Ing. Ivan Michael Gomez Azpilcueta

Algoritmos de Optimización para búsqueda de trayectorias en espacios tridimensionales

2026

Universidad Autónoma de Querétaro

Facultad de Ingeniería

Algoritmos de Optimización para búsqueda de trayectorias en espacios tridimensionales

Que como parte de los requisitos para obtener el Grado de

Maestro en Ciencias en Inteligencia Artificial

Presenta:

Ing. Iván Michael Gómez Azpilcueta

Dirigido por:

Dr. Arturo González Gutiérrez

Co-dirigido por:

Dra. Adriana Rojas Molina

Querétaro, Qro. Febrero de 2026

Universidad Autónoma de Querétaro

Facultad de Ingeniería

Maestría en Ciencias en Inteligencia Artificial

Algoritmos de Optimización para búsqueda de trayectorias en espacios tridimensionales

Tesis

Que como parte de los requisitos para obtener el Grado de

Maestro en Ciencias en Inteligencia Artificial

Presenta:

Ing. Iván Michael Gómez Azpilcueta

Dirigido por:

Dr. Arturo González Gutiérrez

Co-dirigido por:

Dra. Adriana Rojas Molina

Dr. Arturo González Gutiérrez
Presidente
Dra. Adriana Rojas Molina
Secretaria
Dr. András Takács
Vocal
Dr. Fidel González Gutiérrez
Suplente
M.C. Pedro Ayala Elizarraraz
Suplente

Centro Universitario, Querétaro, Qro.

Febrero 2026

México

Dedicado a mis padres Mario Marcos Gómez Sotelo y Adriana Azpilcueta Morales, y en especial a mi esposa Claudia Ángeles Velázquez

Agradecimientos

Se agradece al Consejo Nacional de Ciencia y Tecnología (CONACYT) por proveer los fondos necesarios para realizar esta investigación a través del Programa de Becas Nacionales. Así como a la Universidad Autónoma de Querétaro por la Beca Institucional.

En este espacio quiero expresar mi más sincero agradecimiento al Dr. Arturo González Gutiérrez y a la Dra. Adriana Rojas Molina por su gran apoyo en la realización de esta investigación. Gracias a su conocimiento y dedicación en la enseñanza, me guiaron durante todo el proceso de investigación y desarrollo de este tema innovador dentro de mi área de estudio.

Además, me gustaría resaltar su paciencia y comprensión a lo largo de todo este camino académico y profesional, brindándome un apoyo incondicional y una dirección clara para alcanzar los objetivos planteados en este proyecto de investigación. Sin su ayuda y orientación, no habría sido posible alcanzar este logro académico y estoy profundamente agradecido por su tiempo, conocimiento y dedicación.

Resumen en español.

En Inteligencia Artificial, la búsqueda de trayectorias consiste en encontrar el camino entre dos puntos dados en un escenario creado digitalmente. Se toman en cuenta criterios como la trayectoria más corta, más barata o rápida, entre dos puntos dispersos en un escenario. Los escenarios creados pueden ser multidimensionales.

Existen diferentes algoritmos de búsqueda de trayectorias que aplican técnicas heurísticas para encontrar la solución a la búsqueda de trayectorias. Mediante dichas heurísticas es posible determinar de manera eficiente (rápida) aunque aproximada, qué tan lejos se encuentran un punto de otro en el espacio multidimensional. Normalmente se utiliza la distancia lineal al objetivo, ya que claramente es la mínima y por tanto la más rápida de obtener. Sin embargo, existen escenarios donde, en la búsqueda de la trayectoria, intervienen obstáculos dispersos, los cuales no permiten que la distancia lineal sea la trayectoria viable.

En este trabajo se presenta un análisis comparativo entre los cuatro principales algoritmos de búsqueda de trayectorias en un entorno bidimensional con elementos gráficos tridimensionales y su posterior implementación mediante el lenguaje de programación Python. Dichos algoritmos son: el algoritmo de búsqueda en anchura, el algoritmo de búsqueda en profundidad, el algoritmo de Dijkstra y el algoritmo A*.

Los algoritmos de búsqueda de trayectorias fueron programados en una computadora con un procesador Intel Core i5-3570 a 3.40 GHz de 64 bits y 32 GB de memoria RAM a 1660 MHz. Los criterios utilizados para evaluar los programas y seleccionar aquel con el mejor rendimiento computacional son: número de bloques procesados durante la ejecución del programa, número de pasos a seguir en la trayectoria encontrada, tiempo de ejecución del programa durante las pruebas, así como la memoria y porcentaje de CPU utilizados durante la ejecución de las pruebas.

Con base en el análisis experimental llevado a cabo, se determinó que el algoritmo con mejor desempeño es el Algoritmo A*. Este algoritmo fue implementado finalmente en un entorno de búsqueda de trayectorias en tiempo real.

Palabras clave: Búsqueda de trayectorias, Dijkstra, A*, búsqueda en anchura, búsqueda en profundidad, optimización.

Abstract

In Artificial Intelligence, pathfinding consists of finding the path between two given points in a digitally created scenario. Criteria such as the shortest, cheapest, or fastest path are considered between two scattered points in a scenario. Created scenarios can be multidimensional.

There are different pathfinding algorithms that apply heuristic techniques to find the solution to pathfinding. With these heuristics, it is possible to determine in an efficient (fast) but approximate way how far one point is from another in multidimensional space. The linear distance to the goal is usually used since it is clearly the minimum and therefore the fastest to obtain. However, there are scenarios where scattered obstacles engage in pathfinding, which do not allow the shortest distance to be the viable path. This work presents a comparative analysis between the four main pathfinding algorithms in a three-dimensional environment and their subsequent implementation using the Python programming language. These algorithms are breadth-first search, depth-first search, Dijkstra's algorithm, and the A* algorithm.

Pathfinding algorithms were programmed into a computer with an Intel Core i5-3570 processor at 3.40 GHz, 64-bit architecture, and 32 GB of RAM at 1660 MHz.

The criteria used to evaluate the programs and select the one with the best computational performance are: the number of blocks processed during program execution, the number of steps to follow in the found path, program execution time during testing, as well as memory and CPU percentage used during testing.

Based on the experimental analysis conducted, it was determined that the algorithm with the best performance is the A* algorithm. This algorithm was finally implemented in a real-time pathfinding environment.

Keywords: Pathfinding, Dijkstra, A*, breadth-first search, depth-first search, optimization.

Acrónimos

BFS: Breadth-first search. Búsqueda en anchura.

CPU: Central Processing Unit. Unidad de procesamiento central.

CSV: Comma-Separated Values. Valores Separados por Comas

DFS: Depth-first search. Búsqueda en profundidad.

FPS: Frames per second. Fotogramas por segundo.

GPS: Global Positioning System. Sistema de Posicionamiento Global.

IA: Inteligencia Artificial.

ID: Identificador.

IDA*: Iterative Deepening A*. Profundización Iterativa de A*

MB: Megabyte.

NPC: Non-Playable Character. Personaje no jugador.

PID: Identificador de procesos.

RAM: Random Access Memory. Memoria de acceso aleatorio.

Índice

Índice de Tablas.

