UCUENCA

Universidad de Cuenca

Facultad de Ingeniería

Carrera de Ingeniería en Telecomunicaciones

Evaluación e implementación de un algoritmo para el diseño de rutas en ambientes dinámicos y variantes para vehículo autónomos

Trabajo de titulación previo a la obtención del título de Ingeniero en Telecomunicaciones

Autores:

Jean Carlo Aucapiña Lozada

Luis Carlos Sáenz Delgado

Director:

Luis Ismael Minchala Ávila

ORCID: 0000-0003-0822-0705

Cuenca, Ecuador

2024-09-03

Resumen

Esta investigación se centra en la implementación y evaluación de algoritmos para la planificación de rutas en entornos dinámicos y variantes, aplicados a vehículos autónomos. Se utiliza MATLAB como herramienta de simulación, a lo largo del trabajo se implementan dos algoritmos: D* Lite y Q-learning para comparar su rendimiento con otros algoritmos ya implementados en el entorno de simulación. D* Lite es conocido por su capacidad para replanificar rutas de manera eficiente ante cambios en el entorno, mientras que Q-learning permite mejorar continuamente las decisiones de navegación mediante el aprendizaje por refuerzo. Se llevan a cabo múltiples experimentos para evaluar el rendimiento de estos algoritmos. Antes de realizar la comparación se explica la lógica utilizada para el desarrollo de los algoritmos mencionados. Para la evaluación se consideraron métricas que incluyen el tiempo de ejecución, la longitud del camino y el costo del camino en función de la distancia. Además, se evaluó la capacidad de los algoritmos para evadir obstáculos en entornos controlados y aleatorizados. Para validar la eficacia de los algoritmos desarrollados, se realizó un benchmark comparando estos algoritmos con otros métodos de planificación de rutas existentes.

Los resultados demostraron que ambos algoritmos son efectivos en entornos dinámicos. D* Lite destacó por su simplicidad y rapidez, logrando replanificaciones eficientes y rápidas. Por otro lado, Q-learning mostró una gran adaptabilidad y capacidad de aprendizaje, mejorando sus decisiones de navegación a medida que interactuaba con el entorno.

*Palabras clave del autor: planificación de rutas, MATLAB, D*Lite, Q-learning*

Abstract

This research focuses on the implementation and evaluation of algorithms for route planning in dynamic and environmental changes, applied to autonomous vehicles. MATLAB is used as a simulation tool, and two algorithms are implemented throughout the work: D* Lite and Q-learning, to compare their performance with other algorithms already implemented in the simulation environment. D* Lite is known for its ability to efficiently replan routes when there are environmental changes, while Q-learning allows continuous improvement of navigation decisions through reinforcement learning. Multiple experiments are conducted to evaluate the performance of these algorithms. Before the comparison, the logic used to develop the mentioned algorithms is explained.

For the evaluation, metrics including execution time, path length, and path cost based on distance were considered. Additionally, the ability of the algorithms to avoid obstacles in controlled and randomized environments was evaluated. To validate the effectiveness of the developed algorithms, a benchmark was performed comparing these algorithms with other existing route planning methods.

The results showed that both algorithms are effective in dynamic environments. D* Lite stood out for its simplicity and speed, achieving efficient and quick replanning. On the other hand, Q-learning showed great adaptability and learning ability, improving its navigation decisions as it interacted with the environment.

*Author keywords: path planning, MATLAB, D*Lite, Q-learning*

Índice de contenido

Índice de figuras

Índice de tablas

Agradecimientos

Al culminar nuestro trabajo de titulación, que en ocasiones pareció un objetivo lejano, queremos expresar nuestra gratitud a todas las personas que de una u otra forma han contribuido de manera significativa en su realización.

En primer lugar, extendemos nuestro más profundo agradecimiento a nuestro tutor, Ing. Luis Minchala, por su orientación desde el inicio, guiándonos siempre hacia el camino correcto para alcanzar nuestros objetivos. Sin su experiencia y conocimientos, lograr nuestras metas habría sido complicado. Valoramos enormemente el tiempo dedicado a resolver nuestras inquietudes que surgieron durante este proyecto.

Asimismo, queremos agradecer a los docentes que han formado nuestra base de conocimientos a lo largo de nuestra formación académica. Su conocimiento, experiencias y tiempo compartidos durante la carrera sirvieron como base para el desarrollo de este trabajo. Apreciamos su profesionalismo y compromiso con la educación, y formación de futuros profesionales.

Finalmente, agradecemos a Yang Haodong por desarrollar un entorno de simulación que fue la base de nuestro trabajo, permitiéndonos realizar la comparación y validación de los algoritmos implementados.

Jean, Luis

Dedicatoria

Dedico este trabajo a Nancy y Felipe, mis padres, mis dos pilares, cuyo sacrificio y apoyo incondicional me han permitido llegar a estas instancias. A mí mismo, por ser perseverante y nunca rendirme. A Johanna, por ser mi escucha y ayuda cuando más lo necesitaba, y a mi abuelita Inés, por su apoyo a lo largo de mi vida universitaria.

Sobre todo, a mi querida amiga Doménica, quien siempre ha sido una fuerza extra y una motivación adicional. Tal como se lo prometí, este trabajo lleva consigo las memorias y recuerdos más lindos que compartí con ella. Donde sea que esté, espero que esté orgullosa de mí.

Agradecimientos especiales a Jean, mi compañero de tesis, por ser más que un amigo. Gracias por las noches interminables de trabajo, los momentos difíciles y las innumerables risas. Tu compañerismo y amistad han sido invaluables a lo largo de este proceso.

A mis amigos “shidisimos” y “telecoquets”, quienes han sido una fuente constante de alegrías y apoyo. Gracias por estar siempre a mi lado, por las risas compartidas y por ser un pilar fundamental en mi vida universitaria. Sin ustedes, este camino habría sido mucho más difícil y menos gratificante. Con todo mi cariño,

Luis

Dedicatoria

Dedico este trabajo a mi Tía Bertha, quien, aunque no podrá verme graduado, espero que se sienta orgullosa desde arriba. A mi abuela, por siempre preocuparse por mi bienestar durante mis estudios en la universidad. En especial, a mis padres, María y Miguel, por confiar en mí, nunca dejarme sentir derrotado y mantenerme en el camino correcto. A mi Tío Juan, por su apoyo tanto en lo cotidiano como en lo técnico. A mi prima Pauleth, Tio Vichi y mi Tía Rosa, por ser una parte importante de mi familia.

A Samantha, por estar a mi lado en las victorias y derrotas, apoyándome incondicionalmente con alguna razón que me mantuviera cuerdo. A mis amigos Lucas, Esteban, Erick, Juan Diego, David, Jhonathan, y Henrys, por ser los mejores compañeros que alguien puede tener y por compartir tantas penas, alegrías y enojos juntos. A Romanélia y Milena, por su apoyo y compañía lejos del agobio académico y de la vida. A todos ellos, y a quienes no puedo nombrar, siempre gracias. Son parte fundamental de este logro.

Jean

Abreviaciones y acrónimos

A* *A-Estrella (por sus siglas en inglés, A-Star)*. 21, 28, 29

ABS *Sistema de Frenos Antibloqueo (por sus siglas en inglés, Anti-lock Braking System)*. 19

ADAS *Sistemas de Asistencia Avanzada al Conductor (por sus siglas en inglés, Advanced Driver Assistance Systems)*. 36

AGV *Vehículo Guiado Automatizado (por sus siglas en inglés, Automated Guided Vehicle)*. 27–29

AMR *Robots Móviles Autónomos (por sus siglas en inglés, Autonomous Mobile Robots)*. 19

AUV *Vehículo Submarino Autónomo (por sus siglas en inglés, Autonomous Underwater Vehicle)*. 19

D* *D-Estrella (por sus siglas en inglés, D-Star)*. 21, 32

D*Lite *D-Estrella Lite (por sus siglas en inglés, D-Star Lite)*. 21, 30

FA *Algoritmo de Luciérnagas (por sus siglas en inglés, Firefly Algorithm)*. 29

GNSS *Sistema Global de Navegación por Satélite (por sus siglas en inglés, Global Navigation Satellite System)*. 28, 29

GPS *Sistema de Posicionamiento Global (por sus siglas en inglés, Global Positioning System)*. 26

JPS *Búsqueda de Punto de Salto (por sus siglas en inglés, Jump Point Search)*. 22

LiDAR *Detección y Rango de Luz (por sus siglas en inglés, Light Detection and Ranging)*. 30

PSO *Optimización por Enjambre de Partículas (por sus siglas en inglés, Particle Swarm Optimization)*. 29

RFID *Identificación por Radiofrecuencia (por sus siglas en inglés, Radio Frequency Identification)*. 27, 28

RL *Aprendizaje por Refuerzo (por sus siglas en inglés, Reinforcement Learning)*. 24, 52

RRT *Árboles de Exploración Rápida (por sus siglas en inglés, Rapidly-exploring Random Trees)*. 21

SA *Recocido Simulado (por sus siglas en inglés, Simulated Annealing)*. 29

SLAM *Localización y Mapeo Simultáneos (por sus siglas en inglés, Simultaneous Localization and Mapping)*. 26

UAV *Vehículo Aéreo No Tripulado (por sus siglas en inglés, Unmanned Aerial Vehicle)*. 19, 28–30, 86

UGV *Vehículo Terrestre No Tripulado (por sus siglas en inglés, Unmanned Ground Vehicle)*. 19, 28–30, 86

UWB *Banda Ultra Ancha (por sus siglas en inglés, Ultra-Wideband)*. 30

UWV *Vehículo Acuático No Tripulado (por sus siglas en inglés, Unmanned Water Vehicle)*. 19

VANET *Red de Vehículos Ad Hoc (por sus siglas en inglés, Vehicular Ad Hoc Network)*. 28

Generalidades

1.1. Introducción

El rápido avance en las redes vehiculares, ha generado soluciones para mejorar la conectividad y eficiencia en aspectos como el tráfico, seguridad en las carreteras y la conducción autónoma. Dentro de este contexto, las redes vehiculares sirven para resolver retos de movilidad tanto en ciudades como en áreas de interior. Este trabajo se centra en evaluar e implementar un algoritmo para diseñar rutas en entornos dinámicos y cambiantes, asegurando confiabilidad y eficiencia en la planificación de rutas.

El problema principal abordado es la necesidad de desarrollar métodos que permitan a los vehículos autónomos planificar y replanificar rutas de manera precisa y en tiempo real, adaptándose a cambios inesperados en el entorno. El algoritmo debe garantizar seguridad y eficiencia, dada la complejidad y variabilidad de los escenarios en los que los vehículos operan.

Para abordar este desafío, utilizamos MATLAB como herramienta de simulación por su versatilidad y amplia gama de funcionalidades. Desarrollamos los algoritmos D* Lite y *Q-learning*, combinando métodos heurísticos con métodos de aprendizaje por refuerzo. El algoritmo D* Lite es conocido por su capacidad de replanificación ante cambios en el entorno, mientras que el *Q-learning* permite a los vehículos mejorar sus decisiones de navegación a través de la experiencia.

Cada uno de los algoritmos ha sido detalladamente descrito, incluyendo su desarrollo, pruebas de funcionamiento y validación de parámetros. Se realizaron múltiples experimentos para evaluar su rendimiento en diversas condiciones. También se compararon los algoritmos desarrollados con otros algoritmos de planificación de rutas, evaluando métricas como el tiempo de ejecución, la longitud del camino y el costo del camino en función de la distancia. Estas pruebas se realizaron en entornos controlados y aleatorizados, añadiendo obstáculos que los algoritmos deben esquivar para encontrar la ruta óptima. Además, se generaron diagramas de flujo para ilustrar el funcionamiento de cada algoritmo, facilitando la comprensión de sus procesos y decisiones clave.

Esta investigación no solo busca desarrollar un nuevo enfoque para la planificación

de rutas, sino también validar su eficiencia en comparación con algoritmos existentes. Los resultados obtenidos proporcionan recomendaciones para futuras investigaciones y aplicaciones en el campo de los vehículos autónomos.

El documento se organiza de la siguiente manera: en el Capítulo 2 se proporciona un marco teórico y contextual para la investigación; en el Capítulo 3 se define la metodología utilizada, el entorno de trabajo y las herramientas empleadas; el Capítulo 4 presenta la generación y validación de los algoritmos *D* Lite* y *Q-learning*; el Capítulo 5 muestra los resultados obtenidos y su análisis, discutiendo los resultados del *benchmark* con otros algoritmos de planificación de rutas; finalmente, el Capítulo 6 presenta las conclusiones, recomendaciones y posibles líneas de investigación futuras, destacando las contribuciones de este trabajo al campo de la planificación de rutas para vehículos autónomos.

1.2. Objetivos

1.2.1. Objetivo general

Desarrollar e implementar estrategias de diseño de rutas y evasión de obstáculos en tiempo real para vehículos autónomos, mediante una evaluación de algoritmos y tecnologías actuales, con el fin de mejorar la eficiencia, seguridad y adaptabilidad en entornos dinámicos y variantes.

1.2.2. Objetivos específicos

- Realizar una revisión de la literatura sobre algoritmos de navegación autónoma y tecnologías relevantes en vehículos autónomos.
- Evaluar y comparar algoritmos de navegación autónoma, como A* (A estrella), Dijkstra y métodos basados en aprendizaje profundo.
- Diseñar un modelo de vehículo autónomo en entorno de simulación, considerando características de adaptabilidad y evasión de obstáculos en tiempo real.
- Realizar pruebas para evaluar el rendimiento de las estrategias de diseño de

rutas y evasión de obstáculos en las situaciones simuladas.

1.3. Alcance

El estudio se enfocará en el desarrollo e implementación de un algoritmo de planificación de rutas para vehículos autónomos, basándose en una comparativa exhaustiva de los métodos existentes. Este análisis comparativo evaluará la eficiencia, capacidad de procesamiento, aplicabilidad y viabilidad de los métodos en una diversidad de entornos. Estos entornos incluyen desde áreas urbanas hasta espacios controlados con obstáculos variables, proporcionando un amplio espectro de situaciones para probar la adaptabilidad y eficacia del algoritmo desarrollado.

Para garantizar la precisión y realismo de nuestras evaluaciones, se utilizarán simuladores avanzados que recrean con alta fidelidad los distintos contextos en los que los vehículos autónomos operarían. Este enfoque permitirá no solo probar la funcionalidad del algoritmo en situaciones ideales, sino también su robustez y adaptabilidad frente a una variedad de desafíos inesperados.

Para el desarrollo y evaluación del algoritmo, se empleará MATLAB, una plataforma reconocida por ofrecer un conjunto integral de herramientas que promueven una simulación y análisis de resultados con alta precisión. La decisión de usar MATLAB responde a un enfoque orientado a aprovechar sus capacidades avanzadas en la manipulación de inteligencia artificial y análisis de datos. Este marco de trabajo es necesario para avanzar en la detección de obstáculos y optimizar la toma de decisiones en tiempo real.

1.4. Justificación

La conducción autónoma constituye uno de los avances tecnológicos más prometedores y desafiantes de las últimas décadas, especialmente en entornos dinámicos como áreas urbanas, rurales y controladas. La capacidad de estos sistemas para adaptarse en tiempo real a obstáculos imprevistos y cambios en las condiciones ambientales es fundamental para su desarrollo y aplicación efectiva. Sin embargo, las limitaciones ac-

tuales en la planificación de rutas en escenarios complejos y cambiantes representan un obstáculo significativo que necesita ser abordado.

La presente investigación tiene como objetivo principal superar estas limitaciones mediante una evaluación exhaustiva de diversos métodos y algoritmos de planificación de rutas existentes. Esta evaluación se centrará en la factibilidad, efectividad en tiempo real y capacidad de respuesta ante cambios inesperados en el entorno. Se espera que los hallazgos de esta evaluación sirvan como fundamento para el desarrollo de un algoritmo mejor adaptado a las características dinámicas y variables de los entornos en cuestión.

Marco Teórico

En este capítulo se abordarán los conceptos teóricos relevantes que permitieron desarrollar el presente trabajo de titulación.

2.1. Robots Móviles Autónomos

La autonomía en sistemas móviles representa la capacidad de estos sistemas para tomar decisiones y realizar tareas sin la necesidad de intervención humana. Los avances tecnológicos han permitido la implementación de sistemas autónomos en vehículos actuales, incluyendo el control de crucero y *Sistema de Frenos Antibloqueo (por sus siglas en inglés, Anti-lock Braking System)* (ABS), marcando el inicio hacia una autonomía más completa, donde el vehículo no solo se desplaza sino que también realiza funciones complejas como la evasión de obstáculos y el estacionamiento automatizado [1].

Los *Robots Móviles Autónomos (por sus siglas en inglés, Autonomous Mobile Robots)* (AMR) se clasifican en función de su entorno operativo, con aplicaciones que van desde la inspección autónoma y la vigilancia hasta el mantenimiento [2]. Entre ellos se encuentran:

- Vehículos terrestres no tripulados (*Vehículo Terrestre No Tripulado (por sus siglas en inglés, Unmanned Ground Vehicle)* (UGV)): Utilizados en exploración terrestre y transporte en terrenos difíciles.
- Vehículos acuáticos no tripulados (*Vehículo Acuático No Tripulado (por sus siglas en inglés, Unmanned Water Vehicle)* (UWV)): Empleados en exploración y monitoreo de cuerpos de agua.
- Vehículos submarinos autónomos (*Vehículo Submarino Autónomo (por sus siglas en inglés, Autonomous Underwater Vehicle)* (AUV)): Utilizados para investigaciones submarinas y operaciones de rescate.
- Vehículos aéreos no tripulados (*Vehículo Aéreo No Tripulado (por sus siglas en inglés, Unmanned Aerial Vehicle)* (UAV)): Empleados en vigilancia aérea, entregas y misiones de búsqueda y rescate.

Existen dos términos importantes para este tipo de sistemas: autonomía y autarquía. La *autonomía* se refiere a la toma de decisiones independientes, mientras que la *autarquía* indica el suministro de energía del robot en entornos desafiantes. Para funcionar correctamente, es decir, navegar y realizar tareas en situaciones dinámicas, es necesario poseer características de adaptabilidad al entorno como movilidad adaptativa, percepción del entorno, adquisición de conocimiento, capacidad de interacción, seguridad y procesamiento en tiempo real [3].

La necesidad de detectar y evitar obstáculos, junto con la localización precisa y la planificación de rutas, son elementos que requieren una combinación de sensores (por ejemplo, LiDAR, cámaras, sensores de ultrasonido), algoritmos de fusión de datos y técnicas de inteligencia artificial para interpretar y actuar en el mundo real [2].

2.2. Algoritmos de Búsqueda

Un algoritmo de búsqueda se utiliza para explorar el conjunto de posibles estados de un problema con el propósito de encontrar una solución. Este tipo de algoritmo comienza desde un estado inicial y realiza iteraciones a través de distintos estados hasta encontrar la solución deseada.

Existen dos tipos principales de algoritmos de búsqueda: la búsqueda no informada y la búsqueda informada (heurística). Los vehículos autónomos hacen uso de algoritmos heurísticos o informados para estimar el costo de alcanzar su objetivo, dando prioridad a la exploración de estados que parecen más prometedores.

Como señala Cormen en su libro “Introduction to Algorithms” [4], los algoritmos de búsqueda permiten a las computadoras encontrar información de manera rápida en grandes volúmenes de datos. Estos algoritmos son esenciales para la planificación de rutas, la detección de obstáculos y la toma de decisiones en tiempo real. La eficiencia de estos algoritmos es fundamental para determinar la rapidez de respuesta de un vehículo.

2.2.1. Algoritmo A* (A estrella)

El algoritmo *A-Estrella (por sus siglas en inglés, A-Star)* (A*), es ampliamente utilizado en la planificación de rutas de vehículos autónomos, ya que su objetivo es buscar el camino más corto considerando variables como el tiempo, la distancia y la presencia de obstáculos. A* combina las ventajas de la búsqueda de *costo uniforme* (óptimo) y la *búsqueda voraz* (rápido), lo que permite tomar decisiones basándose en el costo del camino [5]. En A*, el costo total de un camino se calcula como la suma del costo acumulado desde el inicio (g-cost) y una estimación del costo restante hasta el objetivo (h-cost).

2.2.2. Árboles Aleatorios de Exploración Rápida (RRT)

Adecuado para espacios de búsqueda de grandes dimensiones y en entornos desconocidos o dinámicos, los *Árboles de Exploración Rápida (por sus siglas en inglés, Rapidly-exploring Random Trees)* (RRT), son algoritmos que construyen un espacio de búsqueda explorando aleatoriamente el entorno, generando un árbol que se expande gradualmente hacia áreas no exploradas [6]. Un ejemplo práctico de RRT sería su uso en la navegación de drones en entornos urbanos, donde debe evitar edificios y otros obstáculos.

2.2.3. D* (D estrella) y su variante D*-Lite

Diseñados para la planificación de rutas en entornos cambiantes en tiempo real, *D-Estrella (por sus siglas en inglés, D-Star)* (D*) y su variante *D-Estrella Lite (por sus siglas en inglés, D-Star Lite)* (D*Lite) son algoritmos que permiten la replanificación eficiente de rutas cuando se detectan cambios en el mapa o en las condiciones del entorno [7]. Estos algoritmos son cruciales para la navegación de robots en entornos dinámicos, como vehículos autónomos en tráfico urbano.

2.2.4. Algoritmo de Búsqueda de Puntos de Salto (JPS)

El algoritmo *Búsqueda de Punto de Salto (por sus siglas en inglés, Jump Point Search)* (JPS), es una mejora del algoritmo A* que optimiza la búsqueda de caminos al reducir la cantidad de nodos evaluados. En JPS, se utiliza una técnica de poda que omite nodos innecesarios y se enfoca solo en los nodos clave, lo que acelera el proceso de búsqueda considerablemente [8].

El algoritmo JPS es ampliamente utilizado en la planificación de rutas para robots móviles debido a su eficiencia en la reducción del tiempo de búsqueda en mapas de cuadrícula discretos. El JPS mejora la eficiencia de búsqueda del A* al eliminar nodos redundantes y conservar solo aquellos nodos críticos que son esenciales para la formación del camino óptimo [8].Su Funcionamiento es:

1. **Búsqueda Direccional**: El JPS realiza una búsqueda direccional desde el nodo inicial y expande en esa dirección hasta que se encuentra un obstáculo o un punto de salto. Esto reduce significativamente la cantidad de nodos evaluados en comparación con A*.
2. **Puntos de Salto Forzados**: Identifica los puntos de salto forzados, que son aquellos nodos donde el camino cambia de dirección debido a la presencia de obstáculos. Estos puntos son cruciales para la construcción del camino final y optimizado.
3. **Reglas de Poda**: El algoritmo aplica reglas de poda para ignorar nodos que no contribuyen al camino óptimo. Solo los nodos que pueden potencialmente llevar al camino más corto son evaluados, lo que mejora la eficiencia del algoritmo.

En un entorno urbano, el algoritmo JPS puede ser utilizado para planificar rutas eficientes para vehículos autónomos, minimizando el tiempo de cálculo y garantizando rutas seguras y optimizadas al evitar obstáculos y elegir los caminos más directos posibles [8].

El algoritmo JPS, con su enfoque en la poda de nodos y búsqueda direccional, proporciona una solución eficiente para la planificación de rutas en entornos discretos, garantizando caminos más cortos y tiempos de cálculo reducidos en comparación con

otros algoritmos tradicionales y sus variantes.

2.2.5. Algoritmo de Dijkstra

El algoritmo de Dijkstra es una técnica bien conocida para la planificación de rutas óptimas en grafos ponderados. Dado un grafo ponderado, el algoritmo de Dijkstra encuentra el camino más corto entre dos vértices cualesquiera del grafo siguiendo los siguientes pasos [9]:

1. Al principio, todos los nodos se marcan como no visitados. El nodo de inicio se establece como el nodo actual.
2. Se calcula la distancia temporal desde el nodo actual a todos sus nodos vecinos. Por ejemplo, si la distancia registrada desde cualquier nodo vecino al nodo inicial es menor que la distancia registrada previamente desde el nodo inicial a este nodo vecino, se sobrescribe la distancia previamente registrada a este nodo vecino. Cuando se terminan de verificar todos los vecinos del nodo actual, se marca el nodo actual como visitado. Un nodo visitado nunca será visitado de nuevo, y la distancia registrada del nodo visitado es definitiva y óptima.
3. Si todos los nodos han sido marcados como visitados, se termina el proceso. De lo contrario, se continúa con el nodo de menor distancia como el siguiente “nodo actual” y se continúa desde el paso 2.

