 1$
for $j = 1 : x - 1$
$centx(i, j) = i * tf$;
$centy(i, j) = j * tc$;
end
end

% **Graficar la imagen de salida** %

$r = tesv$; % Igualar la matriz de teselaciones a la variable global
$tes = uint8(tes)$; % Igualar la imagen de salida al formato uint8
$figure(1), subplot(1, 2, 2), imshow(tes)$ % Graficar la imagen de salida
$imwrite(tes, 'J.png')$ % Guardar la imagen

7.4. CÁLCULO DEL CENTRO DE MASA

% Inicialización de variables y lectura del entorno %
I = imread('J.png'); % Lectura del entorno
$[f, c] = size(I)$; % Lectura del tamaño del entorno
tesx =$ones(f, c)$; % Se creo una matriz del tamaño del entorno para establecer los límites de las teselaciones
tesx = $255. * tesx$; % El color preestablecido del espacio de configuración es blanco
max = 0;% Valor en el que se almacenará el número de teselaciones
counter = 0; % Contador para reconocer el área de cada teselación
hold on

% Ciclo para establecer los límites de las teselaciones %
for $i = 1 : f - 1$ % Recorrido por filas
for $j = 1 : c - 1$ % Recorrido por culmnas
% Hallar diferencias entre la tonalidad de picseles vecinos
if $(I(i, j) \ = I(i + 1, j))||$
$(I(i, j) \ = I(i, j + 1))||(I(i, j) \ = I(i + 1, j + 1))||(I(i, j) == 255)\ tesx(i, j) = 0$; % Establecer de color negro los límites entre teselaciones
$tesx(i + 1, j) = 0$; % Establecer de color negro los límites entre teselaciones
$tesx(i, j + 1) = 0$; % Establecer de color negro los límites entre teselaciones
$tesx(i + 1, j + 1) = 0$; % Establecer de color negro los límites entre teselaciones
end
end
end

imshow(tesx) % Grafica el entorno con los límites entre teselaciones
L = bwlabel(tesx); % Enumeración de las teselaciones

% Ciclo para hayar la cantidad de teselaciones %
for $k = 1 : f$ % Recorrido por filas
for $l = 1 : cv$ % Recorrido por columnas
if $L(k, l) >= max$ % Comparación, si la etiqueta de la teselación actual es mas grande que el acumulado
$max = L(k, l)$; % Igualar el acumulado al mayor

end
end
end

$z = zeros(f, c)$; % Matriz para encontrar el centro de masa de cada teselación
$centx = zeros(1, max)$; % Vector para guardar las posiciones en el eje X de los centroides
$centy = zeros(1, max)$; % Vector para guardar las posiciones en el eje X de los centroides
$tesy = ones(f, c)$; % Matriz para guardar los valores de la imagen de salida
$tesy = 255. * tesy$; % Imagen de salida con fondo blanco

% Ciclo para establecer los límites de l
as teselaciones de color gris en la imagen de salida % **for** $i = 1 : f - 1$ % Recorrido por filas
for $j = 1 : c - 1$ % Recorrido por columnas
if $(I(i, j) = I(i + 1, j))||(I(i, j) = I(i, j + 1))||(I(i, j) = I(i + 1, j + 1))||(I(i, j) == 255)$
$tesx(i, j) = 0$;
$tesx(i + 1, j) = 0$;
$tesx(i, j + 1) = 0$;
$tesx(i + 1, j + 1) = 0$;
$tesy(i, j) = 125$;
$tesy(i + 1, j) = 125$;
$tesy(i, j + 1) = 125$;
$tesy(i + 1, j + 1) = 125$;
end
end
end

% Ciclo para hallar el centroide de cada teselación %
for $m = 1 : max$% Ciclo para recorrer todas las teselaciones
for $g = 1 : f$ % Recorrido por filas
for $h = 1 : c$ % Recorrido por columnas
% Ciclo para hallar el área de cada teselación
if $L(g, h) == m$
$z(g, h) = 255$;
$counter = counter + 1$; % Contador del área
end
end