Índice de Figuras

Índice de Algoritmos

1 Introducción

Las matemáticas es una ciencia exacta, donde una parte muy importante es la geometría, que estudia las propiedades y medidas de figuras en un plano o espacio. En la civilización babilónica, se utilizaba la geometría para medir volúmenes, longitudes y parcelas de tierra, y posteriormente se transfirió este conocimiento a los griegos, quienes hicieron grandes descubrimientos en matemáticas.

Euclides, un matemático y geómetra griego considerado el padre de la geometría, desarrolló un tratado matemático y geométrico compuesto por trece libros llamados "Los Elementos de Euclides", también conocidos como geometría Euclidiana. Uno de los postulados de Euclides es el espacio euclídeo, un espacio geométrico en el que se analizan proposiciones que se consideran evidentemente verdaderas y a partir de ellas se generan nuevas proposiciones a través de deducciones lógicas, cuyo valor de verdad es verdadero.

La geometría computacional surgió a finales de la década de 1970, enfocándose en el estudio de espacios geométricos para resolver problemas mediante el diseño y análisis de algoritmos. La búsqueda de trayectorias en la geometría computacional consiste en encontrar el camino más corto entre dos puntos dados, librando obstáculos mediante programas de Inteligencia Artificial.

Como problema fundamental de la Inteligencia Artificial, la búsqueda de trayectorias tiene un impacto indirecto en la experiencia de un escenario gráfico. Aunque no modifica visualmente el entorno, el algoritmo de búsqueda de trayectorias determina cómo un personaje puede moverse hacia un destino deseado [1].

1.1 Planteamiento del problema

La búsqueda de trayectorias es una componente importante en muchos campos, como videojuegos, robótica, simulación de multitudes y trazado de trayectorias con GPS [2]. Esta investigación se centrará en la implementación de algoritmos de búsqueda de trayectorias en entornos bidimensionales con elementos gráficos tridimensionales y la búsqueda en tiempo real de objetivos.

En un escenario bidimensional, dado de un punto de origen, un punto de destino y un conjunto de elementos gráficos tridimensionales regulares como obstáculos, la trayectoria más corta es aquella que conecta el origen con el destino, evitando obstáculos y con la longitud total más pequeña.

La búsqueda de trayectorias puede conducir a una pérdida de recursos de la CPU cuando el algoritmo intenta una trayectoria no factible. Por lo tanto, el diseño de algoritmos eficientes es importante para evitar el congelamiento de la imagen, lo que a su vez lleva a un ahorro de recursos y mejora de la experiencia del usuario.

Existen diferentes algoritmos de búsqueda de trayectorias que se enfocan en encontrar la trayectoria más corta, aunque a veces no sea la solución más eficiente para el escenario en el que están implementados. Estos algoritmos se pueden aplicar en entornos estáticos, dinámicos y en tiempo real [3]. Comúnmente, se emplean estructuras de datos predefinidas para guiar el movimiento de los personajes autónomos para resolver trayectorias.

Los avances recientes en inteligencia artificial ofrecen la posibilidad de desarrollar escenarios con personajes autónomos que se adaptan dinámicamente a varios niveles de dificultad del escenario y a entornos variables.

1.2. Justificación

Al implementar un algoritmo eficiente, se busca mejorar el uso de los recursos de una computadora, lo que resulta en un mejor rendimiento del sistema. En los últimos años, los investigadores han estado enfocándose en desarrollar algoritmos de búsqueda de trayectorias que permitan encontrar rutas más cortas y evitar obstáculos en entornos bidimensionales con elementos gráficos tridimensionales. Sin embargo, en la literatura actual, no hay mucha información disponible sobre cómo estos algoritmos afectan el consumo de recursos computacionales, como el uso de memoria o la utilización del CPU. La mayoría de estas investigaciones se centran únicamente en el tiempo de ejecución como único criterio de evaluación. [4]

Cuando se habla de plataformas de desarrollo en entornos bidimensionales con elementos gráficos tridimensionales, no se limita solo a la creación de escenarios para fines recreativos. También se pueden utilizar para resolver problemas que requieren una representación gráfica, como la búsqueda de trayectorias más cortas o la evasión de obstáculos en una trayectoria dada.

La optimización de estos algoritmos de búsqueda de trayectorias es esencial para que los escenarios creados puedan funcionar con memoria y tiempo de CPU limitados. Esta investigación se centrará en la implementación y evaluación de algoritmos de búsqueda de trayectorias. Si bien la búsqueda de trayectorias puede parecer simple, la complejidad aumenta cuando se agregan elementos como la creación de entornos o escenarios con diferentes tipos de superficies o el movimiento utilizado en personajes autónomos que se deben desplazar a una ubicación específica.

En resumen, el objetivo de esta investigación es encontrar algoritmos de búsqueda de trayectorias eficientes y optimizados que permitan resolver problemas complejos en entornos bidimensionales con elementos gráficos tridimensionales, y que sean capaces de funcionar con recursos limitados.

2 Antecedentes

El estudio de la búsqueda de trayectorias es esencial en muchos campos, desde la robótica hasta los videojuegos. A lo largo de los años, se han desarrollado diversos algoritmos y estrategias para abordar este problema, y varios de ellos han sido objeto de estudio y análisis en artículos publicados entre 2007 y 2022.

En 2007, se utilizó un algoritmo genético para explorar el espacio de los algoritmos de búsqueda de trayectorias en Lagoon, una simulación en tiempo real de juegos y entrenamientos de estrategia naval en 3D [5]. La elección de un algoritmo genético se debió a que los algoritmos de búsqueda de trayectorias A* consumen muchos recursos computacionales cuando se trata de varios personajes autónomos.

En 2011, Alex Machado implementó un algoritmo de búsqueda en tiempo real con un algoritmo genético. En su artículo, presentó un método para optimizar el proceso de encontrar trayectorias, utilizando un modelo basado en algoritmos genéticos y A* para sistemas en tiempo real en entornos de realidad virtual [6].

En 2012, Ulysses O. Santos implementó un algoritmo de búsqueda en anchura (BFS, Breadth First Search) con un algoritmo genético. Demostró que la arquitectura “Búsqueda de rutas basada en patrones con algoritmo genético” (PPGA, Pattern-based pathfinding with genetic algorithm) funciona mejor que la clásica A* cuando se trata de más de un personaje autónomo [7].

En cuanto a los videojuegos, en el Congreso Internacional de Computación Evolutiva de 2011 se presentaron varios trabajos sobre la búsqueda de trayectorias en el popular juego de Ms. Pac-Man. Gustavo Recio modificó el juego utilizando un algoritmo de Ant-Colony, que fue probado en las competencias de Ms. Pac-Man celebradas durante el congreso [8]. Sin embargo, en 2016, Iris Hunkeler desarrolló un algoritmo llamado "fairGhosts" que superó en respuesta y tiempos a los fantasmas del juego del algoritmo generado por Gustavo Recio.

Es importante destacar que estos estudios y desarrollos son solo una muestra de la diversidad de enfoques que se están utilizando actualmente para abordar el problema de la búsqueda de trayectorias en distintas áreas, lo que demuestra la relevancia y complejidad de esta área de investigación.