Este algoritmo proporciona la solución al problema de planificación de rutas óptimas discretas en una malla cartesiana dada.

2.2.6. Algoritmo Theta*

El algoritmo Theta* es una variante de A* diseñada para encontrar caminos más cortos y realistas en comparación con A*. Theta* es un algoritmo de planificación de rutas que permite movimientos en cualquier ángulo, eliminando la restricción de movimiento a lo largo de los bordes de la cuadrícula. Durante la búsqueda, Theta* realiza comprobaciones de línea de visión en tiempo real para encontrar caminos más cortos, lo que implica que el algoritmo puede tomar caminos directos entre nodos siempre que no

haya obstáculos en el camino. Esto resulta en caminos más cortos y realistas. Theta* es especialmente útil en entornos donde los caminos más cortos no necesariamente siguen los bordes de una cuadrícula, como en la navegación de robots y videojuegos [10].

2.3. Aprendizaje por Refuerzo

El aprendizaje por refuerzo (*Aprendizaje por Refuerzo (por sus siglas en inglés, Reinforcement Learning)* (RL), por sus siglas en inglés) es una técnica de aprendizaje de máquina en la que un agente toma decisiones interactuando continuamente con su entorno para maximizar una recompensa acumulativa. A diferencia del aprendizaje supervisado, que proporciona ejemplos etiquetados por un supervisor externo, el agente en el aprendizaje por refuerzo no recibe instrucciones explícitas sobre qué acciones debe realizar. En su lugar, descubre las acciones mejor recompensadas mediante prueba y error. Dos características que distinguen al aprendizaje por refuerzo son la **búsqueda por ensayo y error** y la **recompensa diferida**. La recompensa diferida implica que la retroalimentación sobre la efectividad de las acciones no siempre es inmediata, por lo que el agente debe aprender a asociar ciertas acciones con recompensas futuras [11].

Para maximizar la recompensa, el agente debe preferir acciones que ya se hayan probado y demostrado ser efectivas. Sin embargo, también necesita explorar nuevas acciones para descubrir posibles mejoras en su desempeño futuro. Este equilibrio entre explotación y exploración es fundamental en el aprendizaje por refuerzo. No obstante, ninguna de estas estrategias puede seguirse exclusivamente sin comprometer el éxito de la tarea, ya que es un proceso estocástico en el que cada acción debe ser probada repetidamente para estimar de manera confiable su recompensa esperada [12].

La interacción entre el agente y el ambiente en el RL puede representarse de manera esquemática, como se muestra en la Figura 2.1. En este esquema, el agente recibe información sobre el estado del ambiente y basado en esta información, selecciona una acción. El ambiente, a su vez, responde a la acción del agente, proporcionando una nueva observación del estado y una recompensa. Este ciclo de interacción

continúa hasta que se alcanza un criterio de parada definido, como el logro de una recompensa acumulativa específica.

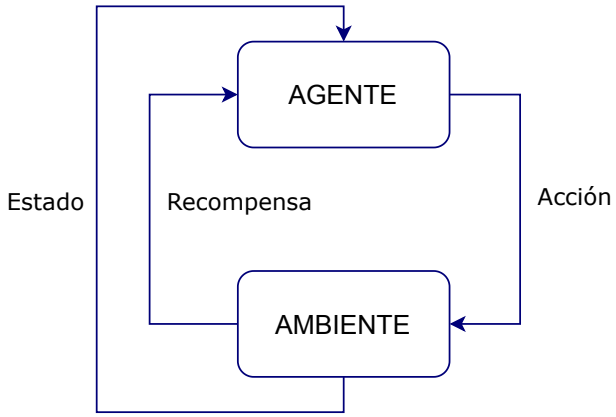


Figura 2.1: Interacción entre agente y ambiente en el aprendizaje por refuerzo

2.3.1. Q-learning

Q-learning es un algoritmo de aprendizaje por refuerzo que permite a un agente aprender a tomar decisiones óptimas a través de la interacción continua con su entorno, con el objetivo de maximizar una recompensa acumulativa. Este método es ampliamente utilizado por su simplicidad y eficacia en entornos donde actúa un solo agente. *Q-learning* es uno de los algoritmos más aplicados dentro del aprendizaje por refuerzo. Es una técnica fuera de política, es decir, que permite al agente aprender la calidad de las acciones independientemente de la política que está siguiendo para actuar [13].

La actualización del valor Q, que representa la **calidad de una acción en un estado dado**, se realiza mediante la ecuación 2.1 tomada de [13] [14].

$$Q(s,a) \leftarrow Q(s,a) + \alpha[R + \gamma max_{a'} Q(s',a') - Q(s,a)] \tag{2.1}$$

donde:

- $Q(s,a)$ es el valor Q actual.
- α es la tasa de aprendizaje y toma un valor entre 0 y 1.
- R es la recompensa recibida.

- γ es el factor de descuento, que reduce el valor de la recompensa con el tiempo.
- $maxQ(s', a')$ es el valor Q máximo esperado para el siguiente estado s'.

El agente selecciona una acción según una política y se mueve al siguiente estado utilizando la Ecuación 2.1. Este proceso se repite varias veces hasta que los valores de Q convergen, permitiendo al agente resolver el problema dado de manera óptima.

A pesar de que Q-learning sea un poderoso algoritmo para entornos de agente único, presenta limitaciones en términos de eficiencia de memoria y capacidad para resolver problemas con grandes espacios de estado-acción [14].

2.4. Navegación y Localización

La navegación implica determinar con exactitud la posición y velocidad de un vehículo en relación con una referencia conocida o durante la planificación de su trayecto [15]. Para lograr esto, es necesario comprender los distintos sistemas de coordenadas. Para lograr esto, es fundamental entender los diversos sistemas de coordenadas. En este contexto, los métodos de localización, como la geolocalización a través del *Sistema de Posicionamiento Global (por sus siglas en inglés, Global Positioning System)* (GPS), resultan indispensables. Localizar vehículos es uno de los retos más complejos debido al ruido presente en los sensores. Lamentablemente, la precisión de los sistemas GPS actuales es de varios metros. Además, la localización se utiliza para determinar la posición relativa entre un vehículo anfitrión y los objetos a su alrededor, lo cual permite evitar obstáculos [16].

Métodos como *Localización y Mapeo Simultáneos (por sus siglas en inglés, Simultaneous Localization and Mapping)* (SLAM) permiten la creación de un mapa y la estimación de la posición en un entorno desconocido, utilizando sistemas *basados en visión*. Los métodos basados en aprendizaje profundo han demostrado mejoras significativas en precisión, confiabilidad y robustez en comparación con los algoritmos SLAM tradicionales [17].

Los vehículos autónomos usan algoritmos para gestionar y controlar la navegación y las rutas, desplazándose eficientemente en cualquier entorno. Estos algoritmos impli-

can la planificación de rutas, calculando el trayecto óptimo desde un punto de inicio hasta un destino.

2.5. Trabajos Relacionados

Los vehículos no tripulados, como los drones, han experimentado un rápido desarrollo, influyendo en diversos aspectos de la sociedad. Actualmente, los vehículos no tripulados cumplen un rol significativo en campos que van desde la industria hasta la investigación, dando como resultado gratos beneficios para la sociedad. Su influencia continúa creciendo, acompañada de desafíos emergentes.

El sector industrial, donde los vehículos autónomos tienen mayores exigencias, se ha observado un considerable progreso a lo largo del tiempo. Investigadores, como se describe en [18], han identificado aspectos clave para la implementación de Vehículos de Guiado Autónomo (*Vehículo Guiado Automatizado (por sus siglas en inglés, Automated Guided Vehicle)* (AGV), por sus siglas en inglés), que incluyen la selección de un sitio operativo, la aplicación, el control y el entorno de software para la programación del vehículo. Estudios como el de Drira subrayan la importancia de tener rutas predefinidas [19], priorizando el rendimiento de producción ante posibles fallos. Gutta [20] propuso la división del espacio de trabajo en polígonos o celdas, con el objetivo de lograr una segmentación eficiente de las instalaciones, áreas de recolección, entrega, entre otras. Sin embargo, estas ideas presentan limitaciones en cuanto a su adaptabilidad, ya que están diseñadas para operar en entornos estáticos, donde se pasan por alto aspectos externos a dicho espacio. De igual manera, en 2017 Beinschob [21] presenta el despliegue de un AGV que funcionaba a partir del escaneo láser, para construir una representación virtual del entorno o mapa semántico, indicando el espacio libre a utilizar para crear la ruta.

En el ámbito de la navegación industrial, se ha empleado la tecnología *Identificación por Radiofrecuencia (por sus siglas en inglés, Radio Frequency Identification)* (RFID) y algoritmos basados en puntos de referencia [22], mientras que el uso de visión por computadora, como se describe en [23], ha demostrado ser eficaz para la detección de obstáculos y la planificación de rutas. Para el posicionamiento y navegación in-

dustrial se ha hecho uso de RFID, por su versatilidad y utilidad en la optimización de logística, las rutas se realizaban mediante algoritmos basados en puntos de referencia.

En diversos entornos, ya sea espacios cerrados o en carreteras, la fiabilidad y seguridad de los vehículos autónomos son aspectos importantes. Una estrategia efectiva, representada en [24], implica la delimitación de carriles mediante la integración de sensores en la vía. Estos sensores desempeñan un papel fundamental al ayudar al vehículo a construir su ruta. En este contexto, se implementan algoritmos basados en visión, y la generación de la ruta se realiza utilizando funciones matemáticas de Hermite.

En el ámbito de la salud, se ha desarrollado un modelo de vehículo específicamente desarrollado para el transporte de alimentos y suministros hospitalarios. Aunque este vehículo se encuentra en fase de prototipo, su evaluación se llevó a cabo mediante el uso del software Simio. Este AGV está equipado con sensores que le permiten detectar otros vehículos y evitar colisiones como se detalla en [25].

La programación efectiva de Vehículos de Guiado Autónomo (AGVs) requiere diversas herramientas. Un estudio comparativo [26] destaca sistemas como *KogMo-RTDB, ADTF, ROS*. ROS, debido a su robustez y versatilidad, emerge como una opción preferida en la programación de AGVs, ofreciendo una arquitectura modular y herramientas como RVIZ y GAZEBO para investigación y desarrollo robótico.

Los UGVs han demostrado utilidad en transporte diario, reconocimiento de selvas y exploración planetaria. La optimización de la navegación, abordada en [27] destaca la importancia de encontrar trayectorias libres de colisiones en entornos complicados. Métodos como A* (A-estrella), Rutas Probabilísticas (PRM) y Método del Campo Potencial (PFM) han sido empleados y planificados, pero la ejecución es donde surgen los desafíos.

La colaboración entre UGV y UAV se ha propuesto como una estrategia efectiva [27], utilizando sistemas *Sistema Global de Navegación por Satélite (por sus siglas en inglés, Global Navigation Satellite System)* (GNSS) y redes vehiculares (*Red de Vehículos Ad Hoc (por sus siglas en inglés, Vehicular Ad Hoc Network)* (VANET)s)

para facilitar la operación entre múltiples vehículos autónomos. El inconveniente de esta colaboración es que son dependientes el uno del otro y no funcionan de manera independiente. UAV genera un mapa de baja resolución para que el UGV planifique los movimientos.

MATLAB se ha destacado como una herramienta fundamental para la simulación en diversos estudios sobre planificación de rutas y movimientos de robots móviles. Por ejemplo, en el trabajo de [28], se realiza una comparación de varios algoritmos de planificación de rutas, incluyendo *Optimización por Enjambre de Partículas (por sus siglas en inglés, Particle Swarm Optimization)* (PSO), *Algoritmo de Luciérnagas (por sus siglas en inglés, Firefly Algorithm)* (FA), y *Recocido Simulado (por sus siglas en inglés, Simulated Annealing)* (SA). Este estudio concluye que el algoritmo FA es útil en aplicaciones que requieren operaciones de alta precisión.

Otra área de investigación relevante es el aprendizaje por refuerzo profundo, una mejora significativa sobre el aprendizaje básico de Q-learning. Este enfoque, como se muestra en [29], optimiza el proceso al calcular únicamente los valores de Q necesarios en lugar de toda la tabla Q. Este trabajo también emplea MATLAB como plataforma de simulación, demostrando su utilidad en la implementación y validación de algoritmos.

El algoritmo de Dijkstra, conocido por su aplicación en la búsqueda de caminos más cortos, ha sido objeto de numerosas optimizaciones. En el estudio [30], se compara el algoritmo Dijkstra tradicional con otros algoritmos como la colonia de hormigas, el algoritmo genético y A*. Utilizando MATLAB en un entorno cuadriculado, se destaca un Dijkstra mejorado que optimiza el tiempo de cálculo y la capacidad de recalcular rutas en presencia de obstáculos.

El algoritmo de A*, también es sujeto a mejoras, para abordar las limitaciones del algoritmo tradicional, tales como el largo tiempo de búsqueda y la presencia de múltiples nodos de expansión inválidos y de giro. En dónde esta mejora del algoritmo mejora la velocidad de búsqueda de rutas, reduce el número de giros y suaviza la trayectoria del AGV, en escenarios con obstáculos [31].

No solo deben depender de sistemas GNSS, ya que el UGV puede asumir la misión

de cartógrafo en un entorno desconocido y rastrear el estado del UAV, esto mediante sensores como cámara omnidireccional, *Detección y Rango de Luz (por sus siglas en inglés, Light Detection and Ranging)* (LiDAR), escaneo 3D y una radio *Banda Ultra Ancha (por sus siglas en inglés, Ultra-Wideband)* (UWB) que se vincula a un radio en el UAV. Como se detalla en [32], esta metodología se aplica en entornos subterráneos o espacios en condiciones desfavorables. La implementación de sistemas cooperativos, en los que el UGV genera un mapa del entorno desconocido y el UAV está equipado con sensores, surge como una estrategia innovadora para abordar desafíos como la evasión de colisiones y generación de rutas en entornos subterráneos.

Se logra la materialización de un vehículo capaz de desempeñar tareas en entornos dinámicos relacionados con el transporte, logística, mensajería o rescates en áreas de difícil acceso, según se describe en [33]. Este logro se ha alcanzado mediante la implementación del UAV “DJI TELLO”. Este vehículo aéreo utiliza un modelo de red neuronal convolucional debidamente entrenado para detectar diversas clases de obstáculos. Además, incorpora el algoritmo D*Lite (D-estrella Lite), reconocido como la elección óptima para la planificación de trayectorias, gracias a su capacidad para recalcular rutas, lo que lo convierte en una herramienta invaluable en escenarios dinámicos.

Tabla 2.1: Resumen de los trabajos relacionados

Referencia	Descripción del Trabajo
[18]	Identificación de aspectos clave para la implementación de Vehículos de Guiado Autónomo (AGV), incluyendo selección de sitio, aplicación, control y entorno de software.
[19]	Importancia de rutas predefinidas en AGVs para mejorar el rendimiento de producción ante posibles fallos.
[20]	Propuesta de división del espacio de trabajo en polígonos o celdas para segmentación eficiente de instalaciones y áreas de recolección/entrega.
[21]	Despliegue de AGV basado en escaneo láser para construir una representación virtual del entorno o mapa semántico.
[22]	Uso de tecnología RFID y algoritmos basados en puntos de referencia para navegación industrial.
[23]	Eficacia de la visión por computadora en la detección de obstáculos y planificación de rutas.
[24]	Delimitación de carriles mediante integración de sensores en la vía y uso de algoritmos basados en visión para generación de rutas.
[25]	Desarrollo de AGV para transporte de alimentos y suministros hospitalarios, evaluado con Simio, equipado con sensores para evitar colisiones.
[26]	Estudio comparativo de herramientas para programación de AGVs, destacando la robustez y versatilidad de ROS.
[27]	Optimización de navegación de UGVs para evitar colisiones en entornos complicados usando métodos como A*, PRM y PFM.
[28]	Comparación de algoritmos de planificación de rutas (PSO, FA, SA) en MATLAB, destacando la utilidad de FA en aplicaciones de alta precisión.
[29]	Aplicación de aprendizaje por refuerzo profundo en MATLAB, optimizando el proceso al calcular solo los valores de Q necesarios.
[30]	Comparación de Dijkstra con otros algoritmos como colonia de hormigas y A* en MATLAB, destacando un Dijkstra mejorado.
[31]	Mejoras en el algoritmo A* para aumentar la velocidad de búsqueda, reducir giros y suavizar la trayectoria en AGVs.
[32]	Estrategia de colaboración entre UGV y UAV utilizando GNSS y VANETs, con UGV asumiendo el rol de cartógrafo en entornos desconocidos.
[33]	Implementación de UAV DJI TELLO equipado con D*Lite para detección de obstáculos y planificación de rutas en entornos dinámicos.

Nuestra propuesta se centra en la implementación y validación de los algoritmos *D* Lite* y *Q-learning* en MATLAB para la planificación de rutas en vehículos autónomos. A diferencia de los trabajos enumerados en la Tabla 2.1, que en su mayoría han utilizado otras plataformas como *ROS* o *Simio*, nuestro enfoque en MATLAB responde a la escasez de trabajos que han implementado estos algoritmos en dicha plataforma. Además, estos algoritmos fueron concebidos para considerar la evasión de obstáculos desde su diseño inicial, lo que asegura una mayor efectividad y adaptabilidad en la planificación de rutas. La combinación de *D* Lite* y *Q-learning* permite abordar tanto la planificación de rutas óptimas en entornos dinámicos como el aprendizaje adaptativo en tiempo real, proporcionando así una solución robusta a los desafíos emergentes en la navegación autónoma.

Definición de Entorno

Este capítulo describe las consideraciones tomadas en cuenta para definir el entorno de simulación utilizado en la evaluación e implementación de un algoritmo para el diseño de rutas en ambientes dinámicos y variantes para vehículos autónomos. En primer lugar, se seleccionó el entorno de simulación más adecuado que facilite tanto la implementación de algoritmos como la interacción con el usuario. También se consideraron factores como la necesidad de un entorno visualmente atractivo, que no solo facilite la observación del trayecto generado, sino que también permita la interactividad para añadir obstáculos durante la simulación.

Para llevar a cabo las simulaciones, se definió MATLAB como herramienta principal, debido a sus capacidades de modelado y simulación. Inicialmente, se consideró el *toolbox* "`robotics-toolbox-matlab`" desarrollado por Peter Corke [34] como entorno de simulación. Aunque esta herramienta es poderosa para simular dispositivos robóticos como brazos, cuadricópteros, uniciclos, incluso cuenta con un ejemplo del algoritmo D* implementado, esta fue descartada porque no ofrecía la interactividad y visualización intuitiva requeridas.

Finalmente, se encontró un entorno de pruebas en un proyecto existente disponible en *GitHub*, cuyo autor es Yang Haodong [35]. Este proyecto no solo considera la interactividad y un funcionamiento intuitivo, sino que también reutiliza algunos componentes de `global_planner`, que incluyen métodos de búsqueda por grafos y muestreo, con el objetivo de realizar una comparación con los algoritmos desarrollados.

El algoritmo D* Lite propuesto se centra en la optimización del tiempo de ejecución mediante la simplificación de la complejidad del código. En la Tabla 3.1, se puede observar que el algoritmo D* Lite reduce casi 100 líneas de código en comparación con el algoritmo D* original, así como el número de condicionales y subfunciones. Esta reducción se logra gracias a la optimización en la implementación y se tiene en cuenta el uso de la distancia de Manhattan, la cual evita problemas con movimientos diagonales entre obstáculos, reduciendo el tiempo de procesamiento.

Adicionalmente, la estrategia de comenzar a calcular desde el nodo objetivo permite que, al replanificar tras encontrar un obstáculo, la evaluación se realice desde el ob-

jetivo hasta el estado actual del nodo, en lugar de reiniciar desde el punto de partida. Esta técnica reduce considerablemente la carga computacional en entornos dinámicos. También se ha considerado la eliminación de listas innecesarias de nodos, lo que implica que no es necesario calcular el costo de todos los nodos, sino únicamente de aquellos relevantes. Este enfoque contribuye a una mayor eficiencia y rapidez en la determinación de rutas.

Por otro lado, nuestra propuesta basada en *Q-learning* destaca por su capacidad de entrenarse una sola vez para mapas grandes, proporcionando buenos resultados a pesar de su simplicidad. El algoritmo de *Q-learning* utiliza una tabla Q que se actualiza durante el entrenamiento.

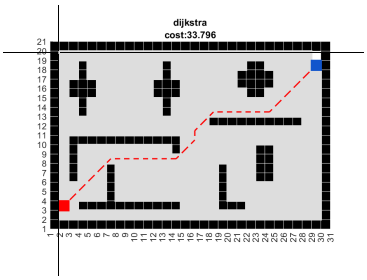
Tabla 3.1: Comparación de Algoritmos dentro del Entorno de Simulación

Parámetro	A*	D* Lite	D*	Dijkstra	Q-learning	GBFS	JPS	Theta*	Lazy Theta*	RTT
Estensión de Código	135	161	266	130	90	131	221	226	246	135
Número de Condicionales	14	18	37	14	5	13	28	32	34	22
Número de Subfunciones	4	5	7	2	1	4	6	6	6	6
Número de Interrupciones (break/return)	2	3	4	2	2	2	4	3	3	4
Problema con movimientos diagonales con obstáculo	Si	No	Si	Si	No	Si	Si	Si	Si	Si
Capacidad de Adaptación Dinámica	Baja	Alta	Media	Baja	Alta	Baja	Media	Media	Media	Baja
Tipo de Movimiento Permitido	8 direcciones	8 direcciones	8 direcciones	8 direcciones	8 direcciones	8 direcciones	8 direcciones	8 direcciones	8 direcciones	-
Velocidad de Convergencia	Media	Alta	Alta	Baja	Variable	Alta	Alta	Alta	Alta	Baja
Robustez en Entornos Dinámicos	Baja	Alta	Alta	Baja	Alta	Baja	Media	Alta	Alta	Alta
Necesidad de Replanificación Completa	Alta	Baja	Muy Baja	Alta	No aplica	Alta	Media	Baja	Baja	Media
Tiempo de Inicialización	Medio	Bajo	Medio	Medio	Alto (Entrenamiento)	Bajo	Bajo	Medio	Medio	Medio
Facilidad de Implementación	Media	Media-Baja	Compleja	Media	Compleja	Media	Media	Compleja	Compleja	Media
Escalabilidad a Mapas Grandes	Media	Alta	Media	Baja	Alta	Media	Alta	Media	Media	Baja
Heurística	Sí	Sí	Sí	No	No	Sí	Sí	Sí	Sí	No
Método de Búsqueda	Basado en costo y heurística	Basado en costo y heurística	Basado en costo y heurística	Basado en costo	Aprendizaje por refuerzo	Basado en heurística	Basado en salto de puntos	Basado en costo y heurística	Basado en costo y heurística	Muestro Aleatorio
Persistencia de Datos	No	No	Sí	No	Sí	No	No	No	No	No
Replanificación Local	No	Sí	Sí	No	No aplica	No	Sí	No	No	Si
Heurística Calculada	Distancia Euclidiana	Distancia Manhattan	Distancia Euclidiana	N/A	N/A	Distancia Euclidiana	Distancia Euclidiana	Distancia Euclidiana	Distancia Euclidiana	-
Manejo de Vecinos	8 Vecinos	4 Vecinos	8 Vecinos	8 Vecinos	8 Vecinos	8 Vecinos	8 Vecinos	8 Vecinos	8 Vecinos	-
Estructura de Datos Principales	Listas de prioridad	Listas de prioridad, Matrices	Listas de prioridad, Matrices	Listas de prioridad	Tablas Q	Listas de prioridad	Listas de prioridad, Matrices	Listas de prioridad	Listas de prioridad	Listas de Muestreo
Actualización de Nodos	Actualización de costos y heurística	Actualización dinámica de rhs y g	Actualización dinámica de costos y heurística	Actualización de costos	Actualización de tabla Q	Actualización basada en heurística	Actualización de puntos de salto	Actualización dinámica de costos y heurística	Actualización basada en visibilidad y heurística	Dinámica
Capacidad de Optimización Local	Baja	Alta	Alta	Baja	Alta	Baja	Alta	Alta	Alta	Alta
Nodos Analizados	Solo relevantes	Solo relevantes	Todos en la primera iteración	Todos	Todos	Solo relevantes	Solo relevantes	Solo relevantes	Solo relevantes	Todos
Ajuste en Tiempo de Ejecución	Ajustable según la heurística	Ajustable según el tamaño del mapa	Ajustable según el tamaño del mapa	Fijo	Ajustable según los parámetros de aprendizaje	Ajustable según la heurística	Ajustable según los puntos de salto	Ajustable según la heurística	Ajustable según la heurística	Ajustable según el tamaño del mapa

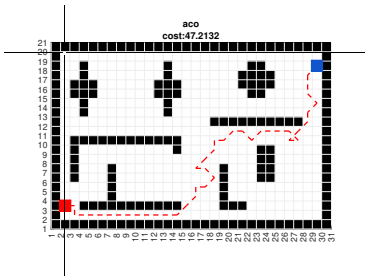
En resumen, nuestra propuesta ofrece soluciones más ligeras y rápidas al reducir la extensión del código, simplificar las estructuras condicionales y subfuncionales, y adoptar metodologías que priorizan la relevancia de los nodos y la eficiencia en la replanificación de rutas. Estos enfoques permiten una ejecución más eficiente y robusta en entornos dinámicos y variantes.