end
if *counter* >= 50 % Si el área es menor a 30 pixeles, se contará como una teselación inválida
% Variables para hallar el centro de masa en el eje X
$tq_1 = 0$;
$M_1 = 0$;
% Variables para hallar el centro de masa en el eje Y
$tq_2 = 0$;
$M_2 = 0$;
% Matrices para realizar producto punto de la matriz de teselaciones con una imagen del mismo tamaño de color blanco
$o = 255. * ones(f, 1)$; % Matriz para realizar el producto punto en el eje Y
$p = 255. * ones(1, c)$; % Matriz para realizar el producto punto en el eje X
for i = 1 : f % Ciclo para hallar la coordenada del centroide en Y
$tq_1 = i * dot(p, z(i, :)) + tq_1$;
$M_1 = dot(p, z(i, :)) + M_1$;
end;
$Cmx_1 = tq_1/M_1$; % Centro de masa en Y
hold on
for j = 1 : c % Ciclo para hallar la coordenada del centroide en X
$tq_2 = j * dot(o, z(:, j)) + tq_2$;
$M_2 = dot(o, z(:, j)) + M_2$;
end;
$Cmx_2 = tq_2/M_2$; % Centro de masa en X
$plot(Cmx_2, Cmx_1, '*g')$ % Graficar el centro de masa como un asterisco verde
$z = zeros(f, c)$; % Reiniciar la matriz de teselaciones
$counter = 1$; % Reiniciar el contador
end
$tesy(round(Cmx_1), round(Cmx_2)) = 0$; % Graficar el centroide en la imagen de salida como un punto negro
end
$tesy = uint8(tesy)$; % Pasar la imagen de salida a un formato de escala de grises

$figure\ imshow(tesy)$ % Graficar la imagen de salida
$imwrite(tesx, 'K.png')$ % Guardar la imagen de límites de teselaciones
$imwrite(tesy, 'L.png')$ % Guardar la imagen con los centroides de cada teselación

7.5. BÚSQUEDA

%% **Anexo - Programa de búsqueda**
I = imread('K.png'); % Cargar imagen del entorno teselado
L = imread('L.png'); % Cargar imagen con los centroides
$[f, c]$ = size(I); % Reconocer el tamaño del entorno
conti = 0; % Contador para conocer la cantidad de teselaciones
grises = 0; % Variable para conocer si pasa a través de obstáculos
imshow(I) % Mostrar imagen
hold on

%% Ciclo para conocer la cantidad de teselaciones
for i = 1 : f % Recorrido por fila
for j = 1 : c % Recorrido por columnas
if L(i,j) == 255 % Condicional para reconocer los puntos blancos sobre la imagen
conti = conti + 1; % Aumentar el contador
end
end
end

tesy = zeros(1,conti); % Vector para guardar las posiciones en el eje Y de los centroides
tesx = zeros(1,conti); % Vector para guardar las posiciones en el eje X de los centroides
contt = 1; % Variable para guardar la cantidad de centroides
for i = 1 : f % Recorrido por filas
for j = 1 : c % Recorrido por columnas
if L(i,j) == 255 % Condicional para reconocer los puntos blancos sobre la imagen
tesx(1,contt) = i; % Guardar las posiciones en X
tesy(1,contt) = j; % Guardar las posiciones en Y
contt = contt + 1; % Aumentar el contador de centroides
end
end
end

%% Definir puntos aleatorios de principio y fin
actual_y = round(rand*(conti - 1))+1; % Posición inicial sobre el vector de centroides

fin_y = round(rand*(conti - 1))+1; % Posición final sobre el vector de centroides
actual_x = tesx(1,actual_y); % Posición inicial en X
actual_y = tesy(1,actual_y); % Posición inicial en Y
fin_x = tesx(1,fin_y); % Posición final en X
fin_y = tesy(1,fin_y); % Posición final en Y
distan = zeros(conti,conti); % Matriz para guardar las distancias entre puntos

%% Ciclo para calcular las distancias
for i = 1 : conti % Selección del punto inicial
for j = 1 : conti % Selección del punto final
distan(i,j) = sqrt((tesx(1,j) - tesx(1,i))$\hat{2}$ + (tesy(1,j) - tesy(1,i))$\hat{2}$); % Cálculo de distancia entre el punto inicial y el final
end
end

%% Ciclo para asegurar una distancia entre puntos por sí mismos for i = 1 : conti
distan(i,i) = 1002; % Asegurar una distancia mas grande que la máxima posible
end