En el año 2016, Moh. Zikky se enfocó en mejorar los algoritmos para la navegación de los fantasmas en el popular juego de Pac-Man. En este contexto, Zikky logró optimizar el algoritmo A* y su método resultó muy efectivo en la búsqueda de trayectorias más cortas. La optimización se llevó a cabo mediante un rastreo efectivo utilizando una línea de segmentación [9].

En 2017, Anggina Primanita llevó a cabo una comparación entre el algoritmo A* y el algoritmo de profundización iterativa A* (IDA*, Iterative Deepening A*), para los personajes no jugadores (NPCs) en juegos de rol. A pesar de que el algoritmo A* es el más utilizado en la búsqueda de trayectorias, demanda un consumo excesivo de memoria en comparación con otros algoritmos, como el algoritmo de profundización iterativa A*. La investigación concluyó que tanto el uso de memoria como el tiempo necesario para A* e IDA* se ven afectados por el tamaño del escenario (cantidad de nodos), la posición de los obstáculos en el mapa y la cantidad de obstáculos. Por lo tanto, el algoritmo IDA* es generalmente más eficiente que A* en cuanto a uso de memoria y tiempo, especialmente si el escenario no tiene obstáculos, pero puede no ser efectivo si el personaje autónomo y el jugador están en la misma posición y cubiertos por obstáculos [10].

En 2018, Aimi Najwa desarrolló una investigación para describir y comparar el algoritmo tradicional, el algoritmo A* y el algoritmo Honey Bee en la búsqueda de trayectorias. El algoritmo Honey Bee fue propuesto por S. Nakrani y C. Tovey en 2004. La investigación demostró que el algoritmo A* es eficiente cuando se trata de escenarios sin obstáculos, ya que puede buscar en una trayectoria directa sin cambiar la ruta. Sin embargo, el algoritmo Bee es significativamente más rápido que A*, especialmente en escenarios más grandes. Por lo tanto, el algoritmo Bee resulta más eficiente que A* en entornos de escenario de mayor tamaño y con la existencia de obstáculos [11].

En [12], Yoppy Sazaki utilizó el algoritmo A* para encontrar la trayectoria más corta para los personajes no autónomos en el escenario desarrollado, combinado con el algoritmo dinámico de búsqueda de trayectorias para evitar obstáculos estáticos o dinámicos en la trayectoria. Los resultados experimentales demostraron que la combinación de ambos métodos resulta en un mejor rendimiento en comparación con A* utilizado solo. La combinación de ambos métodos se puede implementar adecuadamente en escenarios de carreras de autos con condiciones de pista sin obstáculos, así como en condiciones de pista con obstáculos estáticos. Sin embargo, en las trayectorias con obstáculos dinámicos, la combinación de ambos métodos solo es efectiva bajo ciertas condiciones [12].

2.1 Fundamentación Teórica

La Inteligencia Artificial (IA) es un campo de la informática que se dedica al desarrollo de sistemas capaces de realizar tareas que requieren inteligencia humana. Estos sistemas emplean algoritmos y procesos computacionales para simular el aprendizaje, la percepción, el razonamiento y la toma de decisiones. El objetivo de la IA es emular la inteligencia humana y mejorar la capacidad de las máquinas para realizar tareas específicas, como el reconocimiento de patrones, la toma de decisiones, el procesamiento de lenguaje natural y la resolución de problemas. La Inteligencia Artificial tiene aplicaciones en una amplia variedad de campos, desde la asistencia en dispositivos electrónicos hasta la optimización de procesos industriales y la medicina.

2.1.1Teoría de Grafos

Un grafo G=(V,E), también conocido como red, es una estructura matemática compuesta por dos conjuntos: vértices o nodos (V), y bordes o aristas (E) que conectan los nodos. En un grafo no dirigido, los bordes son conjunto de dos vértices, mientras que, en un grafo dirigido, los bordes son pares ordenados de vértices.

Para representar visualmente los grafos, se dibuja cada vértice como un punto en el plano, generalmente en forma de círculo o punto, y cada borde se representa como una curva o segmento de línea recta que conecta dos vértices. Si un grafo no tiene bordes que se crucen en su representación, se dice que es un grafo planar o tiene una incrustación plana. Es importante tener en cuenta que un grafo puede tener múltiples representaciones gráficas.

Los algoritmos de búsqueda de trayectorias se pueden utilizar para planificar el camino entre dos puntos en un mapa, representado mediante un grafo. En este caso, los nodos se representan como puntos de referencia o una malla de navegación, y las aristas se utilizan para conectar estos nodos. El algoritmo navegará a través de estos nodos evitando obstáculos y buscando la mejor trayectoria para llegar a su destino.

2.1.2 Big O

La notación Big O se utiliza para describir el comportamiento asintótico de una función, es decir, cómo se comporta una función cuando su argumento tiende a infinito. En el contexto de análisis de algoritmos, Big O se usa para caracterizar el tiempo de ejecución o el espacio de memoria de un algoritmo en función del tamaño de la entrada.

Definición Formal:

Una función *f(n)* se dice que es O(g(n)) (donde g(n) es una función que describe una cota superior asintótica) si y solo si existen constantes positivas *C* y n_0 tales que:

$$f(n) \leq C * g(n)$$

$$para\ todo\ n \geq n0$$

Aquí:

- *f(n)* es la función que describe el tiempo de ejecución o el uso de memoria del algoritmo.
- *g(n)* es una función que proporciona una cota superior asintótica para *f(n)*.
- *C* es una constante que representa el factor multiplicativo.
- n_0 es un umbral a partir del cual la desigualdad se mantiene.

Explicación:

- **Constantes C y n_0:** Estas constantes aseguran que la función *f(n)* no exceda a la función *g(n)* multiplicada por una constante *C* a partir de un cierto punto n_0. La elección de n_0 asegura que la cota superior sea válida para valores suficientemente grandes de *n*.
- **Función *g(n)*:** La función *g(n)* es elegida para ser lo más simple posible y aun así proporcionar una cota superior adecuada para *f(n)*. En la práctica, elegimos *g(n)* para que refleje el crecimiento asintótico del algoritmo de manera precisa.

Ejemplo:

Supongamos que se tiene un algoritmo cuya función de tiempo de ejecución es

$$f(n) = 3n^2 + 2n + 1$$

Queremos demostrar que

$$f(n) = O(n^2)$$

- **Elección de *g(n)*:** En este caso, elegimos

$$g(n) = n^2.$$

- **Constante *C* y umbral n_0:** Podemos encontrar $C = 6\ y\ n_0 = 1,\ para\ n \geq\ 1$:

$$3n^2 + 2n + 1\ \leq 6n^2$$

- Así que $3n^2 + 2n + 1$ *es* $O(n^2)$ con estas constantes.

La notación Big O es una herramienta fundamental para el análisis de algoritmos, ya que proporciona una forma de expresar cómo el tiempo de ejecución o el uso de memoria crece en función del tamaño de la entrada. La clave está en encontrar una función *g(n)* que actúe como una cota superior para *f(n)*, asegurando que la relación $f(n) \leq C * g(n)$ se mantenga para valores suficientemente grandes de *n*.