En la Figura 3.1 se presentan simulaciones para los algoritmos disponibles en el entorno de simulación definido. Es evidente que el entorno de simulación es bastante intuitivo, presentando los nodos inicial y final como bloques de color azul y rojo. El

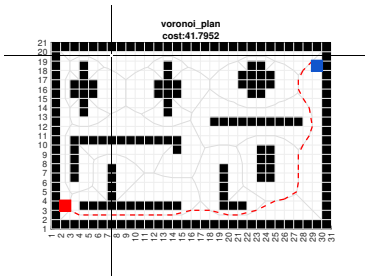
camino generado por los diferentes algoritmos se muestra de color rojo y existe la posibilidad de añadir un obstáculo en cualquier punto del mapa mediante un *clic* con el cursor. En gris se presenta una lista denominada *expand* que representa los nodos procesados por cada uno de los algoritmos disponibles y en la parte superior se muestra el costo asociado a la determinación de la trayectoria generada por los diferentes algoritmos. El término "*cost*" que aparece durante la simulación de los algoritmos se refiere al costo total acumulado del camino encontrado por el algoritmo, desde el nodo inicial hasta el nodo objetivo. Este costo puede ser interpretado como la suma de los valores asignados a cada movimiento o paso en el trayecto, los cuales pueden depender de la distancia recorrida, el tipo de terreno o cualquier otra métrica definida en el algoritmo para evaluar la eficiencia y viabilidad del camino.



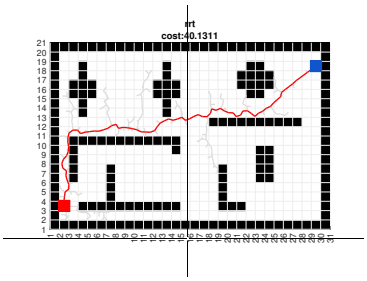
(a) *Resultado del algoritmo Dijkstra*



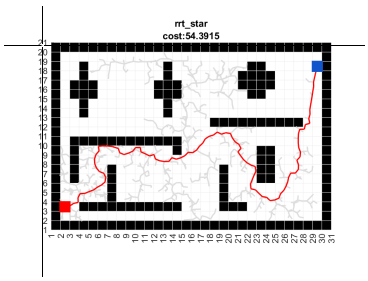
(b) *Resultado del algoritmo ACO*



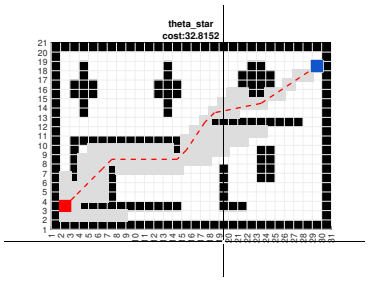
(c) *Resultado del algoritmo Voronoi*



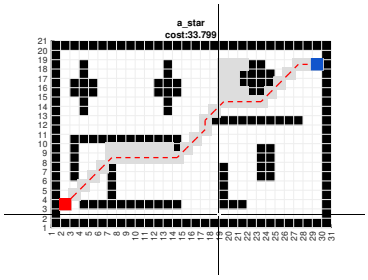
(d) *Resultado del algoritmo RRT*



(e) *Resultado del algoritmo RRT**



(f) *Resultado del algoritmo Theta**



(g) *Resultado del algoritmo A**

Figura 3.1: Entorno de simulación elegido con diferentes algoritmos

Si bien el entorno de simulación elegido presenta la interactividad y visualización deseada, todavía era necesario implementar una simulación más cercana a la realidad,

teniendo en cuenta entornos dinámicos. Para ello, se desarrolló un entorno 3D que incluye no solo la visualización de la trayectoria, sino también un móvil siguiendo dicha trayectoria y realizando una replanificación en caso de encontrar algún obstáculo. El desarrollo de este entorno “3D en MATLAB”, aunque no es nuestro enfoque principal, se basa en el proyecto desarrollado por Yang Haodong [35] mejorando la representación a una dimensión tridimensional. El código carga un mapa y establece puntos de inicio y fin. Utiliza los algoritmos del proyecto, así como los desarrollados, para encontrar la ruta óptima. Las funciones `plot_grid_3d`, `plot_expand_3d`, `plot_path_3d` y `draw_cubo` son necesarios para la visualización 3D, donde `plot_grid_3d` dibuja el mapa, `plot_expand_3d` muestra los nodos expandidos, `plot_path_3d` dibuja la ruta y `draw_cubo` crea los obstáculos y los posiciona en el mapa. Durante la simulación, se actualiza la posición del móvil y se verifica que el siguiente paso no sea un obstáculo. Si se encuentra un obstáculo, se replanifica la ruta usando el algoritmo seleccionado, lo que permite una navegación adaptativa en tiempo real. En la Figura 3.2 se presenta un ejemplo de la representación realizada. En el entorno se muestra de colores rojo y azul los nodos ‘*inicio*’ y ‘*objetivo*’, con una línea roja el trayecto definido por el algoritmo, en azul el móvil que recorre dicho trayecto y con puntos verdes la lista de expansión para los diferentes algoritmos.

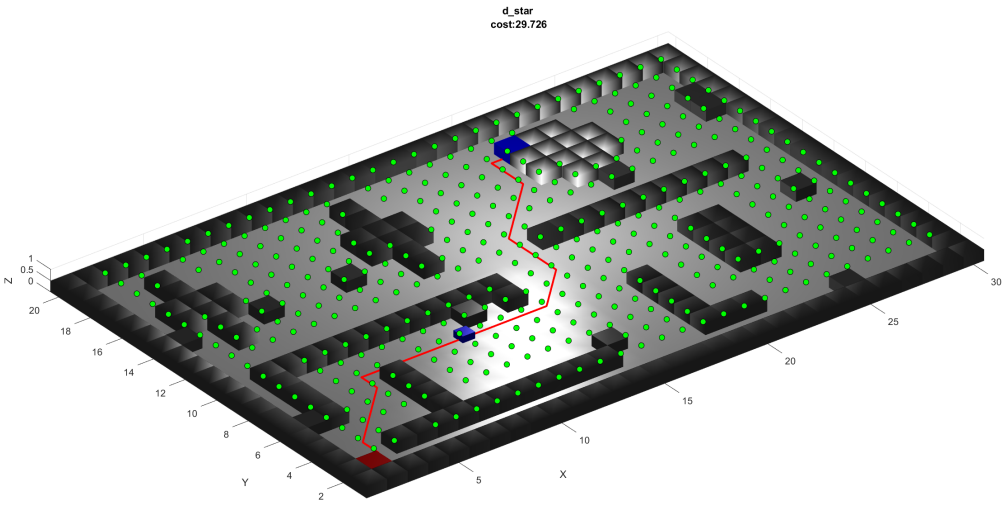


Figura 3.2: Entorno 3D desarrollado

Desarrollo de Algoritmos

En esta sección se presenta el desarrollo de las soluciones para la determinación de trayectorias desde un nodo de partida hacia un nodo objetivo.

4.1. D* Lite

En aplicaciones de navegación de vehículos terrestres, como los sistemas de conducción autónoma y *Sistemas de Asistencia Avanzada al Conductor (por sus siglas en inglés, Advanced Driver Assistance Systems)* (ADAS), se busca resolver problemas de planificación de trayectorias utilizando algoritmos sencillos y eficientes. Estos algoritmos permiten determinar rutas óptimas desde un punto inicial hasta un destino final, evitando obstáculos y minimizando el costo de desplazamiento en términos de tiempo, distancia o energía. La implementación de estos algoritmos garantiza la integridad y navegación de vehículos en entornos dinámicos y complejos. El algoritmo *D** se basa en heurísticas, que se pueden definir como una función que proporciona una estimación del costo desde un nodo inicial hasta el nodo objetivo.

Antes de presentar el algoritmo desarrollado, se expone la lógica seguida para su elaboración.

4.1.1. Lógica de Funcionamiento

4.1.1.1. Determinación de Heurísticas

El paso inicial del algoritmo es la determinación de las heurísticas desde el nodo inicial. En muchos algoritmos revisados para la determinación de estas heurísticas se usa la distancia euclidiana. Sin embargo, cuando se presentaban dos obstáculos en diagonal, como se muestra en la Figura 4.1, el vehículo no evadía dicho escenario.

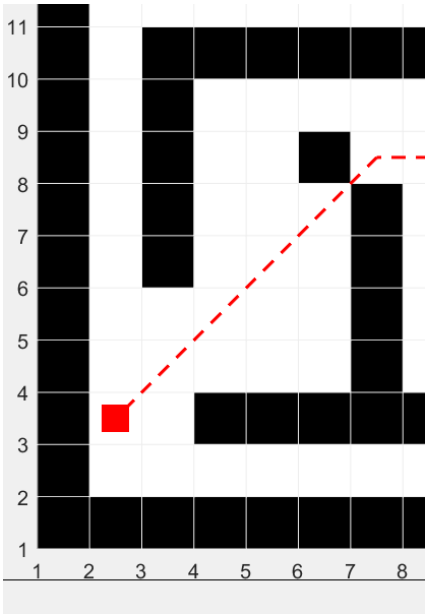


Figura 4.1: Problema de evasión de obstáculos al usar la distancia euclidiana

Este problema surge debido a la asignación de costos iguales a todos los posibles pasos que puede dar el vehículo. La estrategia adoptada consiste en sustituir el cálculo de la heurística mediante la distancia euclidiana por la distancia Manhattan.

La distancia Manhattan, como su nombre indica, se originó del análisis de urbes modernas que básicamente son cuadras rectangulares. En estas urbes, aunque la distancia de esquina a esquina es la más corta, es inviable realizar el desplazamiento diagonal debido a la presencia de edificios; resultando en un desplazamiento en forma de escalera, equivalente a avanzar en línea recta y girar en la esquina deseada.

Antes de presentar la función implementada, revisemos cómo funciona la determinación de la distancia Manhattan dentro de una cuadrícula sin obstáculos, presentada en la Figura 4.2. Se inicia definiendo un punto de partida que en este caso es la mitad de la cuadrícula y tiene como heurística el valor de cero. Desde ahí se incrementa de paso en paso hacia arriba, abajo, izquierda y derecha (en forma de cruz) como se detalla en color naranja en la Figura 4.2.

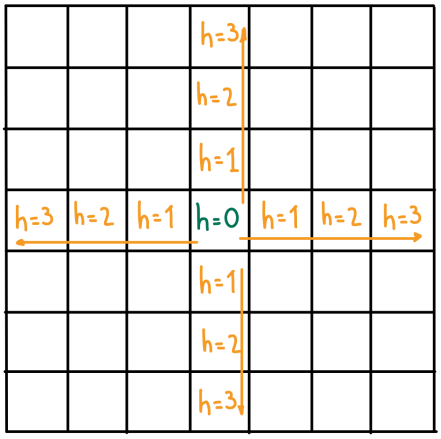


Figura 4.2: Cálculo de primeras heurísticas utilizando la distancia Manhattan

Finalmente, para llenar los valores faltantes, se toma una horizontal o vertical y se avanza de un lado al otro con un paso de uno, obteniendo así las distancias Manhattan para cada punto, como se presenta en la Figura 4.3.

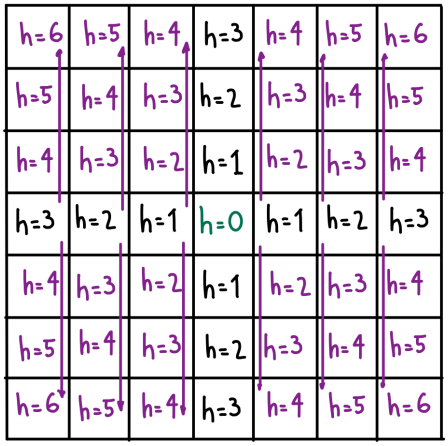


Figura 4.3: Relleno de heurísticas utilizando la distancia Manhattan

En la Figura 4.4 se presenta el valor de la distancia Manhattan para los desplazamientos diagonales, donde se aprecia que los costos aumentan de dos en dos, es decir, que los movimientos en diagonal tienen un costo mayor que los desplazamientos lineales. Por lo tanto, el vehículo en cuestión deja los movimientos diagonales como última opción para llegar al objetivo, minimizando así el problema representado en la Figura 4.1.

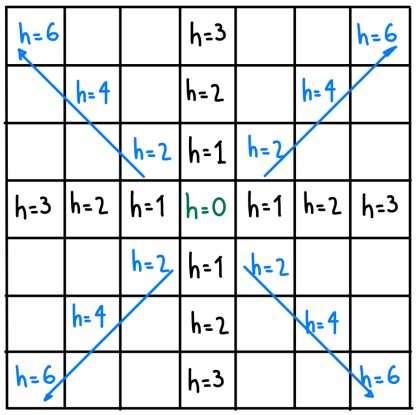


Figura 4.4: Cálculo de heurísticas diagonal utilizando la distancia Manhattan

4.1.1.2. Funcionamiento del Algoritmo

Una vez comprendida la definición de heurísticas, es fundamental entender el funcionamiento del algoritmo D* Lite antes de presentar la codificación. Los nodos inicial y final se representan en verde y rojo, respectivamente. El proceso del algoritmo se puede dividir en los siguientes pasos:

1. **Determinación de heurísticas**: Antes de ejecutar el algoritmo, se deben determinar las heurísticas para todos los nodos desde el nodo inicial, utilizando la distancia Manhattan, como se ilustra en la Figura 4.5a.

2. **Inicialización de valores**: Se inicializan los valores de *rhs* (costo desde el nodo objetivo hasta el inicio) y *g* (costo actual conocido desde el nodo de inicio hasta cualquier nodo) en infinito, como se muestra en la Figura 4.5b.

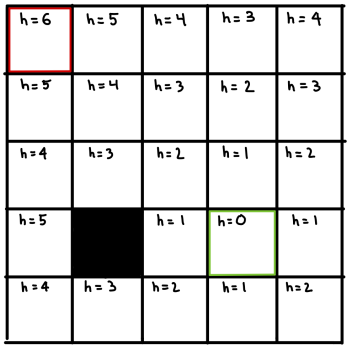
3. **Inicio de iteraciones**: Las iteraciones del algoritmo comienzan desde el nodo objetivo, donde el valor de *rhs* se establece en cero utilizando la distancia Manhattan. A continuación, se calcula un vector clave (k), que tiene como primer valor la suma de la heurística (h) y *rhs*, y como segundo valor el *rhs* del nodo, como se ilustra en la Figura 4.5c.

4. **Cálculo de *rhs* y *k***: En las siguientes iteraciones, se calcula tanto el valor *rhs* como el vector *k* para los posibles movimientos de un solo paso que puede dar el vehículo. Por ejemplo, en la Figura 4.5d, los posibles pasos desde el nodo

objetivo son hacia la derecha, hacia la abajo y en diagonal hacia abajo (representados con una línea azul entrecortada). Los valores de *rhs* se calculan con la distancia Manhattan desde el nodo analizado, y se omite el cálculo si los nodos futuros ya tienen asignado un valor de *rhs* (representados con una línea negra entrecortada cuando ya está calculado). Los valores del vector *k* se calculan de la misma manera (suma de la heurística *h* con *rhs*, seguida del propio valor *rhs* de cada nodo).

5. **Selección del nodo con menor *k***: En la siguiente iteración, presentada en la Figura 4.5e, se eligen siempre los menores valores posibles del vector *k*, priorizando tanto la suma de *h* con *rhs* como solo *rhs*. Una vez elegido el nodo con menor vector *k*, se define el valor de *g* como el valor del vector *rhs* de ese nodo, y se vuelven a calcular los vectores *k* para los posibles pasos que el vehículo puede dar desde este nodo. Considerando que posiblemente ya existen nodos calculados (en este caso, el de la derecha y la diagonal superior), no se deben calcular nuevamente con la distancia Manhattan. En caso de tener nodos con el mismo vector *k*, se puede elegir cualquiera, dado que, si nos alejamos del objetivo, elegir el menor vector *k* resulta en la ruta más corta hacia el nodo inicial. Estas iteraciones repetitivas se presentan en las Figuras 4.5f, 4.5g, 4.5h y 4.5i, hasta llegar al nodo inicial que tiene un vector *k* con valores iguales.

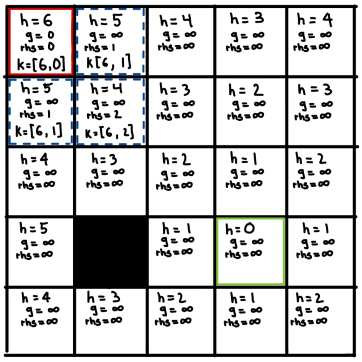
6. **Ventajas de la distancia Manhattan**: En la iteración presentada en la Figura 4.5g, se evidencia la ventaja de usar la distancia Manhattan en lugar de la euclidiana. Si se usara la distancia euclidiana, el camino más corto realizaría un desplazamiento diagonal hacia abajo debido a que todos los posibles pasos tienen el mismo costo, pero en un entorno real pasaría igualmente por el nodo de la derecha, dado que el paso directo es inviable. Al calcular la distancia Manhattan para cada paso, los movimientos diagonales tienen el doble de costo que los lineales. Por esta razón, en esta implementación, los movimientos que el algoritmo considera son los más adecuados basados en aplicaciones reales.



(a) *Cálculo heurísticas (Distancia Manhattan)*

(b) *Inicialización del Algoritmo D* Lite desarrollado*

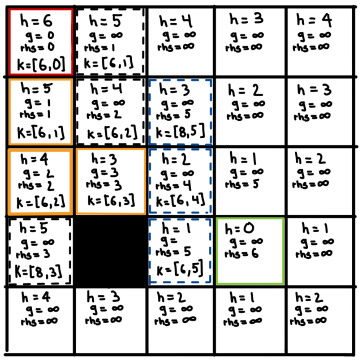
(c) *Primera Iteración del Algoritmo D* Lite*



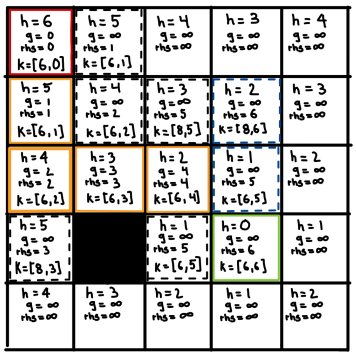
(d) *Segunda Iteración del Algoritmo D* Lite*

(e) *Tercera Iteración del Algoritmo D* Lite*

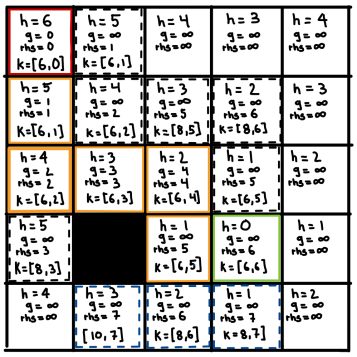
(f) *Cuarta Iteración del Algoritmo D* Lite*



(g) *Quinta Iteración del Algoritmo D* Lite*



(h) *Sexta Iteración del Algoritmo D* Lite*



(i) *Séptima Iteración del Algoritmo D* Lite*

Figura 4.5: Funcionamiento Conceptual del Algoritmo D Lite*

4.1.2. Desarrollo del Algoritmo D* Lite

El objetivo de la implementación del algoritmo D* Lite es simplificar la complejidad del código, disminuir el tiempo de procesamiento y lograr resultados similares o mejores que los algoritmos ya implementados.

4.1.2.1. Función principal

En la Figura 4.6,se presenta el diagrama de flujo de la función principal del desarrollo del algoritmo D* Lite. La función, para ser válida dentro del entorno de simulación, debe dar como resultado el camino resultante, una bandera para saber si existe o no un camino hacia el destino, el costo de dicho camino y una matriz denominada $expand$ que representa los nodos procesados dentro del mapa. Como entrada, la función recibe únicamente el mapa y las coordenadas del nodo inicio y del objetivo.

El algoritmo inicia definiendo las variables de entrada (`sstart, sgoal` y `map`), se calcula el número total de filas y columnas, y se inicializan las matrices g, rhs y k_mat con valores infinitos del mismo tamaño que el mapa. El algoritmo, como se explicó teóricamente, empieza definiendo como cero el valor para rhs en la ubicación del nodo objetivo, y sigue con el cálculo de las heurísticas para todo el mapa desde el nodo de inicio.

Con la matriz de heurísticas h , se puede definir la lista U. La lista U es una lista dinámica de nodos que es útil para identificar los nodos vecinos y tener un registro dinámico de los mismos. El formato de esta matriz se presenta en la Tabla 4.1. Los primeros dos componentes son las coordenadas del nodo, seguidas de la clave calculada para cada uno de ellos.

Tabla 4.1: Formato Matriz lista

Coordenadas Nodo		key	
coordenada 1	coordenada 2	k1	k2
...	...	...	...
coordenada n	coordenada m	kn	km

El nodo inicial de la lista U es el nodo objetivo, añadiendo las coordenadas del nodo seguidas del cálculo vector k. Una vez inicializada la lista U, se definen los condicionales iterativos que se repetirán mientras la lista U no esté vacía o el nodo actual analizado sea el nodo de inicio (teniendo en cuenta también que los valores de rhs y g para ese nodo deben ser los mismos).

La lógica de las iteraciones inicia buscando un "idx" que es el índice o la posición en la que se encuentra el menor primer valor del vector k. Una vez se determina el índice

extraemos las coordenadas de dicho nodo para asignarlas como nuestro nodo actual, y también se extrae un vector k_old correspondiente al vector k de dicho nodo.

El siguiente paso es calcular el vector k para el nodo actual y almacenar los valores de dicho vector dentro de la matriz k_mat en la posición actual. El primer condicional está planteado pensando en cambios en el entorno, si el valor almacenado en la lista U es menor al valor actual calculado, se reinserta dicho nodo a la lista U para su nuevo análisis.

El segundo condicional es el más importante de todos y se activa siempre que el valor de g en la coordenada actual es mayor que el rhs en la misma coordenada. Esto sucede cuando los vecinos en g, no están definidos y tienen valor infinito. Una vez se identifica que no está definido el valor de g, se define en el nodo actual como el valor de rhs en la coordenada actual y se obtienen los vecinos de dicho nodo.

Una vez que se cuenta con la lista de nodos vecinos, para cada uno de los vecinos se realiza lo siguiente:

- Verificar que el nodo vecino sea transitable y no sea el nodo objetivo.
- Actualizar su valor rhs al mínimo entre su valor actual rhs y el valor de g del nodo actual más 1. Esto se hace con la línea de código,
  $rhs(vecino) = min(rhs(vecino), g(actual) + 1)$.
- Actualizar dicha coordenada en la lista de nodos abiertos U utilizando la función `updateNode`. Esto asegura que el nodo vecino sea considerado en futuras iteraciones del algoritmo.