%% Ciclo para encontrar la trayectoria
while actual_y = fin_y —— actual_x = fin_x % Condicional para moverse mientras no halla culminado la trayectoria
%% Ciclo para seleccionar el punto actual
for i = 1 : conti
if actual_x == tesx(1,i) && actual_y == tesy(1,i) % condicional para encontrar el punto actual
ry = i; % Guardar la posición en el vector de centroides del punto
actua
l end
end
mov_x = actual_x; % Guardar el punto actual en X
mov_y = actual_y; % Guardar el punto actual en Y
plot(actual_y,actual_x,'+y') % Graficar los puntos
aa = 0; % Variable para considerar los puntos cercanos
memoria_dist = 1002; % Variable para comparar distancias
while aa = 4 % Condicional para hallar los cuatro puntos mas cercanos

min = 1002; % Seleccionar un valor mas grande que la máxima distancia posible
for i = 1 : conti % Ciclo para encontrar la menor distancia
if distan(ry,i) ¡ min
min = distan(ry,i); % Guardar la menor distancia
dx = i; % Guardar la posición del vector en el punto con menor distancia
end
end
distan(ry,dx) = 1003; % Sobre escribir el valor de la distancia
destino_x = tesx(1,dx); % Seleccionar el siguiente punto de la trayectoria en X
destino_y = tesy(1,dx); % Seleccionar el siguiente punto de la trayectoria en Y
aa = aa + 1; % Aumentar el contador
distanciast = sqrt((destino_x - fin_x)$\hat{}$2 + abs(destino_y - fin_y)$\hat{}$2); % Calcular la distancia entre el punto siguiente y el final
if distanciast ¡ memoria_dist % Condicional para encontrar la menor disntancia
memoria_dist = distanciast; % Guardar la menor distancia
memoria_x = destino_x; % Guardar la posición del siguiente punto en X
memoria_y = destino_y; % Guardar la posición del siguiente punto en Y
end
end

destino_x = memoria_x; % Guardar la posición en X
destino_y = memoria_y; % Guardar la posición en Y
plot(destino_y,destino_x,'*g') % Graficar el siguiente punto
plot(fin_y,fin_x,'*k') % Graficar el punto de meta
mem_x = actual_x; % Almacenar el punto actual en X
mem_y = actual_y; % Almacenar el punto actual en Y
xx = abs(destino_x - actual_x); % Calcular la diferencia de posición sobre el eje X entre los puntos de origen y meta
yy = abs(destino_y - actual_y); % Calcular la diferencia de posición sobre el eje Y entre los puntos de origen y meta

if xx ¿= yy % Comparar la pendiente mayor con el eje X como referencia
pend = (destino_y - actual_y) / (destino_x - actual_x); % Calcular la pendiente con referencia en el eje X
cond = 1; % Condicional para establecer la pendiente sobre el eje X else % Comparar la pendiente sobre el eje Y como referencia

```
pend = (destino_x - actual_x) / (destino_y - actual_y); % Calcular la pendiente con referencia
en el eje Y
cond = 2; % Condicional para establecer la pendiente sobre el eje Y
end
hold on
l=0; % Condicional para conocer la posición entre desplazamientos
cont = 1; % Contador para almacenar la longitud actual del desplazamiento
while l == 0 % Mientras no se halla desplazado hasta el siguiente punto
if cond == 1 % Condicional para desplazamientos con pendiente sobre el eje X
if destino_x ¿ actual_x && destino_y ¿ actual_y
mov_x = mov_x + 1;
mov_y = actual_y + round(cont*pend);
elseif destino_x ¿ actual_x && destino_y ¡ actual_y
mov_x = mov_x + 1;
mov_y = actual_y + round(cont*pend);
elseif destino_x ¡ actual_x && destino_y ¡ actual_y
mov_x = mov_x - 1;
mov_y = actual_y - round(cont*pend);
elseif destino_x ¡ actual_x && destino_y ¿ actual_y
mov_x = mov_x - 1;
mov_y = actual_y - round(cont*pend);
elseif destino_y == actual_y && destino_x ¿ actual_x
mov_x = mov_x + 1;
mov_y = actual_y;
elseif destino_y == actual_y && destino_x ¡ actual_x
mov_x = mov_x - 1;
mov_y = actual_y;
end
elseif cond == 2 % Condicional para desplazamientos con pendiente sobre el eje Y
if destino_x ¿ actual_x && destino_y ¿ actual_y
mov_y = mov_y + 1;
mov_x = actual_x + round(cont*pend);
elseif destino_x ¿ actual_x && destino_y ¡ actual_y
mov_y = mov_y - 1;
mov_x = actual_x - round(cont*pend);
elseif destino_x ¡ actual_x && destino_y ¡ actual_y
mov_y = mov_y - 1;
mov_x = actual_x - round(cont*pend);
```

```
elseif destino_x ¡ actual_x && destino_y ¿ actual_y
mov_y = mov_y + 1;
mov_x = actual_x + round(cont*pend);
elseif destino_x == actual_x && destino_y ¿ actual_y
mov_y = mov_y + 1;
mov_x = actual_x;
elseif destino_x == actual_x && destino_y ¡ actual_y
mov_y = mov_y - 1;
mov_x = actual_x;
end
end
if I(mov_x,mov_y) == 0 % Condicional para conocer la cantidad de pixeles correspondientes a obstáculos que atraviesa
grises = grises + 1; % Aumentar el contador
end
if mov_x == destino_x && mov_y == destino_y % Condicional para conocer si se llego a la siguiente teselación
l=1; % Cambiar condicional de desplazamiento
actual_x = destino_x; % Cambiar la posición actual en X
actual_y = destino_y; % Cambiar la posición actual en Y
end
plot(mov_y,mov_x,'.g') % Graficar desplazamiento
cont = cont + 1; % Aumentar la longitud del desplazamiento
end
end
```