Aquí hay algunos ejemplos de notaciones Big O:

• O(1)- tiempo constante, ejemplo determinar si un número es par

• O(logn)- tiempo logarítmico, ejemplo agregar un elemento a un árbol binario

• O(n)- tiempo lineal, ejemplo encontrar un elemento en una matriz sin clasificar

• O(nlogn)- tiempo asintótico, ejemplo Complejidad teórica del algoritmo de Dijkstra.

• $O(n^2)$- tiempo cuadrático, ejemplo multiplicar dos números de n dígitos a mano

• $O(2^n)$- tiempo exponencial, ejemplo calcular recursivamente el enésimo número de Fibonacci.

• O(n!)- tiempo factorial, ejemplo. generando todas las permutaciones de una cadena.

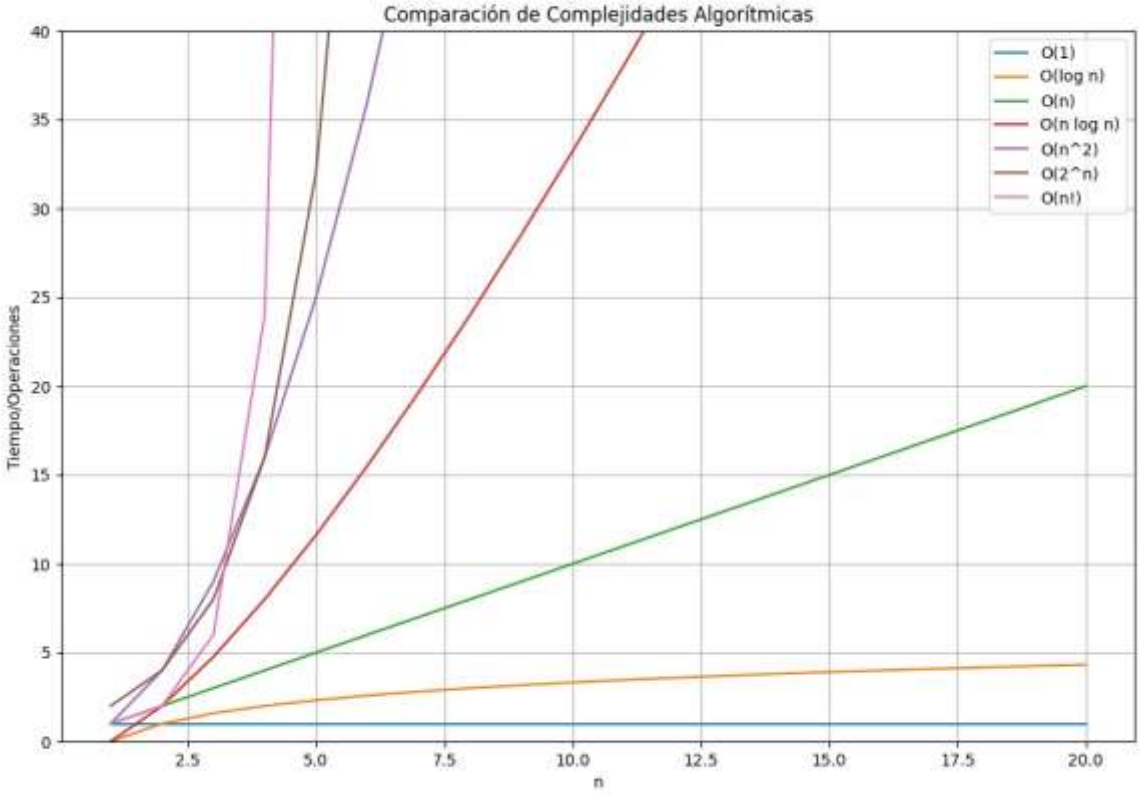

Figura 2.1 Diferentes complejidades del tiempo de ejecución sobre el tamaño de entrada n

2.1.2 Técnicas Heurísticas

Las técnicas heurísticas tienen como objetivo encontrar soluciones a problemas específicos, como la búsqueda de trayectorias, sin garantizar una eficiencia de los recursos utilizados.

La heurística es un enfoque general utilizado en la resolución de problemas que puede carecer de exactitud, a la vez que no garantiza convergencia. Según Newell y Simon, "Teniendo los elementos de una teoría de resolución de problemas heurística, se puede usar esta teoría tanto para comprender los procesos heurísticos humanos como para simular tales procesos con computadoras digitales" [16].

En este trabajo se utilizan cuatro importantes algoritmos para resolver problemas de trayectorias: de Búsqueda en anchura (BFS, Breadth-first search), Búsqueda en profundidad (DFS, Depth-first search), Dijkstra y el Algoritmo A*, que se basan en técnicas heurísticas. Este último es ampliamente utilizado en la resolución de problemas de búsqueda de trayectorias debido a su simplicidad.

La notación Big O para cada algoritmo es:

- DFS: O(n), visita los nodos una sola vez sin cálculos adicionales.
- BFS: O(n), explora por niveles sin considerar pesos.
- Dijkstra: O (n log n), utiliza una cola de prioridad y actualiza las distancias mínimas.
- A*: $O(2^n)$, emplea una estimación del costo restante y puede evitar nodos innecesarios.

2.1.2.1 BFS

Breadth First Search o algoritmo de búsqueda en anchura es un algoritmo de búsqueda utilizado para recorrer y buscar en un grafo o árbol.

El método BFS comienza en un nodo inicial y recorre todos los nodos adyacentes a ese nodo. Luego, para cada uno de esos nodos adyacentes, busca y recorre todos sus nodos adyacentes que aún no han sido visitados. Este proceso se repite hasta que se han visitado todos los nodos del grafo.

Para implementar el método BFS, se utiliza una estructura de datos llamada cola. La cola es una lista de nodos que aún no han sido visitados, y en la cual se añaden los nodos adyacentes a medida que se van visitando los nodos actuales.

El método BFS recorre los nodos del grafo en orden de su distancia al nodo inicial, de manera que los nodos más cercanos son visitados antes que los más lejanos. Esto lo hace útil para problemas en los que se busca encontrar el camino más corto o la distancia mínima desde un nodo a todos los demás nodos en el grafo.

En BFS, desde el nodo ***a*** se visita a cada uno de sus vecinos, antes de visitar a cualquier otro nodo. Esto se puede pensar como buscar nivel por nivel desde ***a***. Una solución iterativa que involucra una cola, generalmente, funciona mejor [17]. A continuación, se presenta el grafo del funcionamiento del algoritmo BFS (Algoritmo 1).