En el último condicional se verifica si el valor de g del nodo actual no es infinito, lo que indica que ha sido procesado previamente. Si es así, establece g del nodo actual como infinito para invalidar su costo acumulado, indicando que ya no es parte de un camino válido. Esto se hace para invalidar caminos cuando se descubre que ya no son óptimos debido a cambios en el entorno, como la aparición de nuevos obstáculos. Luego, se obtienen los vecinos del nodo actual y, para cada uno de ellos, si el vecino es transitable y su valor de rhs depende del valor de g del nodo actual, se establece rhs del vecino como infinito siempre que este vecino no sea igual al nodo objetivo, indicando que su camino también es inválido. Finalmente, se actualizan estos nodos en la cola

de prioridad U para que sean reevaluados en futuras iteraciones, asegurando que el algoritmo encuentre nuevos caminos óptimos y válidos.

Una vez la lista U está vacía o el nodo analizado es el nodo inicial, se asigna el valor de g del nodo inicio al valor de rhs correspondiente. En este punto, ya tenemos definidas las matrices de g y rhs para ciertos nodos. Estos nodos se pueden entender como nodos de interés desde el nodo objetivo hasta el nodo inicial, lo que evita perder tiempo en el cálculo innecesario de nodos que no son relevantes para determinar el camino. Con las matrices de g y rhs ya definidas, simplemente se reconstruye el camino con la función presentada en la Figura 4.8. Finalmente, validamos si el camino está vacío o no para dar un valor booleano a la bandera.

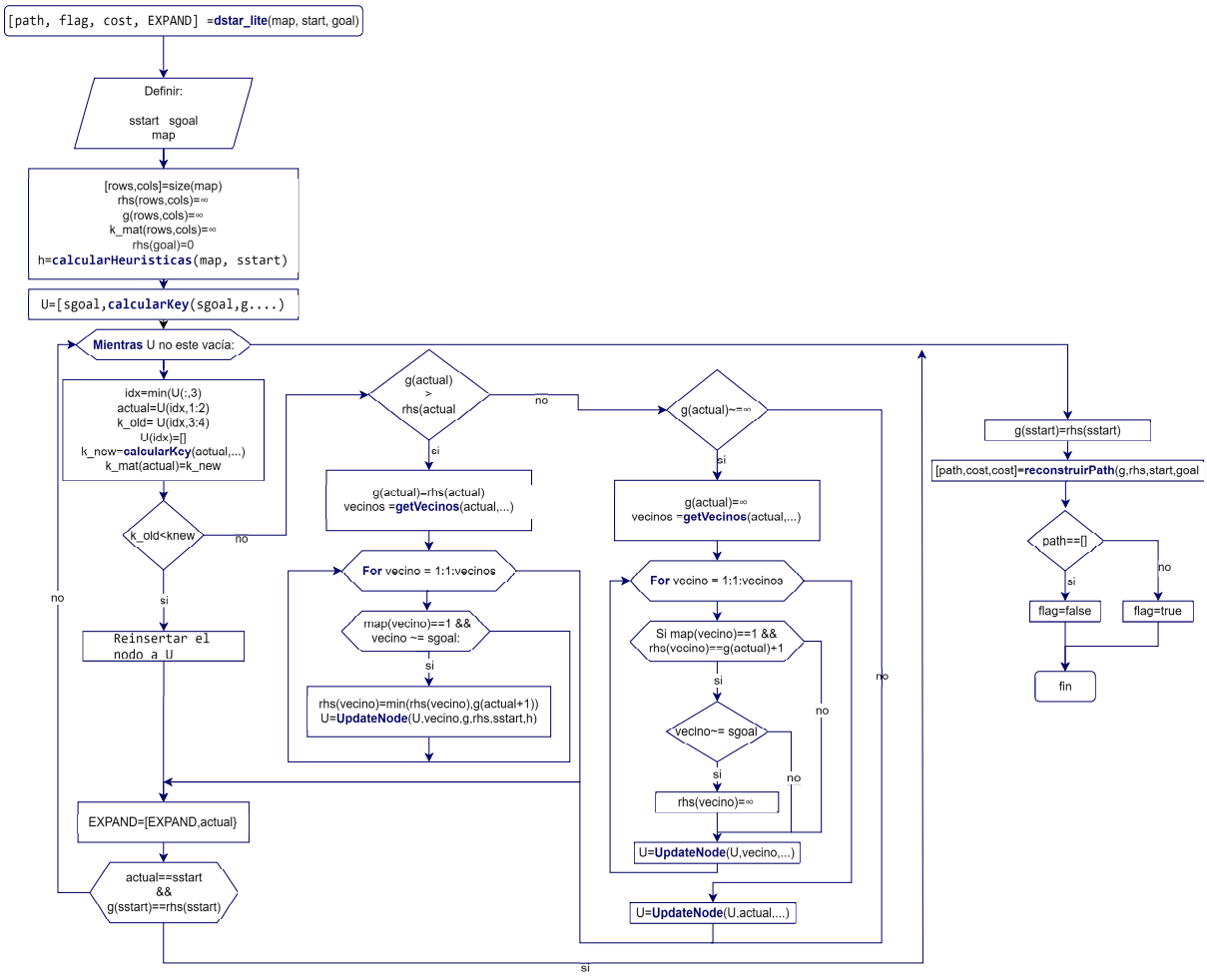


Figura 4.6: Diagrama de flujo función principal dstar_lite

Para validar el algoritmo, comparamos los resultados mostrados en la Figura 4.7 con los obtenidos conceptualmente en la Figura 4.5. Se observa que ambos utilizan el mismo mapa con un único obstáculo. Al analizar los valores de g, h, rhs, y k en cada

nodo, se confirma que los resultados implementados en el algoritmo coinciden con los valores teóricos. La discrepancia principal radica en los nodos procesados en el algoritmo, que se destacan en azul como parte de la lista *expand*. Como resultado de simplificar el proceso, se evalúan todos los movimientos hacia vecinos posibles con el menor costo, no limitándose a uno, como se explica en la teoría. Se espera que esto resulte en tiempos de procesamiento menores en comparación con otros algoritmos.

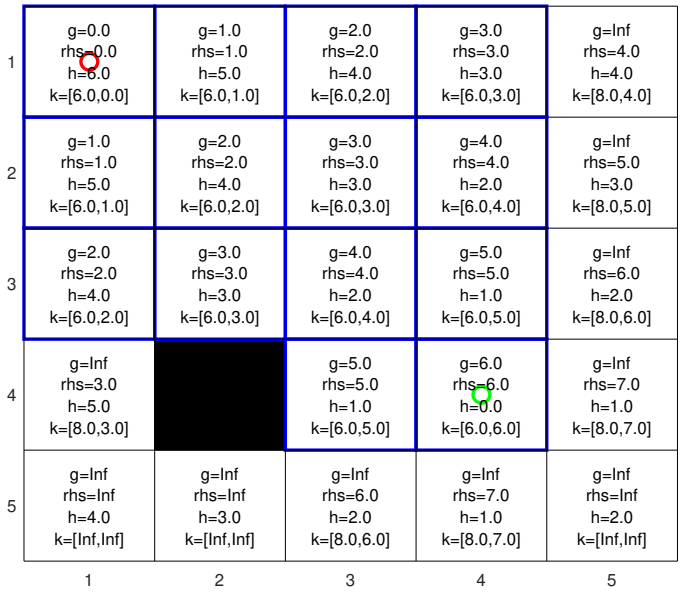


Figura 4.7: Resultado del algoritmo dstar_lite implementado

4.1.2.2. Determinación de Ruta

Como se ha presentado en la función principal `dstar_lite` en la Figura 4.6, existen varias funciones secundarias que son necesarias para el funcionamiento del algoritmo. En la Figura 4.8 se presenta la función que, una vez definidos los valores de rhs y g para nuestros nodos de interés (también almacenados a manera de coordenadas en la matriz *expand*), determina la ruta desde el nodo inicial hasta el objetivo. El primer nodo de dicha ruta corresponde al nodo inicial, y se inicializa el nodo actual como el inicial con un costo de '0'. Mientras el nodo actual no sea el nodo objetivo, se extraen las coordenadas de los vecinos del nodo actual y se validan dichos vecinos para analizar únicamente los que tengan un nodo ya procesado con un valor de g distinto de infinito. Si la lista de vecinos válidos está vacía, se asigna el *path* como vacío

con un costo infinito y se retorna la bandera como falsa. Si la lista no está vacía, se encuentra el vecino válido con menor valor de g y se asigna la posición a la variable 'idx' para almacenar el vecino con menor valor de g como siguiente nodo, agregarlo al *path* incrementando su costo y actualizando el vector actual como este nodo vecino de menor valor g. Una vez se alcanza el nodo objetivo, se cambia la bandera a '*true*' y se finaliza la función.

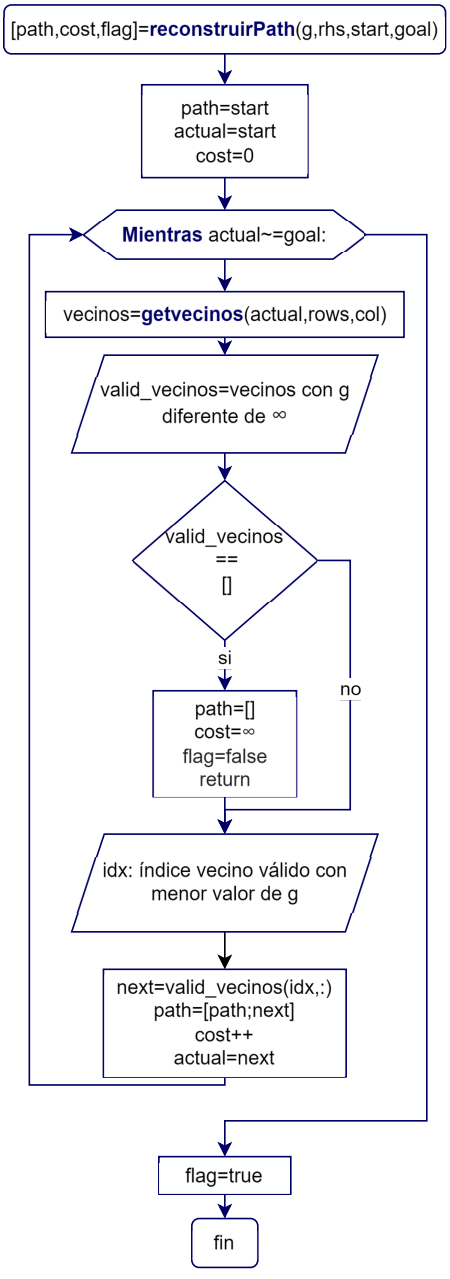


Figura 4.8: Diagrama de flujo de la función para determinar el path desde el nodo destino hacia el objetivo

4.1.2.3. Actualización de la lista U de un nodo específico

La siguiente función **secundaria** se presenta en la Figura 4.9. Consiste en buscar la posición del nodo ingresado como parámetro dentro de la lista U y guardar dicha posición en la variable 'idx'. Si este 'idx' no está vacío, se elimina de la lista U dicha posición. Si el valor de g difiere de rhs en el nodo analizado, se calcula el valor de la clave en dicho nodo y se actualiza la lista U añadiendo el nodo junto con la clave correspondiente.

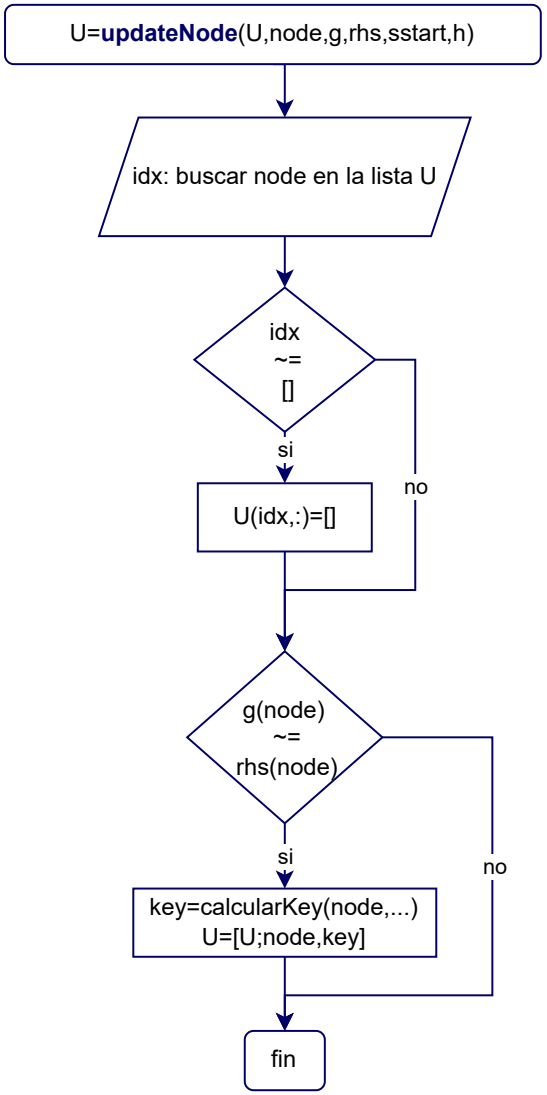


Figura 4.9: Diagrama de flujo de la función para actualizar el nodo en la lista

4.1.2.4. Determinación de vecinos de un nodo

En la Figura 4.10 se presenta una función complemento que es muy utilizada tanto en la función principal como en las secundarias, y tiene un papel fundamental en la aplicación de la distancia Manhattan. La función '`getvecinos`' recibe el nodo del que se extraerán los vecinos, las filas y las columnas. Dentro de la función, se definen las coordenadas del nodo como 'i' y 'j'.

Para identificar y agregar los vecinos adyacentes de un nodo específico, si 'i' es mayor que 1, se agrega el vecino de arriba (i-1, j); si 'i' es menor que el número de filas '*rows*', se agrega el vecino de abajo $(i+1, j)$; si 'j' es mayor que 1, se agrega el vecino de la izquierda $(i, j-1)$; y si 'j' es menor que el número de columnas '*cols*', se agrega el vecino de la derecha $(i, j+1)$. Este procedimiento asegura que solo se consideren vecinos válidos dentro de los límites de la matriz.

Esta restricción de "*solo vecinos lineales*" es clave para emular la distancia Manhattan, ya que, como se presentó en funciones anteriores, los costos asignados hacia estas direcciones son únicamente de 1. La única forma de llegar a las diagonales es mediante los nodos superior y lateral, consiguiendo así un aumento de dos en dos en las diagonales y solo de uno en las lineales.

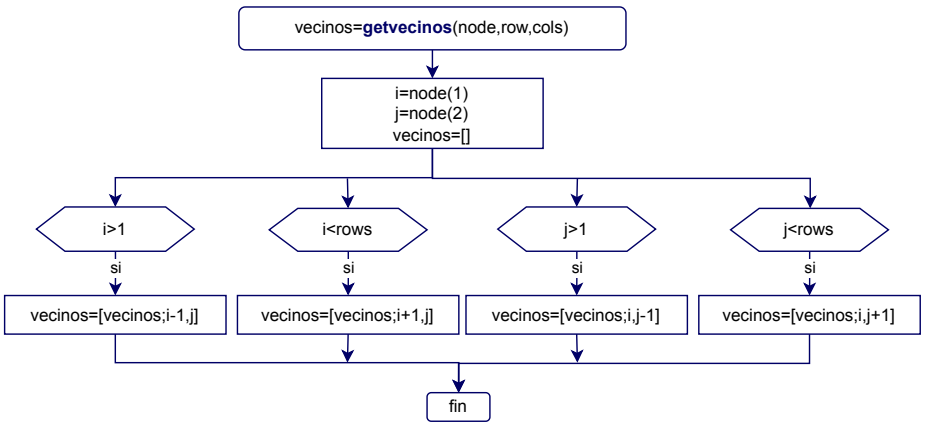


Figura 4.10: Diagrama de flujo de la función para obtener los vecinos

4.1.2.5. Cálculo de las heurísticas

La función secundaria, presentada en la Figura 4.11, es fundamental en la inicialización del algoritmo, ya que calcula la distancia desde el nodo inicial hacia cualquier nodo del mapa. Para ello, primero se extraen las filas y columnas del mapa, se inicializa la matriz h en infinito y se crea una lista denominada 'cola' con el nodo ingresado en la función en primera posición. Mientras esta cola no esté vacía, se realizan los siguientes procesos:

- Extraer el nodo y costo de la cola.
- Limpiar la primera posición de la cola.
- Si el nodo está dentro de los límites del mapa y el valor de h en dicho nodo es mayor al costo:
    - Se asigna como h en ese nodo el valor del costo.
    - Se buscan los vecinos disponibles, y para cada vecino se actualizan los valores de la cola con las coordenadas del vecino seguidas del costo actual aumentado en uno, siempre que dicho nodo no tenga obstáculos y el valor de la heurística sea mayor al del costo más uno.

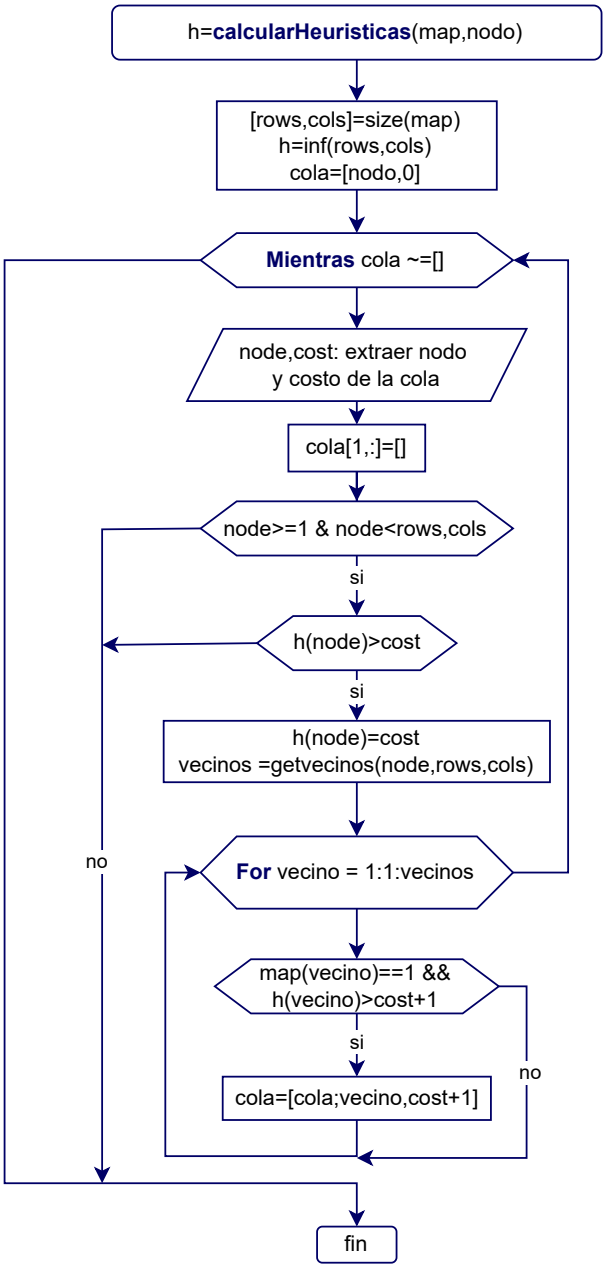


Figura 4.11: Diagrama de flujo de la función para calcular Heurística

La validación de este punto es bastante sencilla, simplemente generamos un mapa sin obstáculos y visualizamos los valores de la heurística h para cada nodo. En la Figura 4.12 se presenta de verde el nodo de inicio y en rojo el nodo de destino. Se puede comprobar que los movimientos lineales se incrementan de uno en uno y los movimientos diagonales aumentan con un paso de dos, como se planteó en un inicio para evitar giros diagonales indebidos como los presentados en la Figura 4.1.

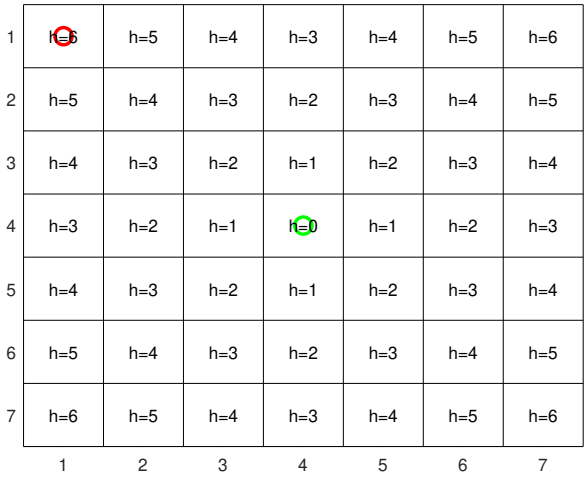


Figura 4.12: Ejemplo de cálculo de heurísticas

4.1.2.6. Determinación del vector k

La función secundaria más sencilla, pero no por ello menos importante, se presenta en la Figura 4.13 y cumple la función de calcular el vector k en cada nodo. Para ello, se necesitan calcular los dos componentes del vector k, denotados en las Ecuaciones 4.1 y 4.2.

$$k1 = min(g(node), rhs(node)) + h(node) \tag{4.1}$$

$$k2 = min(g(node), rhs(node)) \tag{4.2}$$

El componente $k1$ representa el costo estimado más bajo para llegar al nodo objetivo, pasando por el nodo actual, teniendo en cuenta tanto el costo acumulado actual $g(node)$ como el costo acumulado más bajo conocido $rhs(node)$, y añadiendo la heurística $h(node)$. En cuanto al componente $k2$ representa el costo acumulado más bajo conocido para llegar al nodo actual, considerando tanto el costo actual $g(node)$ como el costo más bajo conocido $rhs(node)$. Estas fórmulas se combinan para formar la llave $k = [k1, k2]$, que se utiliza para ordenar y priorizar los nodos en la cola de prioridad, asegurando que el nodo con el costo estimado más bajo sea procesado primero. Esto es fundamental para determinar el *path*.

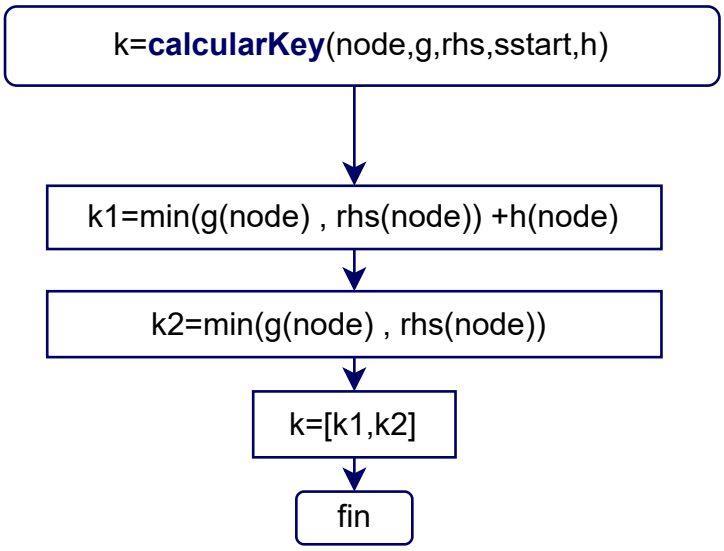


Figura 4.13: Diagrama de flujo de la función para calcular el vector k

4.2. Q-Learning

Se decidió utilizar *Q-learning* como uno de los algoritmos para el diseño de rutas en ambientes dinámicos por varias razones. Primero, *Q-learning* es un algoritmo de aprendizaje por refuerzo conocido por su simplicidad y eficacia en la resolución de problemas de toma de decisiones secuenciales. Su fácil implementación permite su adaptación a una amplia variedad de aplicaciones, incluyendo el diseño de rutas.

Además, *Q-learning* es particularmente adecuado para manejar ambientes dinámicos, permitiendo que el agente se adapte a cambios en el entorno en tiempo real. El algoritmo también equilibra eficientemente la exploración de nuevas rutas y la explotación de rutas conocidas, optimizando el recorrido y evitando obstáculos.

Q-learning maneja adecuadamente la recompensa diferida, un concepto donde las consecuencias de una acción pueden no ser inmediatamente evidentes, pero impactan en el éxito de rutas (*path planning*), donde las decisiones deben considerar efectos a largo plazo.