BIBLIOGRAFÍA

1 Subhrajit Bhattacharya, TOPOLOGICAL AND GEOMETRIC TECHNIQUES IN GRAPH SEARCH-BASED ROBOT PLANNING, University of Pennsylvania, 2012.

2 Frederic Bourgault, Alexei A. Makarenko, Stefan B. Williams, Ben Grocholsky, and Hugh F.Durrant-Whyte. Information based adaptive robotic exploration. In in Proceedings IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS, pags 540-545, 2002.

3 Douglas Demyen and Michael Buro. Eficient triangulation-based pathnding. In AAAI06: Proceedings of the 21st national conference on Artificial intelligence, pags 942 -947. AAAI Press, 2006.

4 D. Grigoriev and A. Slissenko. Polytime algorithm for the shortest path in a homotopy class amidst semi-algebraic obstacles in the plane. In ISSAC '98: Proceedings of the 1998 international symposium on Symbolic and algebraic computation, pags 17-24, New York, NY, USA, 1998. ACM.

5 A.G. Kovalev. Vector bundles. 2007. From http://www.dpmms.cam.ac.uk/agk22/.

6 E. Schmitzberger, J.L. Bouchet, M. Dufaut, D. Wolf, and R. Husson. Capture of homotopyclasses with probabilistic road map. In International Conference on Intelligent Robots andSystems, volume 3, pags 2317-2322, 2002.

7 M. Schwager, J. McLurkin, and D. Rus. Distributed coverage control with sensory feedback for networked robots. In Proc. of Robot.: Sci. and Syst., Philadelphia, PA, August 2006.

8 Subhrajit Bhattacharya, TOPOLOGICAL AND GEOMETRIC TECHNIQUES IN GRAPH SEARCH-BASED ROBOT PLANNING, University of Pennsylvania, 2012, Capítulo 2, pags 9 - 37.

9 Subhrajit Bhattacharya, TOPOLOGICAL AND GEOMETRIC TECHNIQUES IN GRAPH SEARCH-BASED ROBOT PLANNING, University of Pennsylvania, 2012, Capítulo 5, pags 95 - 118.

10 M. Schwager, J. E. Slotine, and D. Rus. Decentralized, adaptive control for coverage with networked robots. In Proc. of the IEEE Intl. Conf. on Robot. and Autom., pags 3289:3294, Rome, Italy, April 2007.

11 C. Stachniss, G. Grisetti, and W. Burgard. Information gain-based exploration using raoblackwellized particle

lters. In Proc. of Robot.: Sci. and Syst., pags 65:72, Cambridge, MA, June 2005.

12 Nejat Karabakal and James C. Bean. A multiplier adjustment method for multiple shortest path problem. Technical report, The University of Michigan, June 1995.