Algoritmo 1: Algoritmo BFS

```
1. Función BFS (Grafo, Inicio, Fin):
2.     # Obtener el número de filas y columnas de la malla.
3       filas = len(malla)
4       columnas = len(malla)
5
6       # Definir los movimientos posibles:
7       # derecha , izquierda, abajo y  arriba.
8       movimientos = [ (0, 1), (0, -1), (1, 0), (-1, 0)]
9
10      # Inicializar estructuras
11      # Crear una lista llamada "cola" y poner ahí el punto de inicio.
12      cola = {[inicio]}
13      # Crear un conjunto llamado "visitados" y agregar el punto de inicio.
14      visitados = set([inicio])
15
16      # Crear un diccionario llamado "padres" para guardar el inicio.
17      # El inicio no tiene padre.
18      padres = {inicio: None}
19
20      # Crear un diccionario llamado "distancia" para contar cuántos pasos desde el inicio.
21      distancia = {inicio: 0}
22      # Mientras haya lugares en la cola por visitar:
23      while cola:
24
25              # Se extrae el primer lugar de la cola y llamarlo "actual".
26              actual = cola.popleft()
27
28              # Si el lugar actual es el destino final:
29              if actual == fin:
30                      # Salir del ciclo (Se encontró el camino).
31                      break
32
33              # Obtener las coordenadas del lugar actual (x, y)
34              x, y = actual
35
36              # Para cada dirección posible (arriba, abajo, izquierda, derecha):
37              for dx, dy in movimientos:
38
39              #Calcular la posición del vecino sumando el movimiento a la posición actual.
40                      vecino = (x + dx, y + dy)
41                      vx, vy = vecino
42                      # Verificar que esté dentro de la malla y no visitado
43                      if 0 <= vx < filas and 0 <= vy < columnas and vecino not in visitados:
```

```
44                          # Marcar el vecino como visitado.
45                          visitados.add(vecino)
46                          # Guardar que su padre es el lugar actual.
47                          padres[vecino] = actual
48                          # Calcular la distancia desde el inicio sumando 1 a la del padre.
49                          distancia[vecino] = distancia[actual] + 1
50                          # Agregar el vecino a la cola para visitarlo después.
51                          cola.append(vecino)
52      return padres, distancia
53 Fin Funcion
```

Algoritmo 2: Reconstruir Camino

```
1.  Función Reconstruir_Camino(padres, inicio, objetivo):
2.# Crear lista CAMINO vacía
3.camino = []
4. # Nodo = OBJETIVO
5. nodo = fin
6.
7. # Mientras Nodo no sea None
8. while nodo is not None:
9.      #Agregar Nodo a CAMINO
10.     camino.append(nodo)
11.
12.     #Nodo = PADRES[Nodo]
13.     nodo = padres.get(nodo)
14.
15.     Invertir CAMINO
16.     camino.reverse()
17.
18.     Si CAMINO[0] es igual a INICIO
19.     if camino[0] == inicio:
20.             #Devolver CAMINO
21.             return camino
22. Fin Función
```

Ejemplo:

Se tiene el siguiente escenario, al manejar arreglos la numeración de los nodos será a partir del punto (0,0), se parte de un punto inicio (1,2) y se desea encontrar el punto fin (7,6), para esta solución se utilizará el algoritmo BFS, en la figura se muestran las diferentes representaciones del escenario. A la izquierda se muestra una malla, en la parte de en medio se muestra el grafo compuesto por nodos y a la derecha, un grafo compuesto por nodos en árbol.

Este escenario será utilizado en los cuatro algoritmos para apreciar el comportamiento de cada uno ante el mismo escenario.

Figura 2.2 Referencia de colores

Nodo inicial: (2,1)
Objetivo: (6,7)

Paso 1 Inicio:

Nodo actual: (2, 1)

Cola: []

Se añade a Padres

Se extrae de la cola (2, 1) y se agrega a la cola sus vecinos: (3, 1), (1, 1), (2, 2), (2, 0)

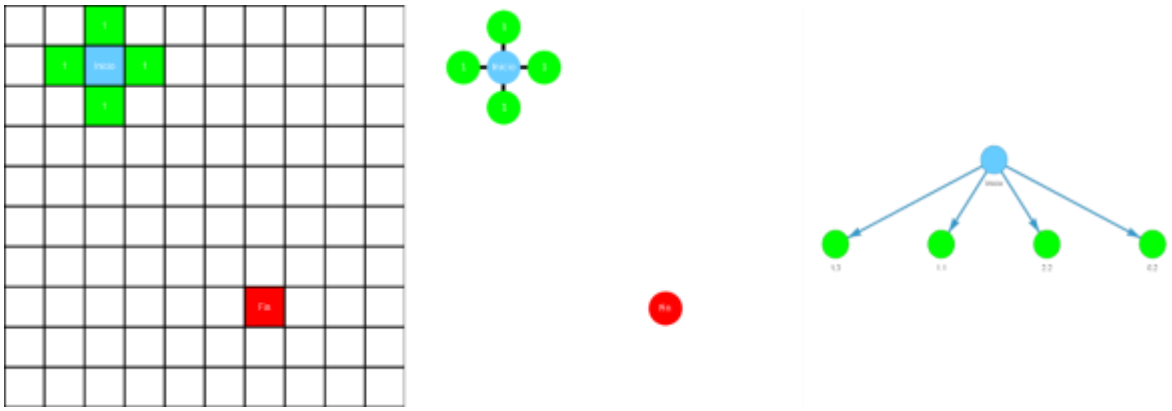

Figura 2.3 Paso 1 - BFS

Paso 2 - Nodo actual: (3, 1)

Cola: [(1, 1), (2, 2), (2, 0)]

Se agrega a Visitados

Se añade a Padres

Se extrae de la cola (3, 1) y se agrega a la cola sus vecinos: (4, 1), (3, 2), (3, 0)

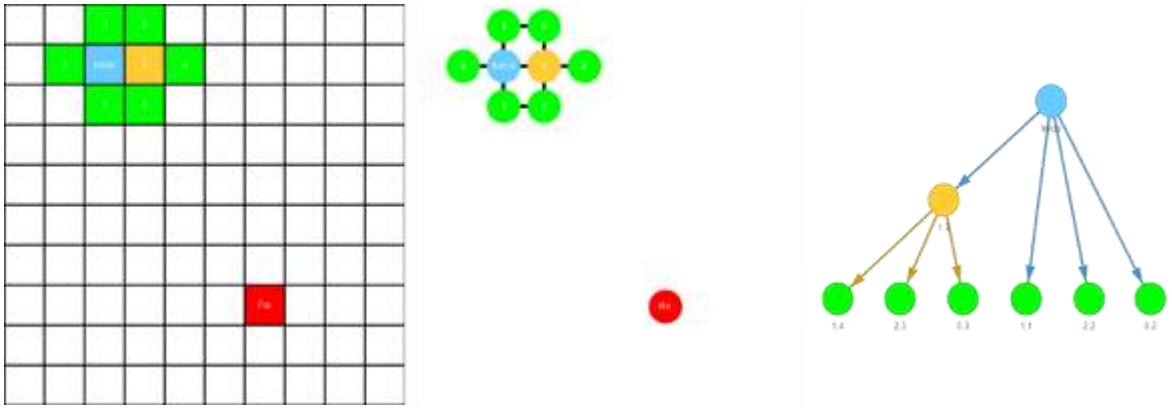

Figura 2.4 Paso 2 -BFS

Paso 3 - Nodo actual: (1, 1)

Cola: [(2, 2), (2, 0), (4, 1), (3, 2), (3, 0)]

Se agrega a Visitados

Se añade a Padres

Se extrae de la cola (1, 1) y se agrega a la cola sus vecinos: (0, 1), (1, 2), (1, 0)