4.2.1. Lógica de Funcionamiento

El algoritmo de *Q-learning* es un método de RL que permite a un **agente** aprender a tomar decisiones óptimas en un entorno, basándose en la interacción que tiene con

este. La idea central es aprender una política que maximice la recompensa acumulada a lo largo del tiempo. Los componentes clave son:

- **Estados**: Las posibles situaciones en las que se puede encontrar el agente.
- **Acciones**: Las posibles acciones que el agente puede tomar desde cada estado, por ejemplo, movimientos como arriba, abajo, izquierda, derecha, o también diagonales.
- **Recompensas**: La retroalimentación que recibe el agente después de tomar una acción en un estado. Pueden ser negativas o positivas.
- **Tabla Q**: La tabla que almacena los valores Q para cada par estado-acción. Estos valores representan la utilidad esperada de tomar una acción en un estado dado.

Para la construcción del algoritmo se siguieron estos pasos:

1. **Inicialización**: Se inicializa la tabla Q y también los parámetros a usar, cómo:
   - Tasa de aprendizaje (α), que determina cuánto de la nueva información sobrescribe la información anterior.
   - Factor de descuento (γ) que determina la importancia de las recompensas futuras.
   - Tasa de exploración (ε), que controla el equilibrio entre la exploración y explotación.
2. **Interacción con el Entorno**: Se elige una cantidad de episodios o repeticiones, que utiliza la tabla Q para "entrenarse".
   - La idea principal es que el agente llegue desde el punto de inicio al punto objetivo. Para esto se debe seleccionar una acción mediante la estrategia "epsilon-codicioso" (conocido en inglés como *epsilon-greedy*), que permite seleccionar una acción aleatoria (exploración) o la mejor acción conocida por la tabla Q (explotación).
   - Ejecutar la acción y otorgar una recompensa acorde al movimiento realizado. Aquí se puede castigar o premiar al agente según el movimiento que

haya elegido.

- Actualizar la tabla Q de estado-acción con la Ecuación 2.1.

3. **Post-entrenamiento**: Después de entrenar al agente por un número de episodios, la tabla Q contiene los valores de utilidad para cada par 'estado-acción'. Esto sirve de base para guiar al agente en la toma de decisiones óptimas en el futuro, eligiendo siempre la acción con el valor Q más alto en cada estado.

4.2.2. Pruebas y selección de modelo

Durante el desarrollo del algoritmo, se realizaron diversas pruebas para identificar el modelo que mejor se adapta a nuestro entorno. Estas pruebas se enfocaron en determinar los valores óptimos de los siguientes parámetros:

- Tasa de aprendizaje α
- Factor de descuento γ
- Número de episodios
- Tipo de Recompensa

Para encontrar los valores óptimos, se desarrolló un programa en MATLAB que prueba diferentes combinaciones de estos parámetros. Este proceso puede considerarse una evaluación tipo Montecarlo, donde múltiples simulaciones permiten estimar el rendimiento del algoritmo bajo diferentes configuraciones. Como métrica de evaluación, se utilizó el costo obtenido en cada prueba para comparar y determinar qué valores proporcionan los mejores resultados.

Los modelos de implementación considerados fueron:

1. Implementación básica.
2. Implementación dinámica.

4.2.2.1. Implementación Básica

La implementación simple y rápida se refiere a un modelo básico donde la mayoría de los parámetros son constantes. Los valores a evaluar y encontrar son α, γ y el número de episodios.

En esta configuración, se estableció un valor fijo para *epsilon* (ε) y recompensas fijas. Por ejemplo, cuando el agente encuentra un espacio libre dentro de los límites, se otorga una recompensa de -1. Si el movimiento no es permitido, la recompensa (castigo) es de -10. Esta condición se ilustra en el diagrama de la Figura 4.14.

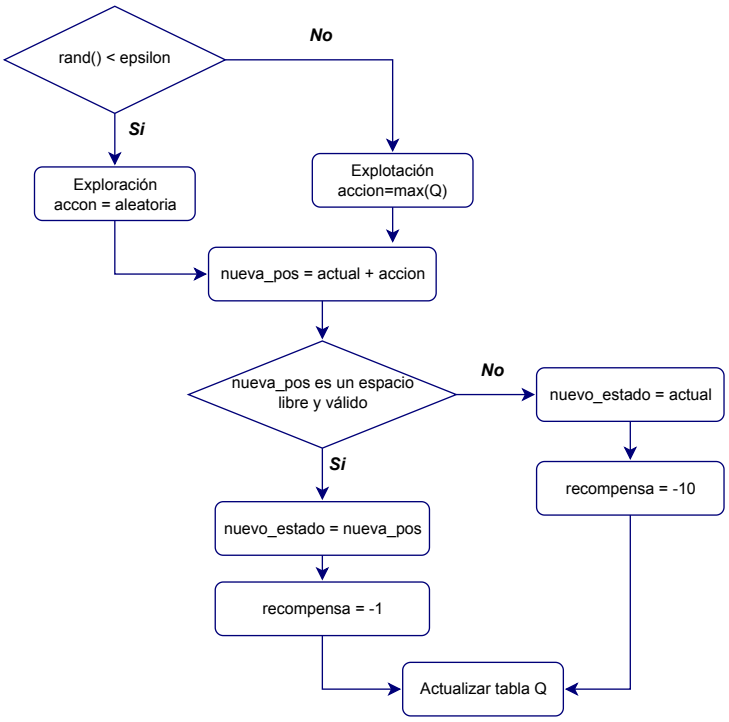
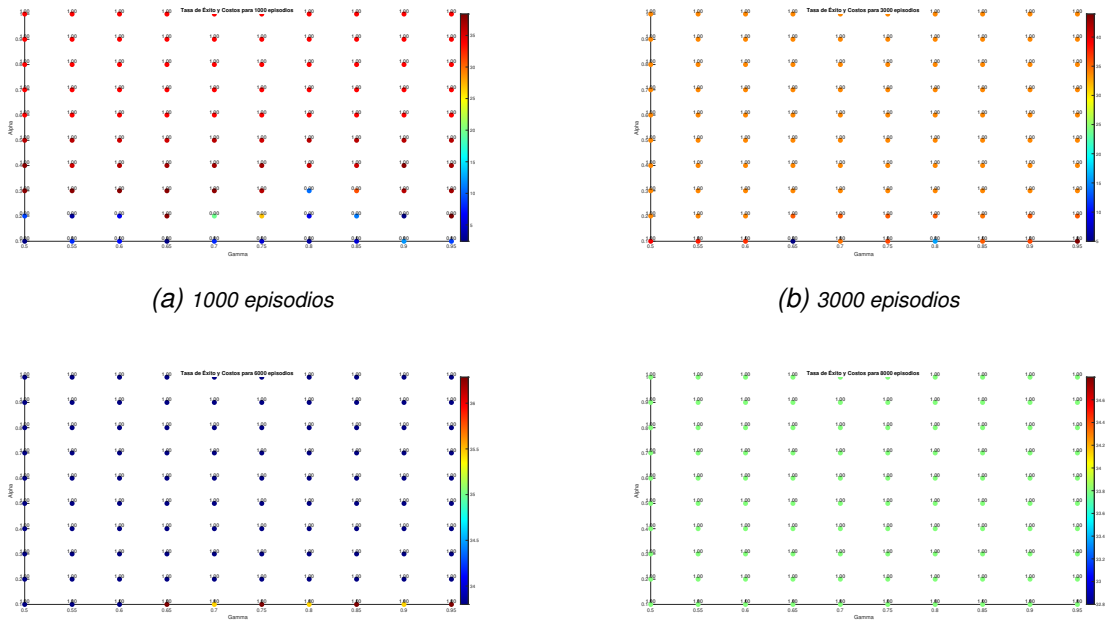


Figura 4.14: Diagrama de flujo que muestra la condición de epsilon y las recompensas

Los resultados de las pruebas con diferentes episodios se presentan en la Figura 4.15. Estas gráficas son mapas de calor, donde el eje de las abscisas representa los valores de γ y el de las ordenadas los valores de α. Los colores de cada nodo indican el costo del camino, con la paleta de colores situada a la derecha de cada gráfica. Los valores numéricos de los nodos varían entre '1' y '0', representando si el camino ha encontrado la meta o no, respectivamente.

Los resultados de la Figura 4.15, indican que, a medida que aumenta el número de episodios, los valores de α y γ se vuelven menos relevantes para el algoritmo, en



(a) 1000 episodios *(b) 3000 episodios*

(c) 6000 episodios *(d) 8000 episodios*

Figura 4.15: Comparativa del costo de la ruta en función de los episodios - Relación entre [α] (Eje Y) y [γ] (Eje X)

el sentido de que ya no afectan al costo de la ruta encontrada. Esto se observa en la Subfigura 4.15d, donde todos los nodos se pintan de verde, indicando el mismo costo y que todos han encontrado el camino hacia el objetivo. Sin embargo, debido al incremento en el tiempo de cómputo con más episodios, se buscaron rangos de valores permisibles para α y γ que proporcionen buenos resultados con un menor número de episodios.

En la Tabla 4.2, se presentan los rangos de valores permisibles encontrados y los valores específicos seleccionados para las simulaciones con una implementación básica. Se escogieron los valores de $\alpha = 0.2$ y $\gamma = 0.8$, aunque existe un rango de valores en los cuales pueden funcionar. Este rango debe estar alineado con el número de episodios seleccionados, ya que con una menor cantidad de episodios, el algoritmo tiende a encontrar soluciones subóptimas o a no encontrar ninguna solución. Las pruebas realizadas mostraron que, con más de 6000 episodios, se **garantiza encontrar una ruta que llegue al objetivo y sea óptima** en términos de distancia y costo de cómputo.

Tabla 4.2: Parámetros para el algoritmo de Q-learning, Tasa de aprendizaje α y Factor de descuento γ seleccionados para pruebas en una implementación sencilla.

	Rango de Valores	Valores Elegidos	Número de Episodios
α	0.2 - 1.0	0.2	Mayor a 6000
γ	0.8-0.95	0.8	

4.2.2.2. Implementación Dinámica

Una implementación dinámica implica variar los parámetros de evaluación para agilizar el cálculo y mejorar los procedimientos. En este caso, el objetivo es que el algoritmo alcance los mejores valores y entregue la mejor ruta en función de su distancia.

Para esta configuración, se mantiene el uso del método epsilon-codicioso (conocidd en inglés como *epsilon-greedy*). La diferencia radica en que ahora se varían los parámetros de ε, α y la recompensa que maneja el agente. Las consideraciones del algoritmo fueron las siguientes:

- Se inicia con un valor alto de (ε) para garantizar una mayor exploración al inicio del cálculo, estableciendo también límites para controlar el rango de exploración.
- Se utiliza un umbral de rendimiento para ajustar y evaluar los valores de (ε) y α.
- Las recompensas varían según los movimientos del agente y su interacción con el entorno:
  - **Movimiento válido**: Recompensa básica de -1.
  - **Acercarse al objetivo**: Recompensa adicional de +1.
  - **Alejarse del objetivo**: Recompensa de -2.
  - **Movimiento inválido**: Recompensa de -10.
  - **Alcanzar el objetivo**: Gran recompensa de +100.

La Figura 4.16 muestra el diagrama de flujo de esta configuración, ilustrando las condiciones mencionadas anteriormente para la implementación dinámica.

Una vez realizadas las pruebas con la implementación dinámica y variando los valores de α y γ, se obtuvieron los resultados presentados en la Figura 4.17. Estos resultados

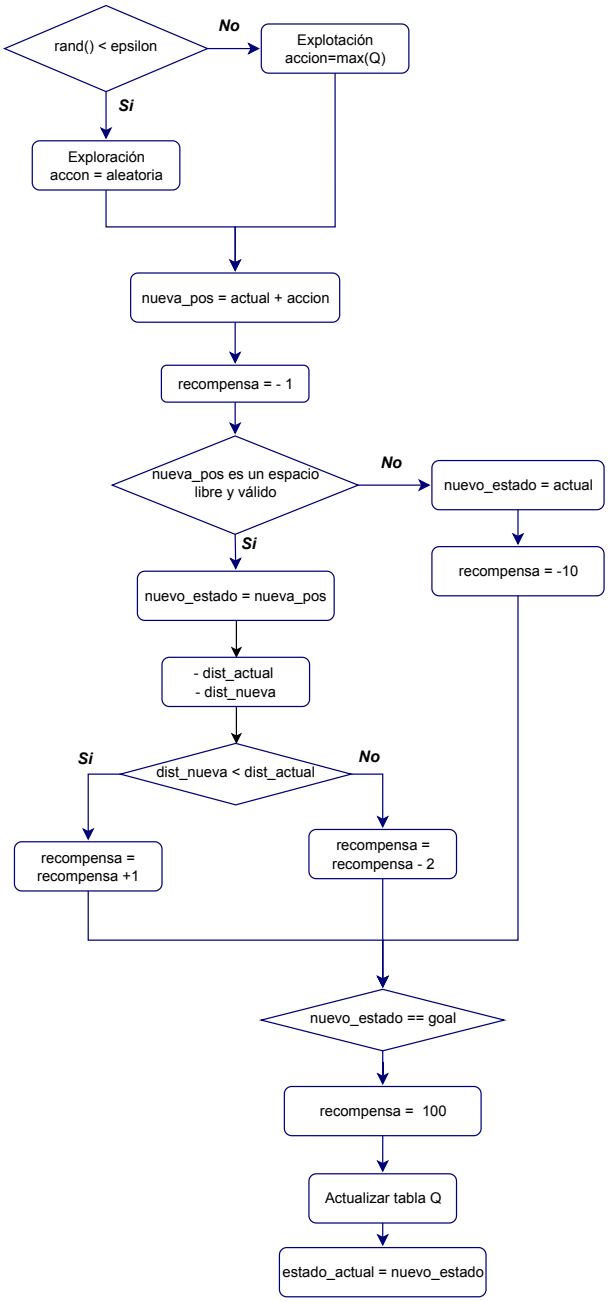
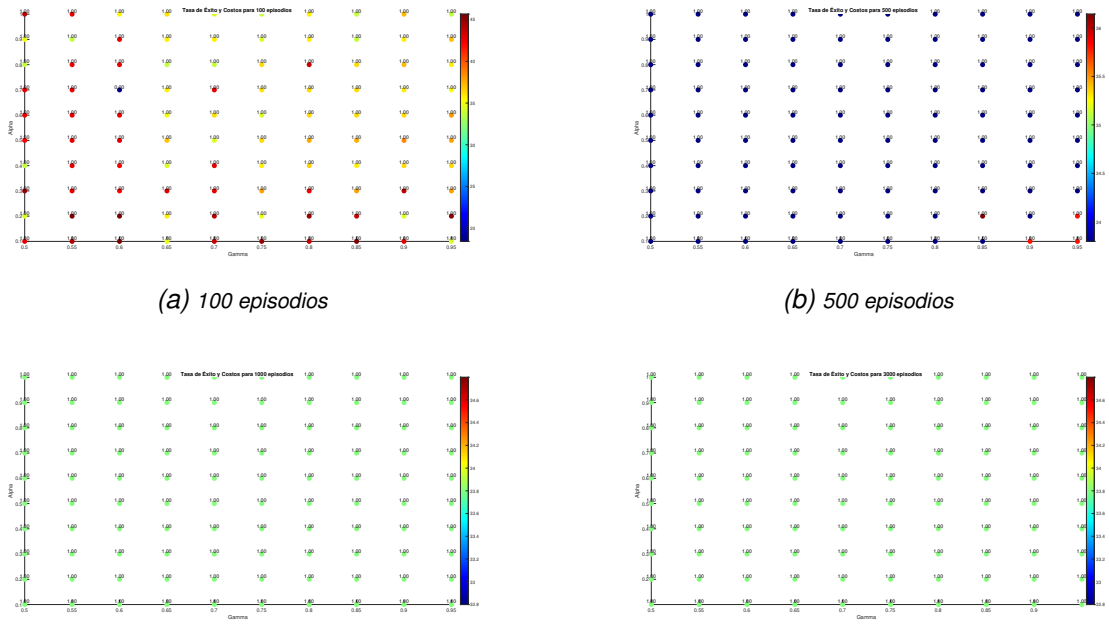


Figura 4.16: Diagrama de flujo que indica la variación de las recompensas según la interacción con el ambiente

indican que, con una menor cantidad de episodios, se lograron obtener mejores resultados en comparación con la implementación básica. Sin embargo, es importante mencionar que la cantidad de episodios debe ser mayor a 1000 para garantizar que se alcance el objetivo de manera consistente.

Además, se pueden considerar los mismos valores de α y γ que se obtuvieron en la implementación básica. Los datos para esta simulación se muestran en la Tabla 4.3.



(a) 100 episodios *(b) 500 episodios*

(c) 1000 episodios *(d) 3000 episodios*

Figura 4.17: Comparativa del costo de la ruta en función de los episodios - Relación entre [α] (Eje Y) y [γ] (Eje X)

Tabla 4.3: Parámetros para el algoritmo de Q-learning: Tasa de aprendizaje α, Factor de descuento γ, epsilon (ε) y número de episodios seleccionados para la implementación dinámica

	Rango de Valores	**Valores Elegidos**	**Número de Episodios**
α	0.2 - 1.0	0.2	Mayor a 1000
γ	0.8 - 0.95	0.8	
(ε)	Inicia en 1.0 y varía en función de la recompensa		

4.2.3. Modelo implementado

Comparando los resultados presentados en las Tablas 4.2 y 4.3, se observa que utilizar un método básico conlleva una gran cantidad de episodios para alcanzar el resultado esperado, a diferencia de la implementación dinámica que utiliza menos episodios.

Sin embargo, esta última toma más tiempo en la ejecución debido a las decisiones implementadas y al dinamismo de los parámetros como *epsilon* y las recompensas.

Con esta idea en mente, se plantea una combinación de ambas implementaciones para optimizar el rendimiento del algoritmo de *Q-learning*. La configuración es la siguiente:

- α: adaptable a lo largo de los episodios, iniciando con un valor alto para aprender de las experiencias iniciales y luego estabilizándose para aprovechar el conocimiento acumulado.
- γ: valor fijo de 0.8, escogido de la Tabla 4.2.
- ε: tasa de exploración decreciente con el tiempo para equilibrar la exploración y la explotación.
- Recompensas: fijas como en la implementación básica.
- Episodios: cantidad reducida, entre 2000 y 4000, según lo indicado en la Tabla 4.3.

El diagrama de flujo del algoritmo combinado de *Q-learning* se presenta en la Figura 4.18, que muestra la lógica utilizada para inicializar los parámetros y trabajar según la interacción del agente con el mapa.

Como punto de validación, se comparó este algoritmo combinado con las implementaciones descritas en las Subsecciones 4.2.2.1 y 4.2.2.2, en términos de tiempo de ejecución y uso de la CPU. Los resultados obtenidos, presentados en la Figura 4.19, se encuentran en segundos. Se realizaron un total de cincuenta pruebas por cada algoritmo, obteniendo un valor promedio y un intervalo de confianza del 95%. En ambas métricas (uso de CPU 4.19b y tiempo de ejecución 4.19a), el algoritmo `entrenamiento_q_opt` demostró ser el más eficiente, seguido de la implementación simple representada por el algoritmo `entrenamiento_q`. El algoritmo de implementación dinámica, `entrenamiento_q_adaptativo`, resultó ser el menos eficiente, consumiendo significativamente más tiempo y recursos de CPU. Esto sugiere que la mejor opción entre los tres algoritmos evaluados es `entrenamiento_q_opt`, es decir, el algoritmo combinado de la Figura 4.16.

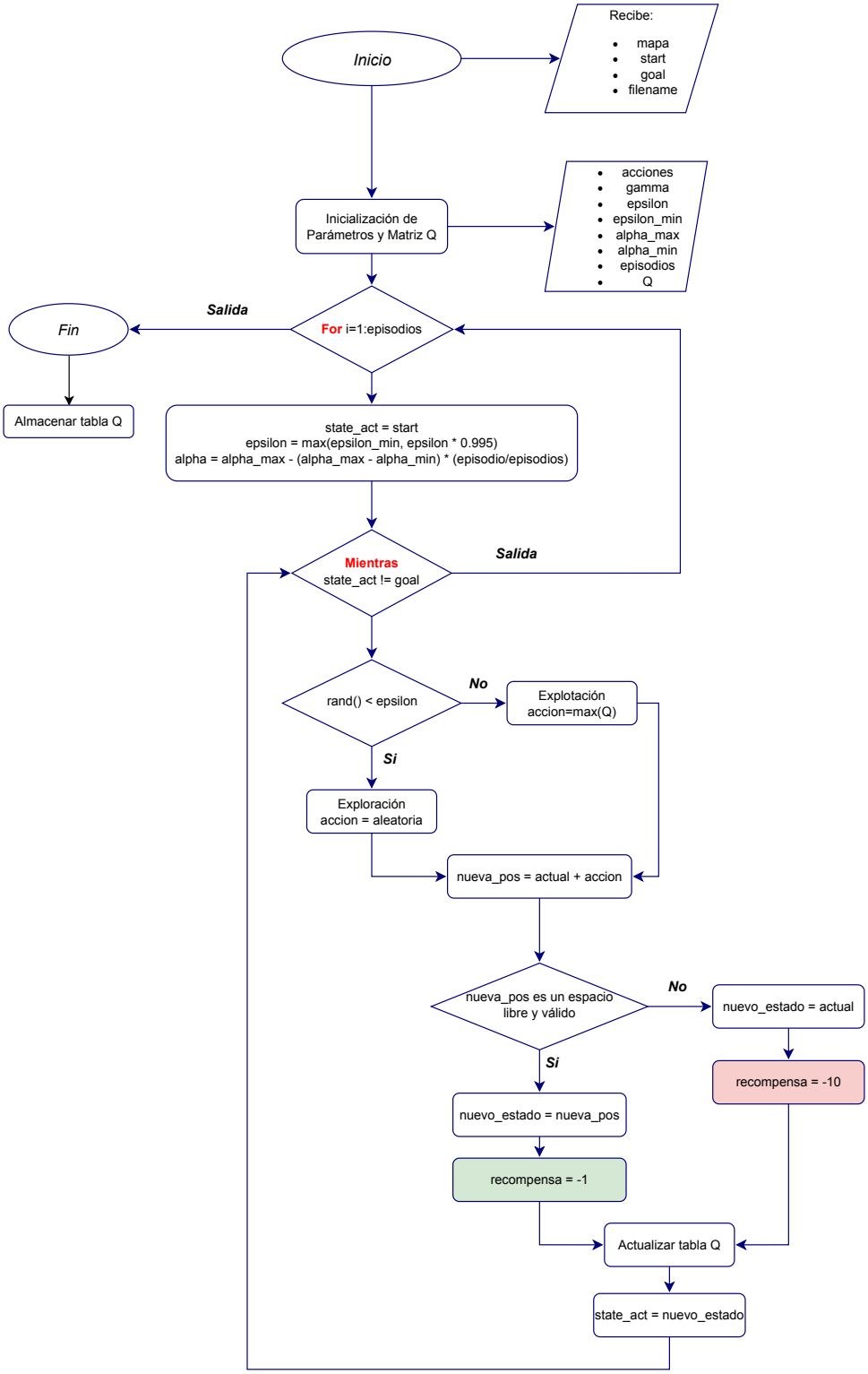
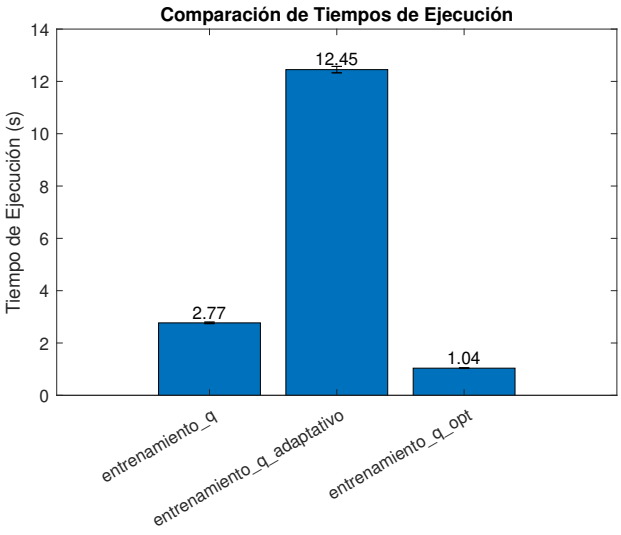
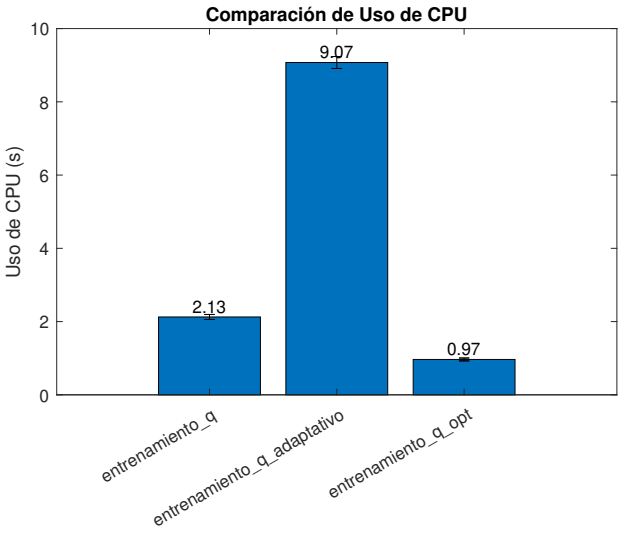


Figura 4.18: Diagrama de flujo del algoritmo de Q-learning implementado



(a) Comparación de Tiempos de Ejecución



(b) Comparación de Uso de CPU

Figura 4.19: Comparación de los algoritmos Q-learning en términos de tiempo de ejecución y uso de CPU. De izquierda a derecha: implementación básica, implementación dinámica e implementación combinada

4.2.3.1. Generación de la ruta

Una vez entrenada la tabla Q utilizando el modelo combinado mostrado en la Figura 4.18, se procede a encontrar la ruta óptima hacia el objetivo y calcular el costo asociado a dicha ruta. Este proceso se realiza mediante los siguientes pasos:

- **Inicialización del Estado Inicial**: Se comienza desde el estado definido en el entorno, que representa la posición de origen del agente.
- **Selección de Acciones Óptimas**: A partir del estado actual, se selecciona la acción con el mayor valor Q, que representa la mejor decisión en ese estado. Esta acción guía al agente al siguiente estado.
- **Actualización del Estado:** El agente se mueve al siguiente estado y se actualiza el estado actual con este nuevo estado.
- **Cálculo del Costo de la Ruta:** Se acumulan los costos asociados a cada movimiento realizado. Este costo se actualiza en cada paso.
- **Condición de Finalización**: El proceso continúa iterativamente hasta que el agente alcanza el estado objetivo. La secuencia completa presenta la ruta óptima encontrada y el costo total refleja la eficiencia de dicha ruta.