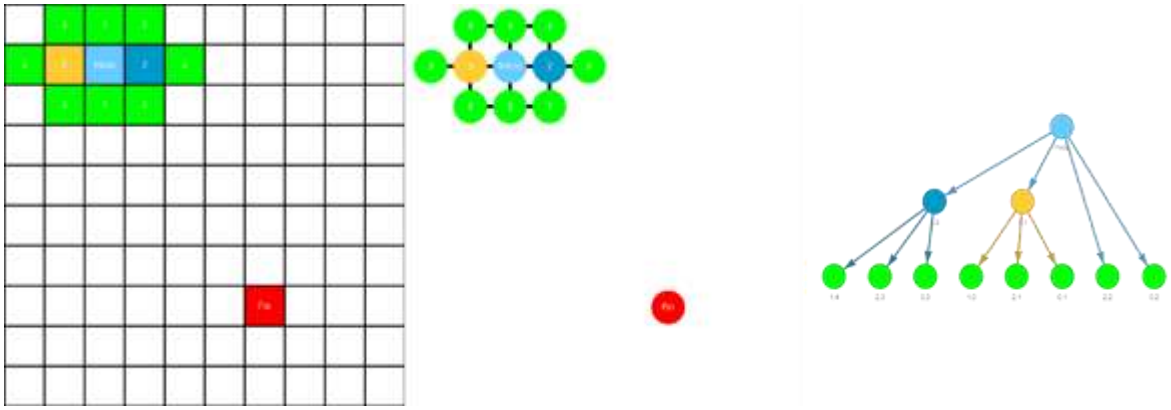

Figura 2.5 Paso 3 -BFS

Paso 4 - Nodo actual: (2, 2)

Cola: [(2, 0), (4, 1), (3, 2), (3, 0), (0, 1), (1, 2), (1, 0)]

Se agrega a Visitados

Se añade a Padres

Se extrae de la cola (2, 2) y se agrega a la cola sus vecinos: (2, 3)

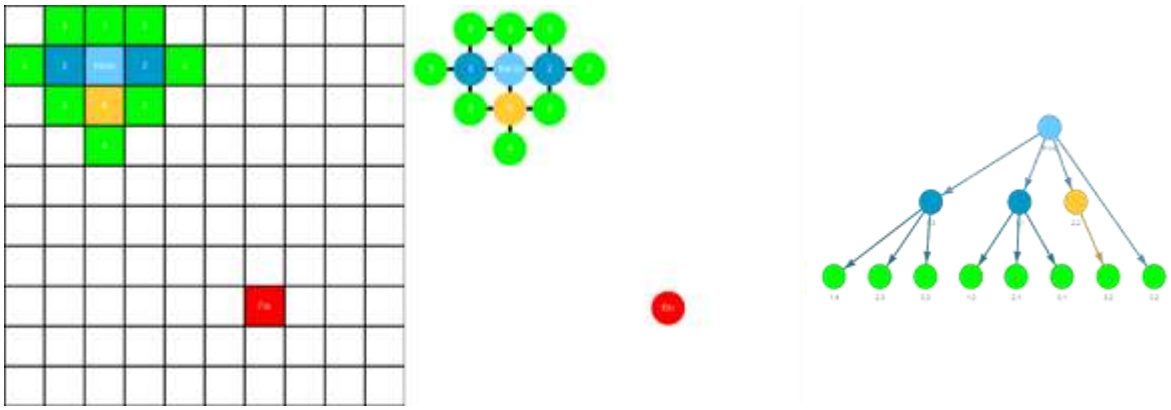

Figura 2.6 Paso 4 -BFS

Paso 5 - Nodo actual: (2, 0)

Cola: [(4, 1), (3, 2), (3, 0), (0, 1), (1, 2), (1, 0), (2, 3)]

Se agrega a Visitados

Se extrae de la cola (2, 0) y no se agregan vecinos nuevos.

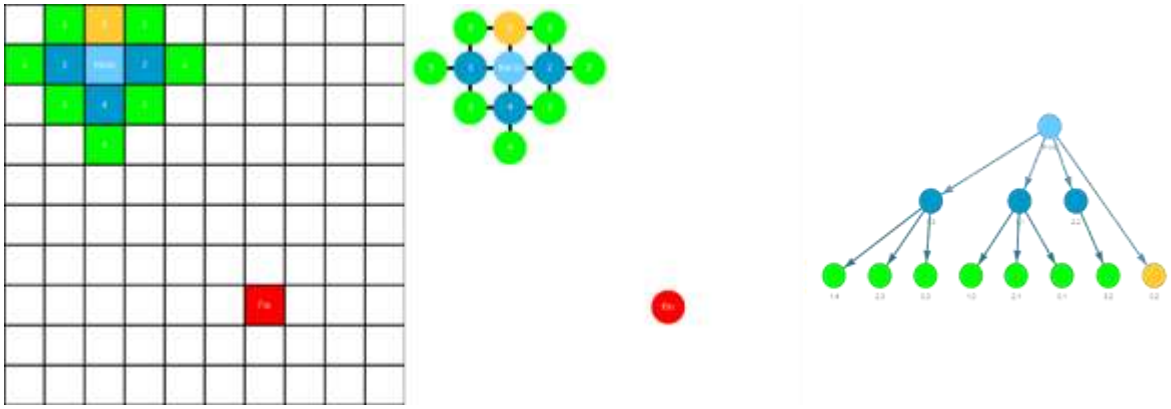

Figura 2.7 Paso 5 -BFS

Paso 6 - Nodo actual: (4, 1)

Cola: [(3, 2), (3, 0), (0, 1), (1, 2), (1, 0), (2, 3)]

Se agrega a Visitados

Se añade a Padres

Se extrae de la cola (4, 1) y se agrega a la cola sus vecinos: (5, 1), (4, 2), (4, 0)

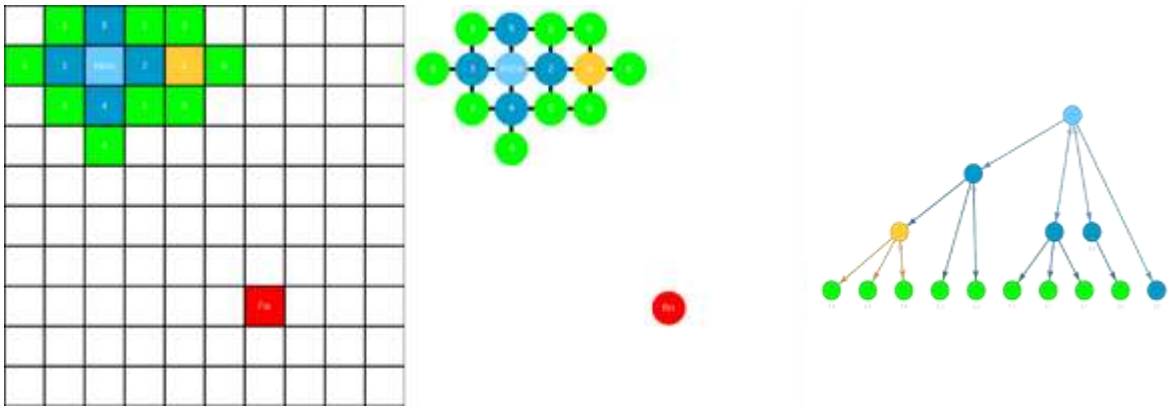

Figura 2.1 Paso 6 -BFS

Paso 7 - Nodo actual: (3, 2)