El diagrama de flujo en la Figura 4.20 proporciona una representación visual de este proceso, destacando cada uno de los pasos mencionados.

4.2.4. Validación del Algoritmo

Validar el rendimiento de un algoritmo es necesario para conocer su eficacia. Algunas métricas utilizadas en esta validación son la recompensa acumulada, "*pérdida de valor estimado*" y tiempo de ejecución.

El algoritmo utilizado se muestra en la Figura 4.18. Estas pruebas indican que el algoritmo no solo aprende, sino que lo hace de manera efectiva, buscando siempre obtener la mejor solución en función de lo que el agente aprende.

Los parámetros seleccionados para evaluar el algoritmo se presentan en la Tabla 4.4. Estos valores, que han demostrado ser los más eficaces para el funcionamiento del

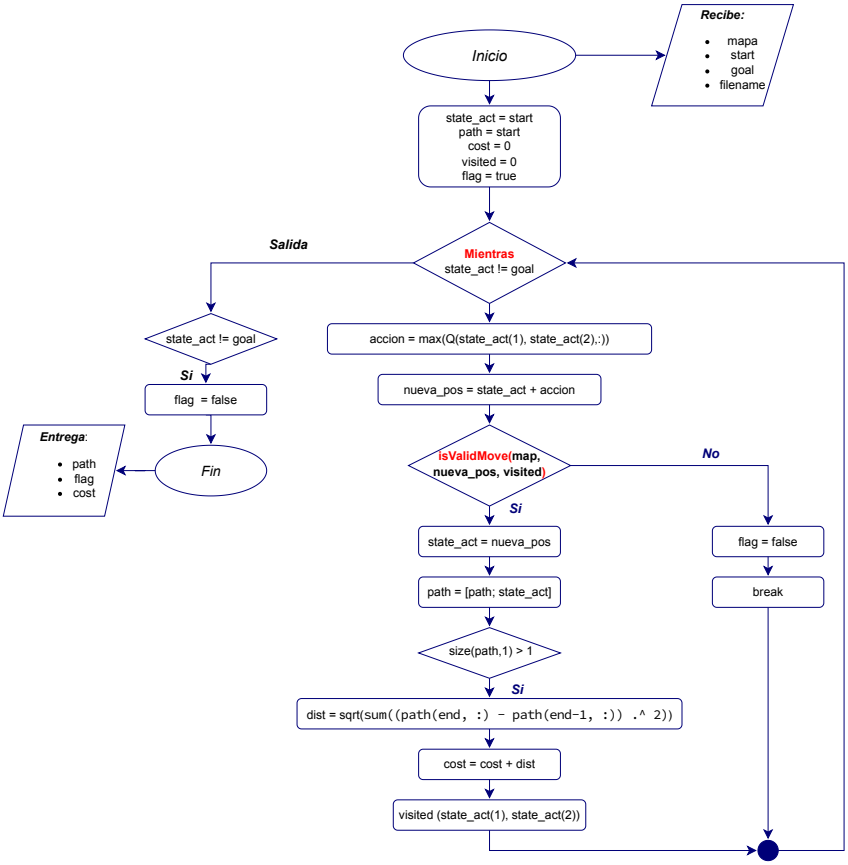


Figura 4.20: Diagrama de flujo del programa que encuentra la ruta a partir de la tabla Q

algoritmo, se describieron en detalle en la Subsección 4.2.3.

Tabla 4.4: Parámetros utilizados para la validación del algoritmo

Parámetros Iniciales	
α_{max}	0.5
α_{min}	0.1
γ	0.8
ε	0.9
Episodios	3000

Pruebas Realizadas
Recompensa Acumulada
Pérdida de Valor Estimado
Tiempo de ejecución (Pasos - Tiempo)

4.2.4.1. Recompensas Acumuladas

Esta métrica evalúa la recompensa total acumulada por el agente a lo largo de cada episodio de entrenamiento. En la Figura 4.21, se puede observar que el agente parte de una recompensa muy negativa, debido a que el punto de inicio se encuentra a una gran distancia del punto objetivo. Esto resulta en penalizaciones iniciales significativas, ya que cada movimiento hacia el objetivo conlleva una penalización. A medida que el agente aprende la mejor política para llegar al objetivo, la recompen-

sa acumulada comienza a aumentar y tiende a estabilizarse cerca de la recompensa mínima configurada en el algoritmo, lo que indica que el agente está recibiendo más incentivos que castigos. Este comportamiento sugiere que el agente está mejorando su capacidad de tomar decisiones que maximicen su recompensa total, validando así la eficacia de la política de decisión implementada.

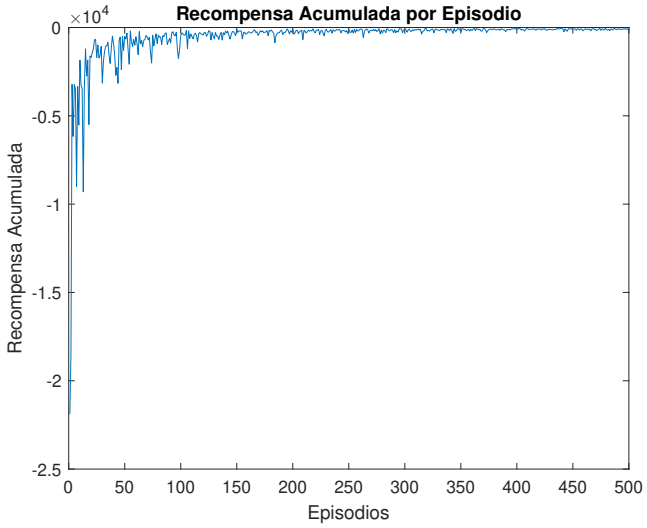


Figura 4.21: Recompensa acumulada por episodio durante el entrenamiento

4.2.4.2. Pérdida de Valor Estimado

La métrica de pérdida de valor estimado (también conocida como *Value Loss*) mide la diferencia entre el valor Q actual y el valor Q predicho en cada actualización del algoritmo. Esta métrica monitorea cómo los valores Q convergen a lo largo del tiempo. En la Figura 4.22 se observa la tendencia de la tabla Q al pasar los episodios. Inicialmente, la pérdida de valor puede ser alta debido a la gran variabilidad y la falta de información precisa en las primeras etapas del aprendizaje. Sin embargo, a medida que el agente experimenta y aprende, los valores Q se estabilizan y tienden a cero. Esta disminución en la pérdida de valor indica una convergencia hacia una política óptima, donde el agente tiene una estimación precisa del valor esperado de las acciones en cada estado. Esto demuestra que el algoritmo está aprendiendo de manera efectiva y alcanzando una estabilidad en su proceso de aprendizaje.

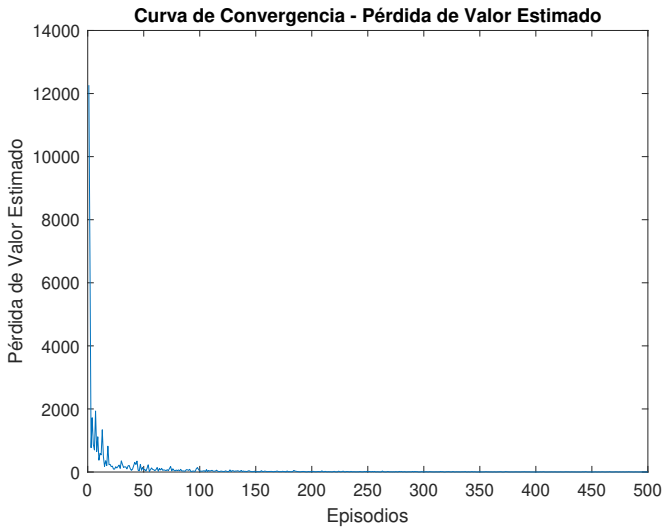
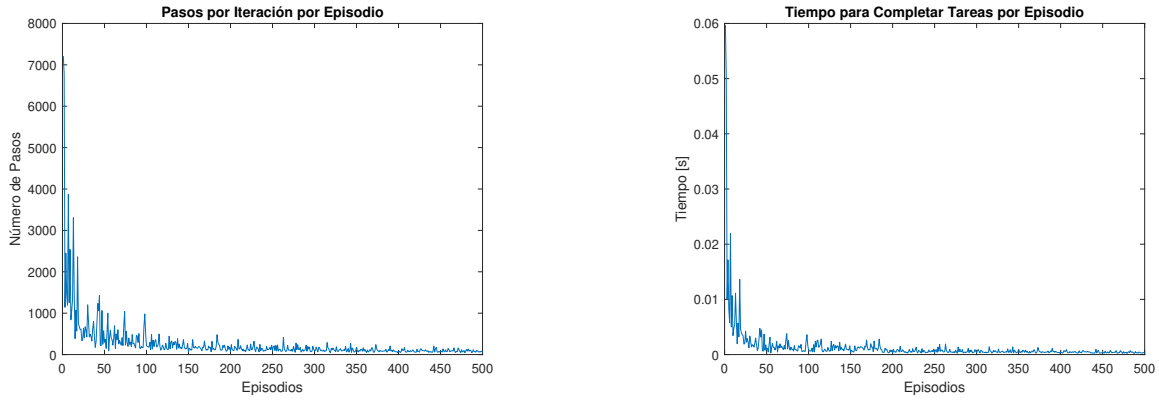


Figura 4.22: Tendencia de la métrica 'Pérdida de Valor Estimado' en función del número de episodios

4.2.4.3. Tiempo de Ejecución

El tiempo de ejecución se mide en términos del número de pasos o iteraciones necesarias para que el agente alcance el objetivo en cada episodio. En la Figura 4.23a se observa que el número de pasos necesarios para encontrar el objetivo tiende a reducirse conforme pasan los episodios. Esto indica que el agente está aprendiendo a encontrar rutas más eficientes hacia el objetivo, reduciendo el número de movimientos necesarios. Además, en la Figura 4.23b se presenta que el tiempo de ejecución de cada iteración también disminuye a lo largo del entrenamiento. Esta reducción en el tiempo de cálculo sugiere que el agente no solo está aprendiendo a tomar decisiones más rápidas, sino también más eficientes, optimizando así su desempeño global. La combinación de estas dos métricas refuerza la evidencia de que el agente está mejorando continuamente en su capacidad de alcanzar el objetivo de manera rápida y eficiente.

Los pasos se cuentan dentro del bucle *while* del algoritmo presentado en la Figura 4.18, que evalúa si el estado actual ha alcanzado el objetivo y continúa el entrenamiento por el número de episodios especificado.



(a) Número de Pasos vs. Episodio *(b) Tiempo vs. Episodio*

Figura 4.23: Validación del algoritmo: comparación del número de pasos y tiempo vs. la cantidad de episodios

Resultados

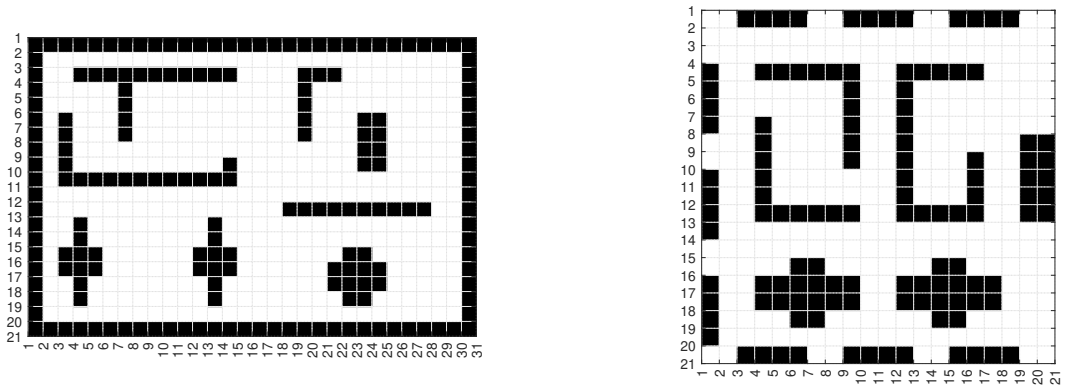
En esta sección se presentan y analizan los resultados obtenidos en las simulaciones realizadas. El entorno utilizado es MATLAB, en el cual se ha generado distintos escenarios, con un enfoque en algoritmos de planificación de rutas en entornos dinámicos. Los algoritmos evaluados incluyen *A**, *Dijkstra*, *D**, *D* Lite*, *RRT* y *Q-learning*.

5.1. Presentación de Escenarios

Los escenarios utilizados son modelos básicos que permiten realizar pruebas iniciales de los algoritmos diseñados, evaluando su funcionamiento en un entorno controlado y sencillo antes de escalar a mapas más complejos y extensos. Los escenarios se muestran en la Figura 5.1, en donde los obstáculos se representan mediante recuadros de color negro, mientras que el espacio libre se muestra en color blanco.

El *Escenario 1*, que se presenta en la Figura 5.1a, es un modelo de 20×30 unidades. El *Escenario 2*, mostrado en la Figura 5.1b, es un modelo de 20×20 unidades. Estos escenarios forman una base para observar cómo los algoritmos se comportan y manejan la planificación de rutas en un entorno de mayor dimensión con obstáculos dispersos.

Las pruebas sobre estos modelos simples proporcionan información sobre el comportamiento de los algoritmos, identificando posibles áreas de mejora y adaptaciones necesarias para su expansión a mapas de mayor complejidad.



(a) Escenario 1 *(b) Escenario 2*

Figura 5.1: Modelos de escenarios básicos utilizados para pruebas de los algoritmos de planificación de rutas

5.2. Comparativa de Interacción de Algoritmos con los Escenarios

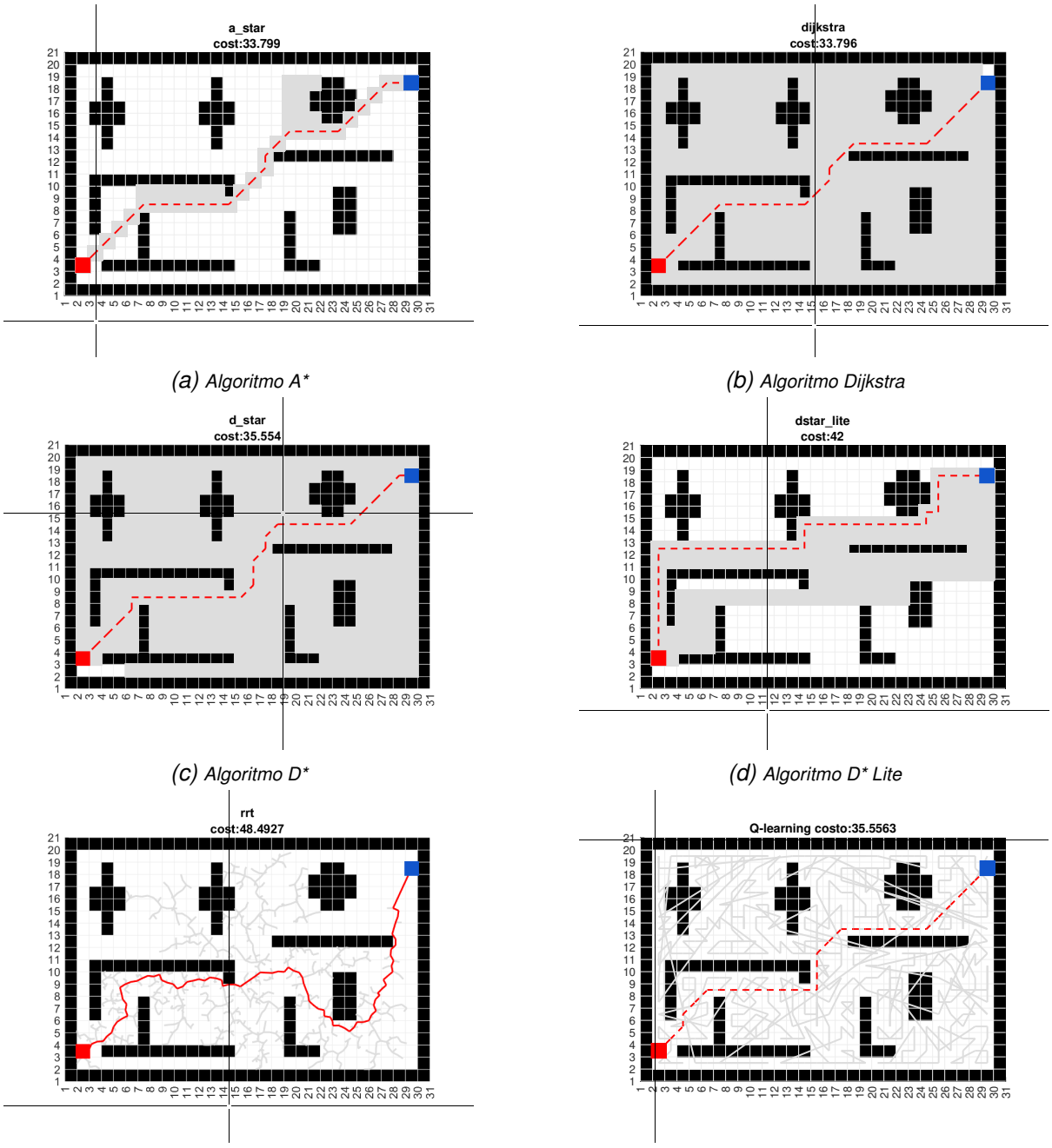
5.2.1. Expansión de Nodos durante la Planificación de Rutas

Esta comparación se enfoca en revisar cómo los algoritmos encuentran la ruta, para esto está el término *expand* en los algoritmos, que almacena los nodos de expansión visitados hasta encontrar el nodo objetivo.

En la Figura 5.2 se presentan los resultados de distintos algoritmos en el escenario 1 de la Figura 5.1a. Es evidente que el algoritmo *A** (Figura 5.2a) procesa menos nodos, pero no es adecuado para entornos dinámicos. El algoritmo *Dijkstra* (Figura 5.2b) procesa todos los nodos del mapa, resultando en altos costos de procesamiento.

El algoritmo *D** (Figura 5.2c) también procesa todos los nodos en su primera iteración, lo cual es ineficiente en comparación con *D* Lite* (Figura 5.2d), que procesa solo los nodos necesarios para alcanzar el objetivo, evitando cálculos innecesarios cuando el objetivo está cerca en un mapa grande.

El algoritmo basado en aprendizaje por refuerzo, *Q-learning* (Figura 5.2f), muestra una expansión diferente, más similar al algoritmo *RRT* (Figura 5.2e), ya que la lista *expand* representa el recorrido del entrenamiento para generar las listas *Q*, mostrando su capacidad de aprendizaje y adaptación en el mapa.



*(a) Algoritmo A**

(b) Algoritmo Dijkstra

*(c) Algoritmo D**

(d) Algoritmo D Lite*

(e) Algoritmo RRT

(f) Algoritmo Q-Learning

Figura 5.2: Comparación de las listas de expansión de diferentes algoritmos

5.2.2. Simulaciones sobre Escenarios

Conociendo la expansión y el funcionamiento de los algoritmos en el escenario propuesto, se compara su comportamiento ante un escenario cambiante. Este escenario varía con el aumento de las iteraciones, y el algoritmo debe adaptarse a estos cambios.

En las simulaciones realizadas sobre los escenarios de las Figuras 5.1a y 5.1b, se observan los siguientes elementos: el punto rojo indica el punto de partida, el punto verde marca la meta y la ruta calculada se representa en color cian. En cada escenario se compara la primera simulación con la simulación número cien para evaluar el comportamiento del algoritmo de planificación de rutas.

En la primera simulación de cada escenario (Ver Figuras 5.3 y 5.5), no se colocaron obstáculos, lo que permite observar la ruta inicial calculada por el algoritmo en un entorno despejado. En contraste, en la última simulación (Ver Figuras 5.4 y 5.6), se añaden obstáculos dentro del mapa, obligando al algoritmo a replantear la ruta original. Esto demuestra su capacidad para interactuar dinámicamente con el mapa y recalcular la ruta a medida que aparecen nuevos obstáculos.