Cola: [(3, 0), (0, 1), (1, 2), (1, 0), (2, 3), (5, 1), (4, 2), (4, 0)]

Se agrega a Visitados

Se añade a Padres

Se extrae de la cola (3, 2) y se agrega a la cola sus vecinos: (3, 3)

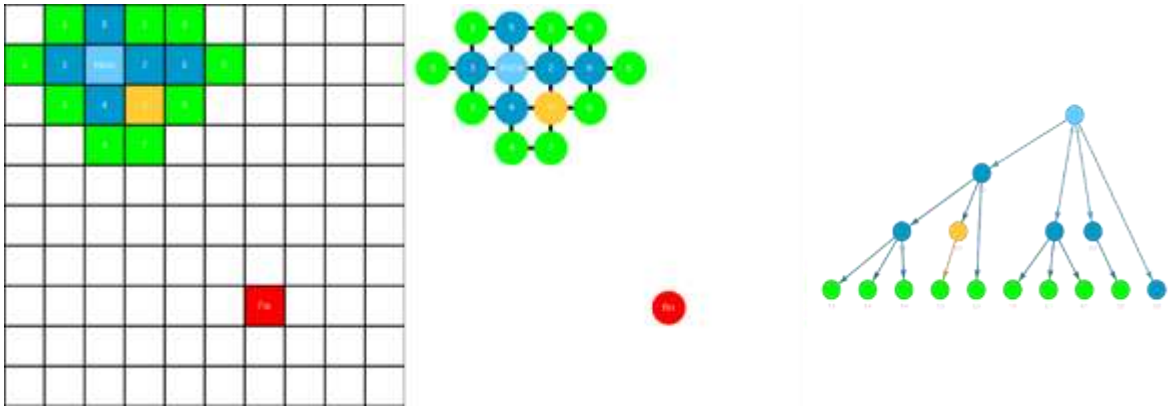

Figura 2.2.2 Paso 7 -BFS

Paso 8 - Nodo actual: (3, 0)

Cola: [(0, 1), (1, 2), (1, 0), (2, 3), (5, 1), (4, 2), (4, 0), (3, 3)]

Se agrega a Visitados

Se extrae de la cola (3, 0) y no se agregan vecinos nuevos.

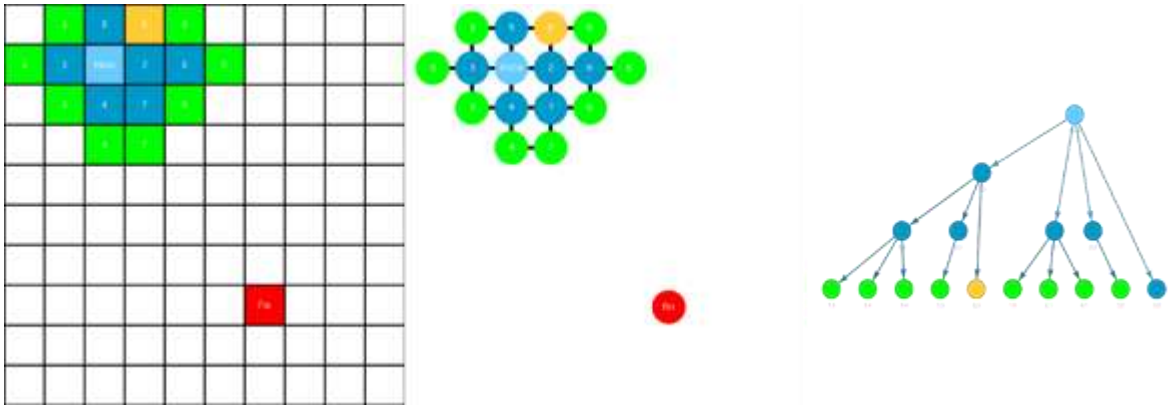

Figura 2.3 Paso 8 -BFS

Paso 9 - Nodo actual: (0, 1)

Cola: [(1, 2), (1, 0), (2, 3), (5, 1), (4, 2), (4, 0), (3, 3)]

Se agrega a Visitados

Se añade a Padres

Se extrae de la cola (0, 1) y se agrega a la cola sus vecinos: (0, 2), (0, 0)

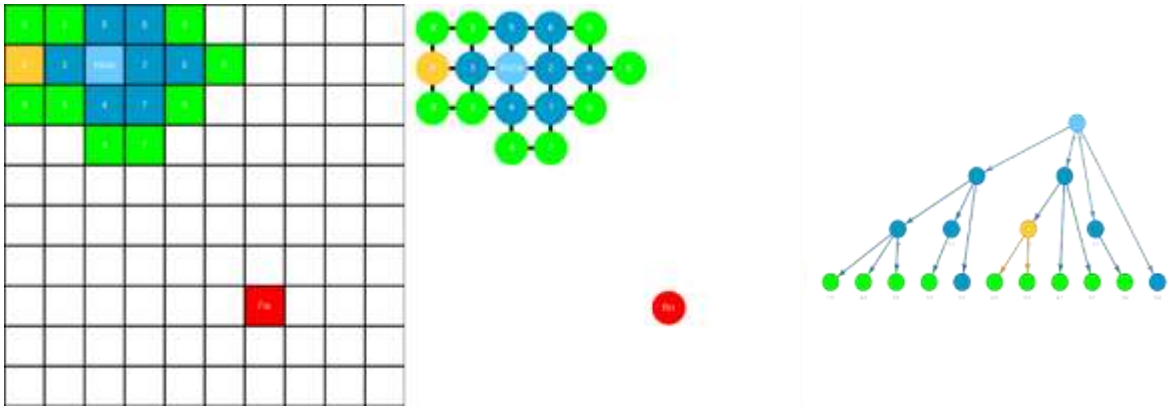

Figura 2.4 Paso 9 -BFS

Paso 10 - Nodo actual: (1, 2)

Cola: [(1, 0), (2, 3), (5, 1), (4, 2), (4, 0), (3, 3), (0, 2), (0, 0)]

Se agrega a Visitados

Se añade a Padres

Se extrae de la cola (1, 2) y se agrega a la cola sus vecinos: (1, 3)

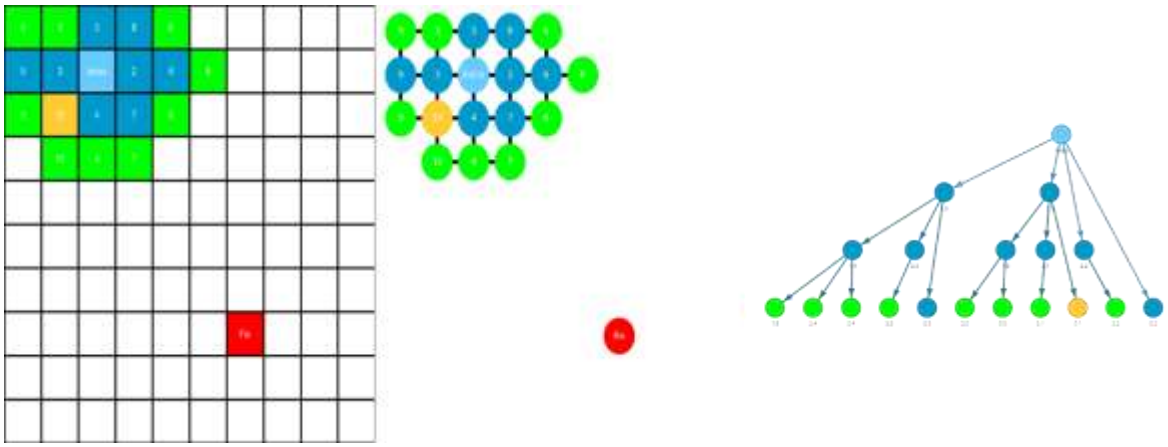

Figura 2.5 Paso 10 -BFS

Paso 20 - Nodo actual: (2, 4)