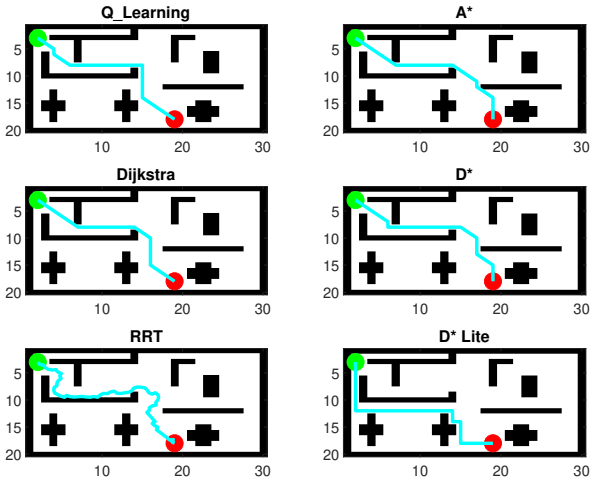


Figura 5.3: Simulación sin obstáculos en el escenario 5.1a

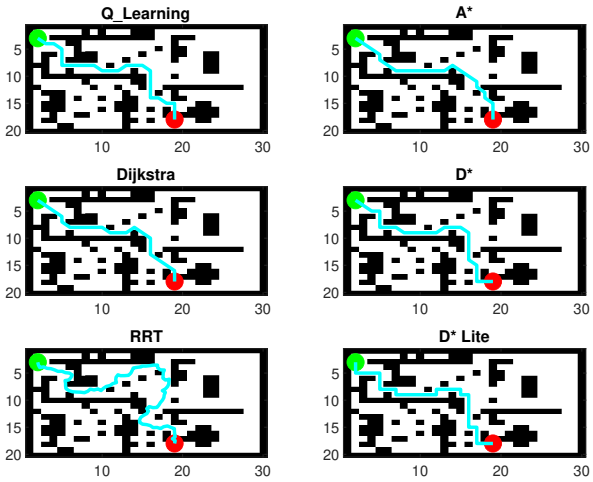


Figura 5.4: Simulación con obstáculos en el escenario 5.1a

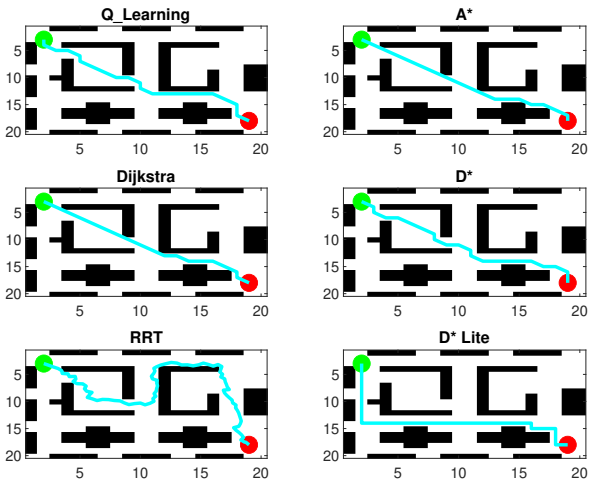


Figura 5.5: Simulación sin obstáculos en el escenario 5.1b

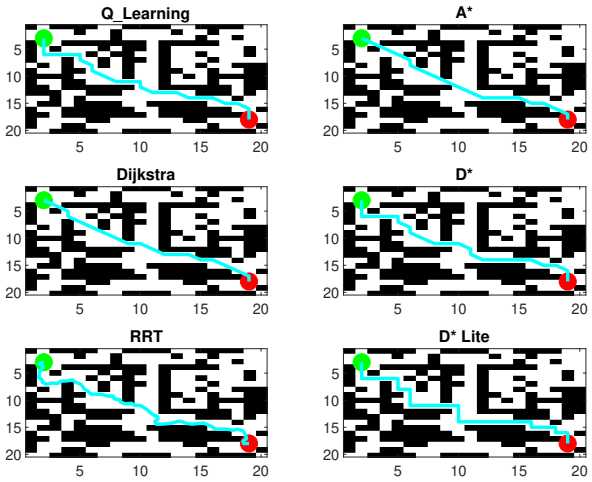


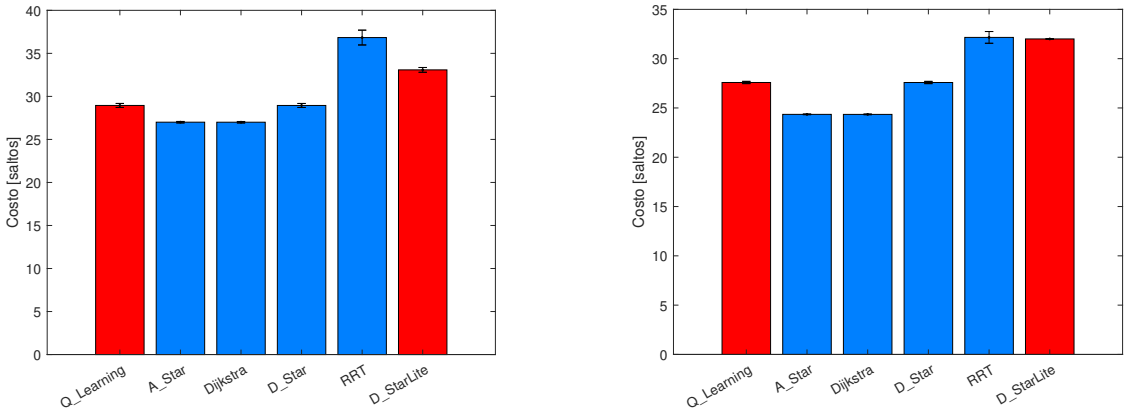
Figura 5.6: Simulación con obstáculos en el escenario 5.1b

Para revisar las métricas obtenidas y comparar los resultados entre algoritmos, se evalúan en función del costo, que es el número de saltos que toma el algoritmo; en función de la longitud, que considera la distancia entre nodos; y en función de los tiempos de ejecución, que es el tiempo que cada algoritmo tarda en planificar la ruta. Los algoritmos *D* Lite* y *Q-learning* se representan en color rojo, mientras que el resto se muestran en color azul, para diferenciar claramente entre los algoritmos desarrollados y los ya implementados.

Se han definido cien simulaciones y se ha calculado un intervalo de confianza del 95 % en los gráficos de barras para asegurar la precisión y fiabilidad de los resultados.

5.2.2.1. Comparación de Costos

Para la prueba de costos, se considera la distancia entre el punto de inicio (*start*) y el punto objetivo (*goal*) que calcula cada algoritmo. La Figura 5.7 muestra que los algoritmos *D* Lite* y *Q-learning* no presentan el menor costo entre todos los algoritmos evaluados, lo que cuestiona su eficiencia.



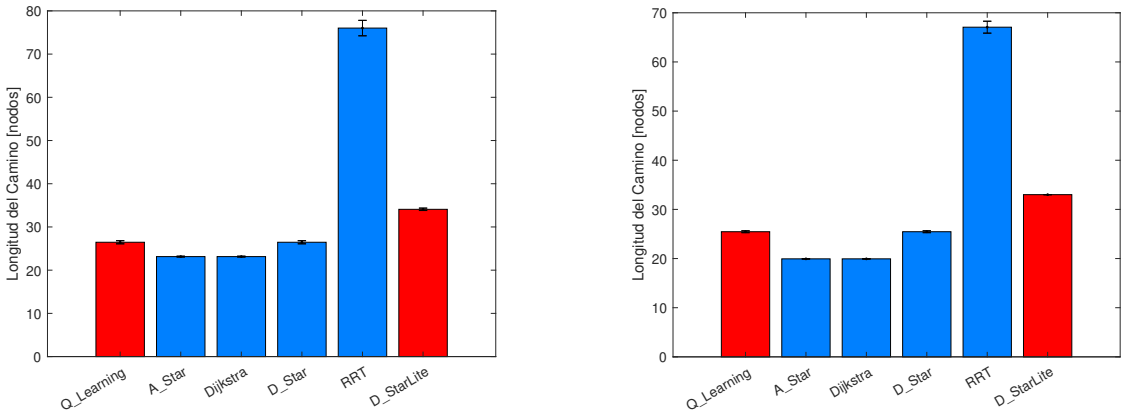
(a) *Prueba Escenario 1* (b) *Prueba Escenario 2*

Figura 5.7: Comparación de costos entre algoritmos en diferentes escenarios

- Aunque los algoritmos *D* Lite* y *Q-learning* tienen un costo medio mayor en comparación con *A**, *Dijkstra* y *D**, generan rutas considerando restricciones específicas, como evitar que dos bloques sean adyacentes diagonalmente (ver Figura 4.1). Esta consideración es crucial para aplicaciones en vehículos autónomos en entornos reales.
- Al revisar los intervalos de confianza de cada algoritmo, se aprecia que *D* Lite* y *Q-learning* son más fiables, ya que sus costos son más consistentes. En contraste, los algoritmos *A** y *Dijkstra* muestran menos variabilidad en sus rutas, indicando una menor sensibilidad a los cambios en el entorno.
- El algoritmo *RRT* obtuvo los peores resultados debido a la aleatoriedad en la generación de sus rutas, por lo tanto, su relevancia en esta evaluación es mínima.

5.2.2.2. Comparación de Longitud de Ruta

La prueba de comparación de la longitud de la ruta está estrechamente relacionada con la prueba de costos, ya que el costo se mide en función de los pasos y la longitud se basa en la distancia de estos puntos hacia el objetivo. La Figura 5.8 muestra esta comparación, donde la tendencia es similar a la de la Figura 5.7. Aunque los demás algoritmos obtienen longitudes de ruta menores, *D* Lite* y *Q-learning* presentan los mejores resultados al considerar las restricciones de movimiento de los vehículos.



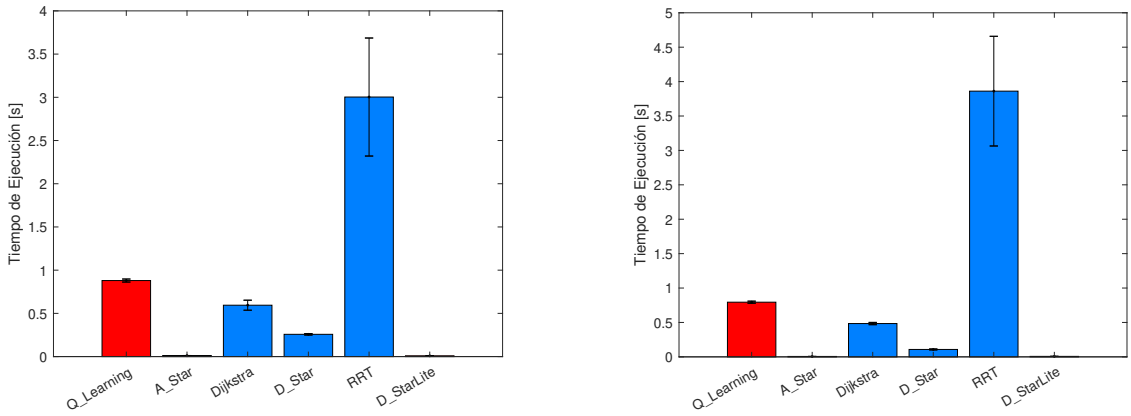
(a) *Prueba Escenario 1* (b) *Prueba Escenario 2*

Figura 5.8: Comparación de longitud de rutas entre algoritmos en diferentes escenarios

5.2.2.3. Comparación de Tiempos de Ejecución

La eficiencia en términos de tiempos de cómputo es determinante para los vehículos autónomos que requieren decisiones en tiempo real. La comparación de los tiempos de ejecución se presenta en la Figura 5.9.

- El algoritmo *D* Lite* destaca sobre el resto, con tiempos de cálculo en milisegundos, comparado con el algoritmo *Q-learning*, que tiene un promedio mayor a un segundo.
- *D* Lite* presenta uno de los mejores intervalos de confianza, indicando alta fiabilidad. *Q-learning* muestra cierta variabilidad en sus tiempos de ejecución, aunque no es preocupante. Esta variabilidad se debe al proceso de entrenamiento de la tabla *Q*, que puede requerir diferentes cantidades de tiempo en cada iteración dependiendo de la complejidad del entorno y el número de episodios de entrenamiento.
- Una vez que la tabla *Q* está bien entrenada, el algoritmo es capaz de generar rutas eficientes y adaptativas. Aunque el tiempo de cálculo inicial puede ser mayor y variable, el rendimiento final de *Q-learning* es robusto y efectivo, justificando el tiempo adicional invertido en su entrenamiento.



(a) Prueba Escenario 1 *(b) Prueba Escenario 2*

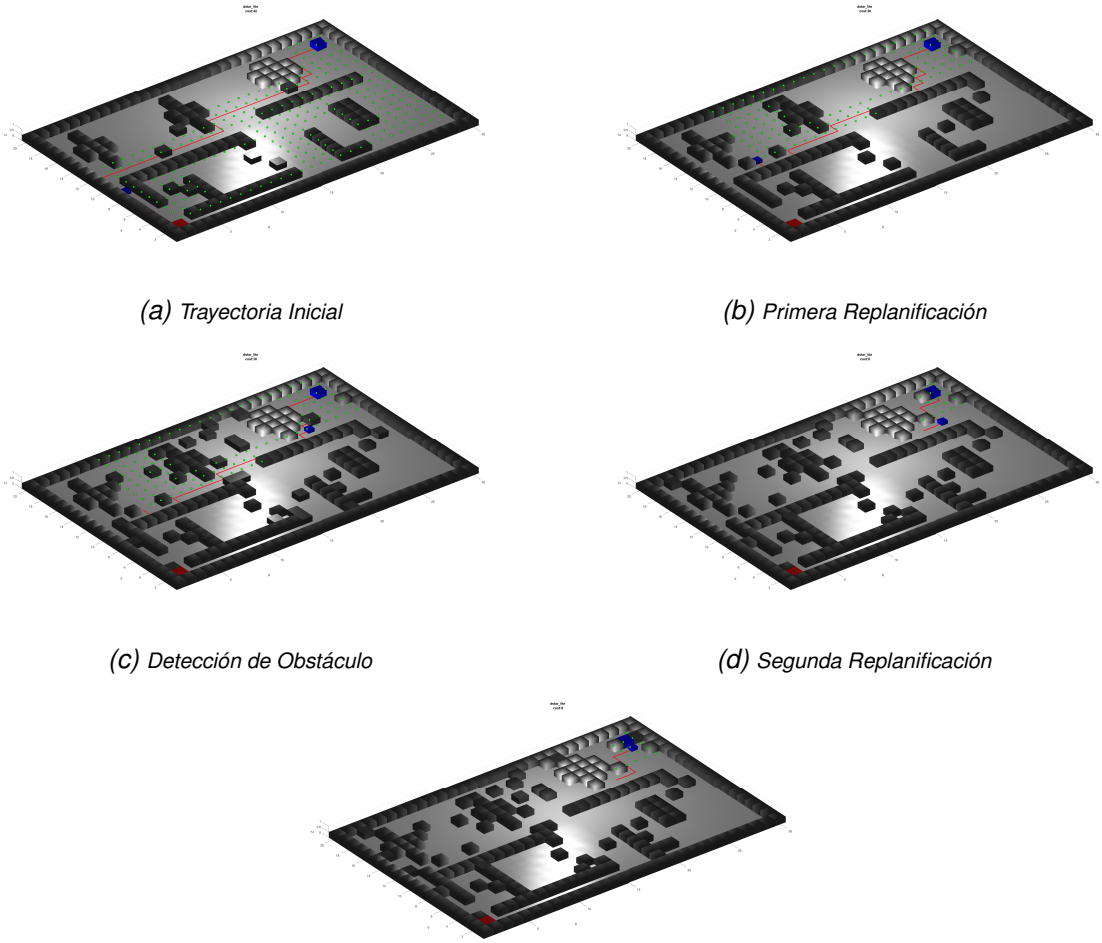
Figura 5.9: Comparación de tiempos de ejecución entre algoritmos en diferentes escenarios

5.3. Simulación en Entornos Dinámicos

En la Figura 5.10 se presenta una simulación en un entorno completamente dinámico. Se usó un entorno de simulación 3D y se añadieron obstáculos de manera incremental aleatoria conforme avanzaba la simulación.

En la Figura 5.10a, la trayectoria calculada por el algoritmo se muestra en línea roja. Con la adición aleatoria de obstáculos, dos de ellos interfieren con la trayectoria. El móvil, representado en azul, sigue la trayectoria hasta que se enfrenta al primer obstáculo (Figura 5.10b). A medida que avanza, se encuentra con un segundo obstáculo (Figura 5.10c), lo que provoca una segunda replanificación (Figura 5.10d). Finalmente, después de las replanificaciones, el móvil llega a su objetivo (Figura 5.10e).

Solo los algoritmos *D* Lite*, *Q-Learning* y *D** completaron esta simulación. Los demás algoritmos presentaron conflictos, como demoras en la determinación de la trayectoria, lo que redujo las rutas disponibles a medida que aumentaban los obstáculos, o fueron incapaces de replanificar y se detuvieron ante el primer obstáculo.



(a) *Trayectoria Inicial*

(b) *Primera Replanificación*

(c) *Detección de Obstáculo*

(d) *Segunda Replanificación*

(e) *Llegada al Destino*

Figura 5.10: Secuencia de simulaciones en 3D mostrando el avance del móvil en un entorno dinámico

5.3.1. Entorno Completamente Aleatorio

En esta sección se presentan los resultados ante un incremento en la aleatorización. Se define un mapa de simulación de los escenarios propuestos y se eligen de manera aleatoria los puntos de inicio y objetivo, así como la colocación de obstáculos dentro de la trayectoria calculada. Esto permite evaluar el desempeño de los algoritmos en entornos aleatorios, tanto con como sin obstáculos, a lo largo de un gran número de iteraciones.

Los resultados se presentan mediante un gráfico de barras que muestra el promedio de la longitud del camino y el tiempo de ejecución para cada algoritmo, junto con el intervalo de confianza del 95%. Este intervalo representa cuánto pueden variar los resultados respecto a la media observada, proporcionando una medida de la precisión

de los resultados y mostrando el rango en el que se espera que se encuentre el verdadero valor promedio con un 95% de certeza.

La Figura 5.11 muestra que, tal como se anticipaba, el algoritmo *D* Lite* presenta una longitud promedio mayor en comparación con los otros algoritmos. Esto se debe a que la distancia de Manhattan, implementada en *D* Lite*, restringe el avance diagonal, salvo que no haya otra opción. No obstante, la trayectoria resultante es realista y carece de pasos no ejecutables en la práctica. A pesar del aumento en la longitud del camino, la diferencia en la distancia no es significativa, incrementándose solo en 4 pasos con respecto al algoritmo *D** normal. En este caso el intervalo de confianza es amplio, dado que las distancias, al tener puntos de inicio y objetivo aleatorios, ocasionan que las longitudes de los caminos resultantes de los distintos algoritmos sean muy variables.

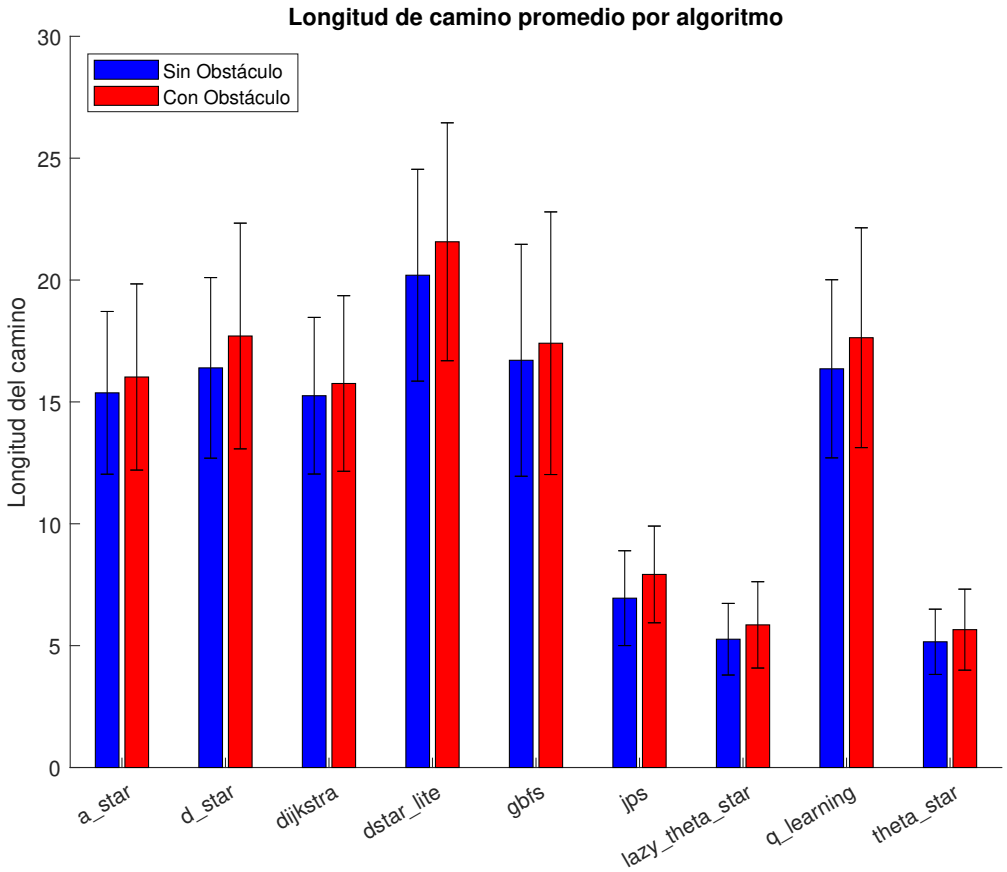


Figura 5.11: Comparación de longitud de camino promedio entre algoritmos en un entorno aleatorio

En la Figura 5.12 se presenta una comparación de los tiempos de ejecución entre los diferentes algoritmos. Al centrarse en los algoritmos *D** y *D* Lite*, se observa una

disminución significativa en el tiempo de ejecución. En particular, el intervalo de confianza para *D* Lite* es pequeño, lo que sugiere estabilidad en sus procesos. Esta baja variabilidad contrasta con la alta variabilidad observada en *Dijkstra* y *Q-Learning*, que muestran el peor desempeño en términos de tiempo de ejecución.

Este análisis demuestra la capacidad del algoritmo *D* Lite* para mantener tiempos de ejecución bajos y consistentes, convirtiéndolo en la mejor opción para aplicaciones dinámicas, donde la rapidez en la adaptación y respuesta es crucial. La alta variabilidad en el tiempo de ejecución de *Dijkstra* sugiere que no es adecuado para tales entornos.

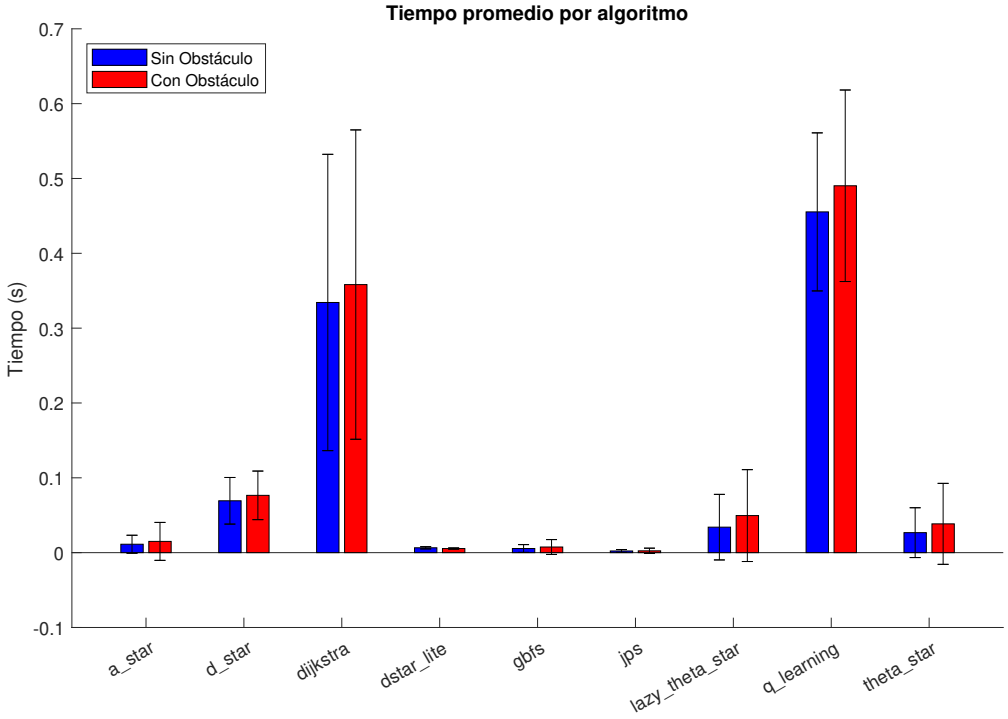


Figura 5.12: Comparación de Tiempos de Ejecución entre algoritmos

5.4. Paralelización D* Lite

En el documento de Ye [36], se presentan diversas ventajas al utilizar la paralelización en el algoritmo de Dijkstra. Entre estas ventajas se destacan la reducción significativa del tiempo de ejecución al distribuir la carga de trabajo entre múltiples núcleos, lo que resulta en una mejor utilización de los recursos del sistema. La paralelización permite que cada núcleo identifique su vértice más cercano al vértice fuente, realice una selección paralela para identificar el vértice globalmente más cercano, y luego transmita

este resultado a todos los núcleos. Esto mejora considerablemente el rendimiento del algoritmo en redes de gran escala y en aplicaciones que requieren procesamiento en tiempo real, como la planificación de rutas en sistemas de navegación y redes vehiculares.

Siguiendo este planteamiento, se puede considerar la paralelización del algoritmo D* Lite implementado. Para ello, se ha utilizado el algoritmo original como base y se han realizado varias modificaciones para adaptarlo a la paralelización en MATLAB. Específicamente, se dividió el grafo en subgrafos, permitiendo que cada procesador maneje un subconjunto de los nodos. Además, se modificó la actualización de las claves y los valores de costo para que cada procesador calcule estos valores de manera independiente, utilizando un bucle `parfor` en MATLAB para ejecutar estas operaciones en paralelo. Esta adaptación permite implementar la paralelización en el algoritmo D* Lite.

En la figura 5.13 se presenta la longitud promedio de los caminos generados por cada algoritmo. Se puede observar que la longitud de los caminos es consistente en todos los casos, lo que demuestra que la paralelización del algoritmo D* Lite produce rutas similares a las obtenidas con la versión no paralela.

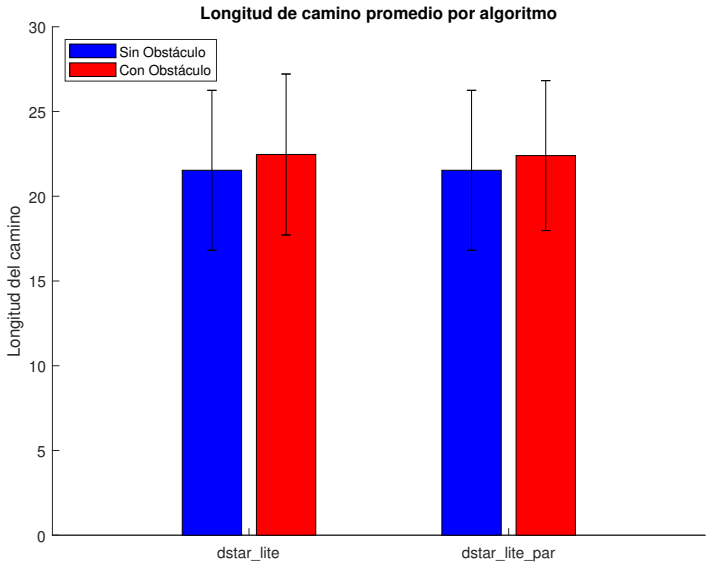


Figura 5.13: Comparación de longitud de camino promedio con y sin paralelización

En la figura 5.14 se presenta el tiempo promedio de ejecución por algoritmo, una mé-

trica de gran importancia. El tiempo promedio sin obstáculos es 0.345 segundos, considerablemente mayor en comparación con la versión no paralela del algoritmo. Esto se debe a que MATLAB necesita iniciar el conjunto de procesos de paralelización (*parallel pool*) en cada iteración, lo que introduce variabilidad y aumenta el intervalo de confianza. Sin embargo, una vez inicializado el conjunto de procesos y al introducir un obstáculo, el tiempo de ejecución se reduce de 0.129 segundos a 0.0100 segundos, disminuyendo el intervalo de confianza casi al mínimo. Esto sugiere que la paralelización no solo proporciona un tiempo promedio de determinación de ruta más bajo, sino también más consistente.

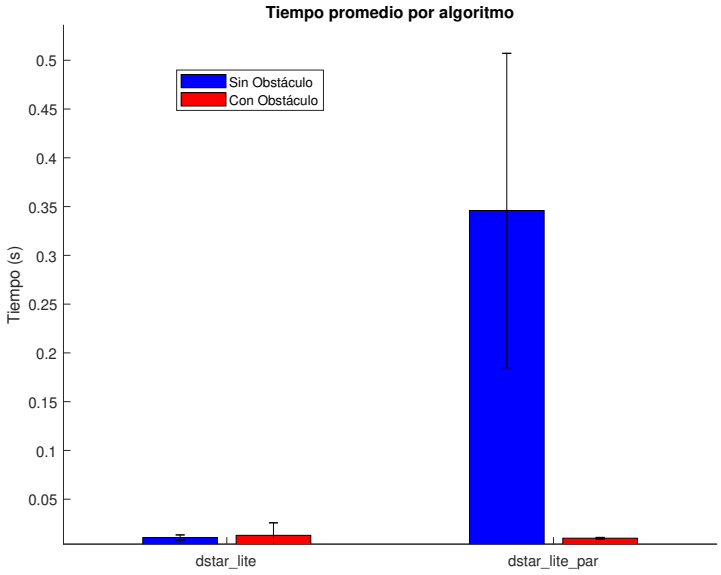


Figura 5.14: Comparación de Tiempos de Ejecución entre algoritmos con y sin paralelización

5.5. Paralelización Q-learning

Aunque el algoritmo de *Q-learning* es capaz de encontrar la ruta óptima durante su proceso de aprendizaje, este entrenamiento requiere un tiempo considerable. Sin embargo, como se mostró en las Figuras 5.9 y 5.12, este tiempo no representa una limitante significativa en comparación con otros algoritmos. Con el objetivo de reducir los tiempos de entrenamiento, se intentó paralelizar el algoritmo utilizando la funcionalidad `parfor` de MATLAB.

Desafortunadamente, este enfoque resultó contraproducente, ya que en lugar de dis-

minuir los tiempos de ejecución, estos se incrementaron drásticamente. Esto se debe a que el algoritmo de *Q-learning* no es fácilmente paralelizable debido a las actualizaciones en la tabla *Q*. Cada actualización de la tabla *Q* depende de pasos anteriores y coordinar los procesos paralelizados requiere un tiempo adicional significativo, lo que contrarresta los beneficios esperados de la paralelización.

5.6. Pruebas Sobre un Escenario Realista

Los algoritmos desarrollados pueden extrapolarse a otros enfoques, por ejemplo, se genera un entorno realista parecido a las calles de una urbe para evaluar el rendimiento de los algoritmos de planificación de rutas en condiciones cercanas a las del mundo real.

En la Figura 5.15 se presenta el escenario que simula una sección de una ciudad con varias cuadras delimitadas por calles. Este entorno fue construido utilizando la herramienta `DrivingScenario`, que permite la creación y manipulación de entornos de conducción.

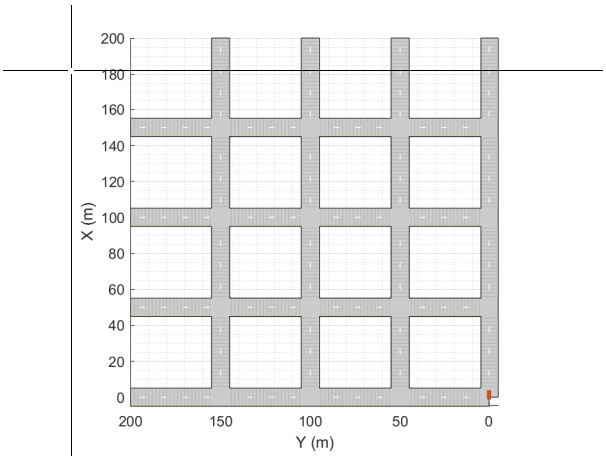


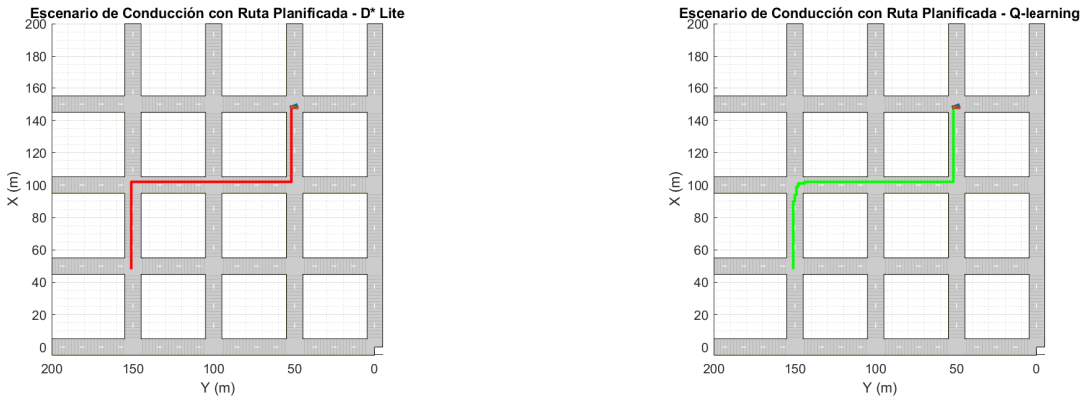
Figura 5.15: Escenario Realista Diseñado

Durante la creación de un entorno realista, se encontraron ciertos problemas que afectan la capacidad de los algoritmos para encontrar una ruta. Estos problemas incluyen:

- **Dirección de las calles**: Los algoritmos presentan dificultades para interpretar las direcciones de las calles.
- **Adaptación a las restricciones de tráfico**: Es necesario ajustar los algoritmos

de planificación de rutas para que puedan incorporar y respetar las direcciones y restricciones de tráfico, como la señalización y los semáforos, lo cual les permitirá comprender mejor el entorno y generar rutas más realistas.

Aunque se presentaron estas dificultades, se logró probar los algoritmos *D* Lite* y *Q-learning* en dicho entorno, sin considerar la dirección de las calles, obteniendo un funcionamiento correcto, como se presenta en las Figuras 5.16a y 5.16b. En estas figuras, las rutas se presentan sobre el mapa con diferentes colores para distinguir el algoritmo utilizado, y se ha implementado un vehículo que sigue dichas trayectorias.



(a) D Lite en un entorno urbano simulado* *(b) Q-learning en un entorno urbano simulado*

Figura 5.16: Pruebas de los algoritmos D Lite y Q-learning en un escenario realista utilizando* `DrivingScenario`.

El entorno de `DrivingScenario` ofrece la capacidad de controlar la velocidad del vehículo, lo que lo convierte en un escenario ideal para realizar pruebas más avanzadas en el futuro. Utilizando este entorno, se pueden mejorar tanto los algoritmos presentados como el propio escenario, incorporando elementos adicionales como semáforos, señalización y restricciones de tráfico. En esta ocasión, se ha empleado un escenario básico para evaluar el funcionamiento inicial de los algoritmos en un entorno más realista. Para trabajos futuros, se recomienda expandir y sofisticar el escenario, permitiendo una evaluación más completa y detallada de los algoritmos de planificación de rutas.

Conclusiones y recomendaciones

6.1. Conclusiones

Tanto el algoritmo D* Lite como Q-Learning demostraron ser capaces de superar exitosamente entornos dinámicos. La concepción inicial del algoritmo D* Lite, enfocada en la simplicidad, resultó ser beneficiosa. Como se observó en los resultados, esta simplicidad contribuyó a la reducción del tiempo de procesamiento en la determinación de la trayectoria, permitiendo su implementación en entornos realistas donde el tiempo de replanificación debe ser mínimo para evitar obstáculos y generar una nueva ruta según sea necesario.

Una de las ventajas clave del D* Lite es su enfoque en calcular la lista de nodos procesados desde el nodo objetivo. Esta estrategia minimiza los cálculos innecesarios cuando se presenta un obstáculo y se requiere una replanificación. Al reducir la carga computacional, el algoritmo puede reaccionar rápidamente a cambios en el entorno, lo cual es indispensable en aplicaciones donde la adaptabilidad y la velocidad son fundamentales. El enfoque de expandir esta lista a todos los vecinos con coste mínimo no solo simplificó el algoritmo, sino, permite tener en cuenta rutas alternativas a la principal, como se evidenció en las diversas simulaciones.

Por otro lado, el algoritmo Q-Learning también mostró un desempeño robusto en entornos dinámicos. Su capacidad para aprender y adaptarse a través de la interacción con el entorno le permite encontrar soluciones óptimas en escenarios complejos y cambiantes. Esta adaptabilidad es particularmente valiosa en situaciones donde las condiciones del entorno pueden variar de manera impredecible. La ventaja principal del aprendizaje por refuerzo es que no necesita una base de datos preexistente, ya que mediante el entrenamiento aprende el entorno para determinar la trayectoria más adecuada.

El algoritmo propuesto es una versión híbrida entre los modelos básico y dinámico presentados, combinando recompensas estáticas con la variación de parámetros como la tasa de aprendizaje y la tasa de exploración. Lo interesante de este algoritmo es lo relativamente rápido que entrena la tabla Q y crea el camino óptimo, tal como se mostró en las simulaciones. Sin embargo, uno de los puntos negativos es el tiempo de

ejecución, que es el segundo más alto en comparación con el resto de los algoritmos. Esto se debe a que en cada iteración debe recalcular la tabla Q, ya que el entorno cambia tras la aparición de un obstáculo. A pesar de esto, el intervalo de confianza que proporciona en las pruebas es muy bajo, generando una alta confiabilidad.

Si bien nuestro enfoque no toma en cuenta los tiempos de viaje, los algoritmos fueron probados ante obstáculos. Estos obstáculos no se restringen a objetos específicos, sino a cualquier elemento que se atraviese en el camino, como bloqueos por accidentes o congestionamiento vehicular. El algoritmo busca la ruta óptima en función de la disponibilidad del mapa, intentando que esta sea la más corta y eficiente. Los escenarios propuestos no cuentan con limitantes como tráfico vial o la dirección de las calles, por lo que no se probó la métrica de tiempo de viaje. No obstante, la aplicabilidad del algoritmo dependería del escenario y del vehículo en particular, ya que la ruta será determinada por el algoritmo, pero el movimiento del vehículo no está configurado en este trabajo. Nos centramos en entornos dinámicos y la rapidez en el procesamiento, dado que el tiempo de viaje depende en gran medida de los bloqueos que existan hacia el objetivo, lo cual no es relevante para el enfoque planteado. Futuras investigaciones podrían integrar métricas de tiempo de viaje y evaluar el rendimiento del algoritmo en escenarios con tráfico real y restricciones de dirección, proporcionando así una solución más completa y adaptable a las condiciones del tráfico urbano.

La simplicidad y eficiencia del D* Lite, combinadas con la adaptabilidad del Q-Learning, demuestran que estos algoritmos son opciones viables y efectivas para la navegación en entornos dinámicos, considerando los movimientos permitidos para un vehículo autónomo. La reducción en el tiempo de procesamiento y la capacidad de replanificación eficiente son aspectos que fortalecen su aplicabilidad en contextos de ambientes reales.

6.2. Recomendaciones

Antes de realizar la implementación, es necesario entender cómo funciona el algoritmo iteración por iteración, con el objetivo de tener presente durante la codificación, el proceso que se debe seguir para obtener los resultados deseados y evitar procesos

innecesarios.

Para el desarrollo de algoritmos de planificación de trayectorias en entornos dinámicos es importante mantener el tiempo de procesamiento al mínimo para que la replanificación ante cambios de entorno sea lo más rápido posible. Esto se puede conseguir de dos formas, la primera consiste en mantener un enfoque lo más simple posible, evitando el uso de condicionales y variables innecesarias, un segundo enfoque se puede implementar al optimizar al máximo un algoritmo ya existente.

El uso de la distancia euclidiana para la determinación de pesos en los nodos, resulta teóricamente en caminos más cortos, sin embargo, para *Vehículo Terrestre No Tripulado (por sus siglas en inglés, Unmanned Ground Vehicle)* (UGV)s se debe comprobar que su uso no genere movimientos no realizables en la práctica, como se mostró en la Figura 4.1. Implementar físicamente los algoritmos permitiría revisar su funcionamiento y validar que los movimientos sean ejecutables por el vehículo. Esto aseguraría que las pruebas no se limiten a la teoría y simulación, sino que se verifiquen en condiciones reales, garantizando así la aplicabilidad y fiabilidad de los algoritmos.

Utilizar la distancia Manhattan resulta beneficioso en entornos urbanos con obstáculos que impidan el paso del vehículo UGV. Este enfoque es adecuado para escenarios donde los movimientos están restringidos a caminos rectos, como en ciudades con calles y avenidas bien definidas o entornos de rescate mapeados a modo de cuadrícula. Sin embargo, en el caso de vehículos aéreos no tripulados (*Vehículo Aéreo No Tripulado (por sus siglas en inglés, Unmanned Aerial Vehicle)* (UAV)s), el enfoque Manhattan podría no ser la mejor opción. Los UAVs tienen la capacidad de moverse libremente en tres dimensiones, y los movimientos diagonales (distancia euclidiana) suelen ser más eficientes y realistas. En estos casos, es preferible utilizar un enfoque que permita movimientos diagonales, ya que llegar de un punto a otro de manera directa es más eficiente que realizar movimientos en forma de escalera (primero en una dirección y luego en otra).

En simuladores de redes vehiculares como SUMO, se utilizan comúnmente los algoritmos Dijkstra y A*, los cuales, como se presentó en el desarrollo de este trabajo, tienen un peor desempeño en comparación con el algoritmo D* Lite. Sería beneficioso implementar D* Lite para el proceso de definición de rutas, ya que no solo se disminuiría

el tiempo de generación de rutas, sino que también se podría adaptar para detectar aglomeraciones de vehículos debido a congestiones o accidentes como obstáculos y replanificar en consecuencia. Esto permitiría una mayor eficiencia y adaptabilidad en la gestión del tráfico vehicular en entornos urbanos dinámicos. Además, el uso de ROS (Sistema Operativo de Robots) como plataforma de simulación podría proporcionar un ambiente flexible y robusto para probar y evaluar los algoritmos de planificación de rutas propuestos, debido a su facilidad para integrar diferentes sensores y módulos, así como su amplia adopción en la investigación robótica.

Referencias

[1] A. Fallis, *Autonomous Ground Vehicle*, 2013, vol. 53, num. 9.

[2] K. Berns y E. von Puttkamer, *Autonomous Land Vechicles - Step towards Service Robots*.

[3] N. Phillips, “Autonomous Vehicles - Safety, Deployment and Effect on Infraestructure,” 2002.

[4] H. S. Becker, “Introduction to the second edition,” *Campus Power Struggle*, pp. 1–4, 2017.

[5] P. E. Hart, N. J. Nilsson, y B. Raphael, “A formal basis for the heuristic determination of minimum cost paths,” *IEEE Transactions on Systems Science and Cybernetics*, vol. 4, num. 2, pp. 100–107, 1968.

[6] S. M. LaValle, “Planning algorithms,” *Planning Algorithms*, vol. 9780521862059, pp. 1–826, 2006.

[7] S. Koenig y M. Likhachev, “D*lite,” in *AAAI/IAAI*, 2002. [En línea]. Disponible: https://api.semanticscholar.org/CorpusID:208940224

[8] Y. Luo, J. Lu, Q. Qin, y Y. Liu, “Improved jps path optimization for mobile robots based on angle-propagation theta* algorithm,” *Algorithms*, vol. 15, num. 6, p. 198, 2022.

[9] K. Yigit, “Path planning methods for autonomous underwater vehicles,” Cambridge, Massachusetts, Jun. 2011.

[10] P. Mendonca, “C THETA* : Path Planning on Grids,” Master’s thesis, University of Windsor, 2016. [En línea]. Disponible: https://scholar.uwindsor.ca/etd/5654

[11] R. S. Sutton, A. G. Barto, y A. B. Book, “Reinforcement Learning, 1st ed,” 1998.

[12] S. Russell y P. Norving, “Artificial Intelligence - A Modern Approach.”

[13] B. Jang, M. Kim, G. Harerimana, y J. W. Kim, “Q-Learning Algorithms: A Comprehensive Classification and Applications,” *IEEE Access*, vol. 7, pp. 133 653–133 667, 2019.

[14] J. Clifton y E. Laber, “Q-Learning: Theory and applications,” *Annual Review of Statistics and Its Application*, vol. 7, pp. 279–301, 2020.

[15] G. Cook y F. Zhang, *Mobile robots: Navigation, control and sensing, surface robots and AUVS: Second edition*, 2020.

[16] H. Cheng, *Autonomous Intelligent Vehicles - Theory, Algorithms, and Implementation*, 2011.

[17] I. K. Sethi, *Autonomous Vehicles and Systems*, 2023.

[18] “Vista de Vehículos de guiado autónomo (AGV) en aplicaciones industriales: una revisión | REVISTA POLITÉCNICA.” [En línea]. Disponible: https://revistas.elpoli.edu.co/index.php/pol/article/view/1478/1208

[19] A. Drira, H. Pierreval, y S. Hajri-Gabouj, “Facility layout problems: A survey,” *Annual Reviews in Control*, vol. 31, num. 2, pp. 255–267, 2007. [En línea]. Disponible: https://www.sciencedirect.com/science/article/pii/S1367578807000417

[20] G. P. Reddy, V. S. Chinthala, M. Raja Venkatesh, V. MVN, y R. Purohit, “A Review On Facility Layout Design Of An Automated Guided Vehicle In Flexible Manufacturing System,” *Materials Today: Proceedings*, vol. 5, pp. 3981–3986, 2018.

[21] P. Beinschob, M. Meyer, C. Reinke, V. Digani, C. Secchi, y L. Sabattini, “Semi-automated map creation for fast deployment of AGV fleets in modern logistics,” *Robotics and Autonomous Systems*, vol. 87, 2016.

[22] S. Lu, C. Xu, R. Zhong, y L. Wang, “A RFID-enabled positioning system in automated guided vehicle for smart factories,” *Journal of Manufacturing Systems*, vol. 44, pp. 179–190, 2017.

[23] J. Kang, J. Lee, H. Eum, C. H. Hyun, y M. Parks, “An application of parameter extraction for AGV navigation based on computer vision,” *2013 10th International Conference on Ubiquitous Robots and Ambient Intelligence, URAI 2013*, pp. 622–626, 2013.

[24] K. Osman, J. Ghommam, y M. Saad, “Combined road following control and automatic lane keeping for automated guided vehicles,” 2016, pp. 1–6.

[25] M. Pedan, M. Gregor, y D. Plinta, “Implementation of Automated Guided Vehicle System in Healthcare Facility,” *Procedia Engineering*, vol. 192, pp. 665–670, 2017. [En línea]. Disponible: http://dx.doi.org/10.1016/j.proeng.2017.06.115

[26] A. M. Hellmund, S. Wirges, Ö. . Ta, C. Bandera, y N. O. Salscheider, “Robot operating system: A modular software framework for automated driving,” *IEEE Conference on Intelligent Transportation Systems, Proceedings, ITSC*, pp. 1564–1570, 2016.

[27] M. Wei, S. Wang, J. Zheng, y D. Chen, “UGV Navigation Optimization Aided by Reinforcement Learning-Based Path Tracking,” *IEEE Access*, vol. 6, pp. 57 814–57 825, 2018.

[28] D. Agarwal y P. S. Bharti, “Matlab simulation of path planning and obstacle avoidance problem in mobile robot using sa, pso and fa,” in *2020 IEEE International Conference for Innovation in Technology (INOCON)*, 2020, pp. 1–6.

[29] Y. Long y H. He, “Robot path planning based on deep reinforcement learning,” in *2020 IEEE Conference on Telecommunications, Optics and Computer Science (TOCS)*, 2020, pp. 151–154.

[30] Y. Sun, M. Fang, y Y. Su, “Agv path planning based on improved dijkstra algorithm,” *Journal of Physics: Conference Series*, vol. 1746, num. 1, p. 012052, jan 2021. [En línea]. Disponible: https://dx.doi.org/10.1088/1742-6596/1746/1/012052

[31] X. Wang, J. Lu, F. Ke, X. Wang, y W. Wang, “Research on agv task path planning based on improved a* algorithm,” *Virtual Reality Intelligent Hardware*, vol. 5, num. 3, pp. 249–265, 2023. [En línea]. Disponible: https://www.sciencedirect.com/science/article/pii/S2096579622001176

[32] J. Gross, M. De Petrillo, J. Beard, H. Nichols, T. Swiger, R. Watson, C. Kirk, C. Kilic, J. Hikes, E. Upton, D. Ross, M. Russell, Y. Gu, y C. Griffin, “Field-testing of a

UAV-UGV team for GNSS-denied navigation in subterranean environments," *Proceedings of the 32nd International Technical Meeting of the Satellite Division of the Institute of Navigation, ION GNSS+ 2019*, pp. 2112–2124, 2019.

[33] D. A. Álvarez, "Planificación de trayectorias para el control autónomo de un cuadricópteroutilizando técnicas de visión artificial y aprendizaje profundo," 2021. [En línea]. Disponible: https://github.com/diegoalvaroalvarez/TFM2021UC-Diego-Alvaro

[34] P. Corke, "Robotics toolbox for matlab," 2024, accessed: 2024-06-19. [En línea]. Disponible: https://github.com/petercorke/robotics-toolbox-matlab

[35] ai winter, "matlab_motion_planning," 2023, gitHub repository. [En línea]. Disponible: https://github.com/ai-winter/matlab_motion_planning/tree/master

[36] Z. Ye, "An implementation of parallelizing dijkstras algorithm," *CSE633 Course Project, ID*, pp. 3715–8138, 2012.

Anexo

A.1. Anexo A: Enlace de Repositorio

El código fuente de este proyecto se puede encontrar en el siguiente enlace de GitHub:

`https://github.com/LucasSaenz4118/Path_planning_Matlab2024`