Cola: [(6, 1), (5, 2), (5, 0), (4, 3), (3, 4), (0, 3), (1, 4)]

Se agrega a Visitados

Se añade a Padres

Se extrae de la cola (2, 4) y se agrega a la cola sus vecinos: (2, 5)

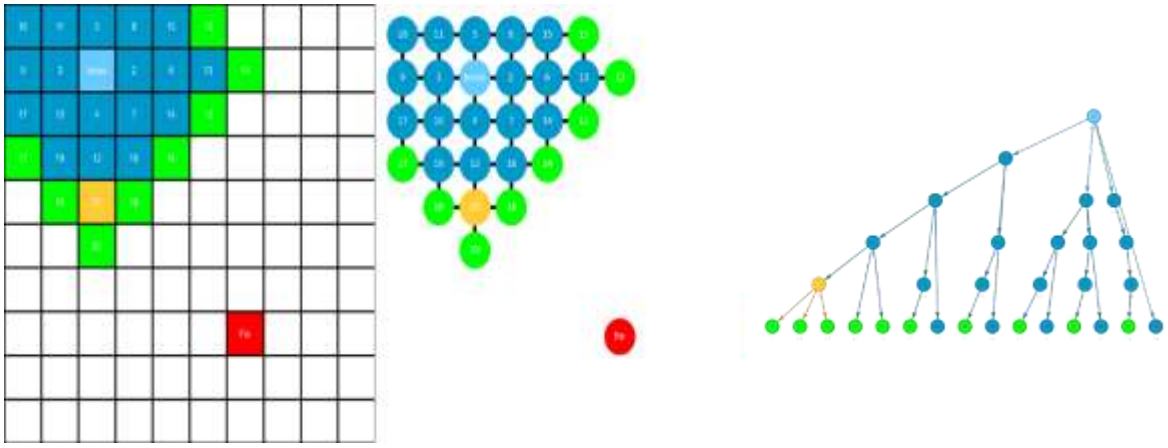

Figura 2.6 Paso 20 -BFS

Paso 30 - Nodo actual: (6, 2)

Cola: [(6, 0), (5, 3), (4, 4), (3, 5), (0, 4), (1, 5), (2, 6), (8, 1), (7, 2), (7, 0)]

Se agrega a Visitados

Se añade a Padres

Se extrae de la cola (6, 2) y se agrega a la cola sus vecinos: (6, 3)

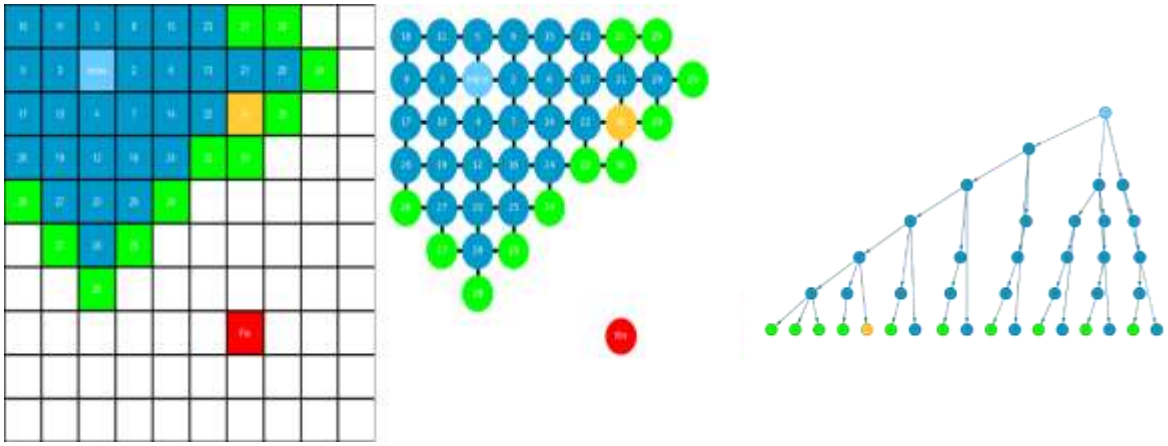

Figura 2.7 Paso 30 -BFS

Paso 40 - Nodo actual: (7, 0)

Cola: [(6, 3), (5, 4), (4, 5), (3, 6), (0, 5), (1, 6), (2, 7), (9, 1), (8, 2), (8, 0), (7, 3)]

Se agrega a Visitados

Se extrae de la cola (7, 0) y no se agregan vecinos nuevos.

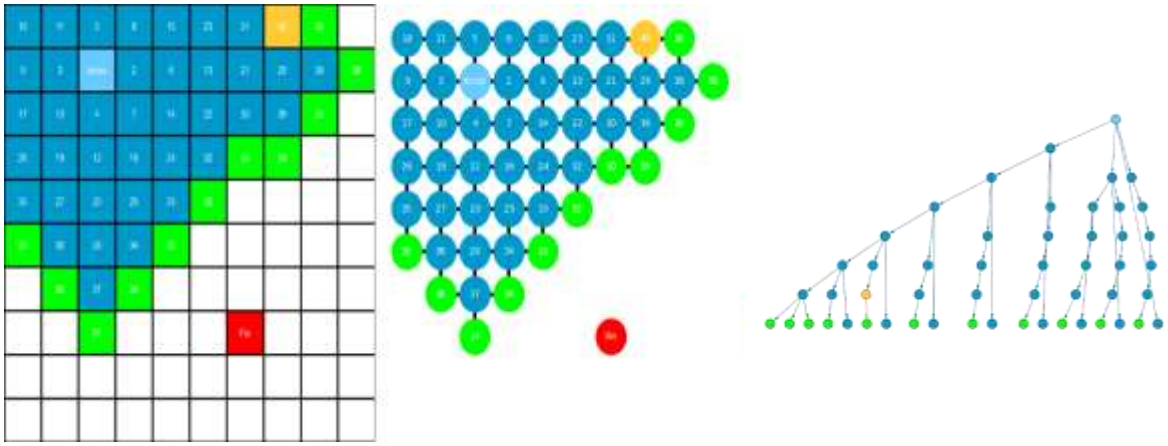

Figura 2.8 Paso 40 -BFS

Paso 50 - Nodo actual: (8, 0)

Cola: [(7, 3), (6, 4), (5, 5), (4, 6), (3, 7), (0, 6), (1, 7), (2, 8), (9, 2), (9, 0), (8, 3)]

Se agrega a Visitados

Se extrae de la cola (8, 0) y no se agregan vecinos nuevos.

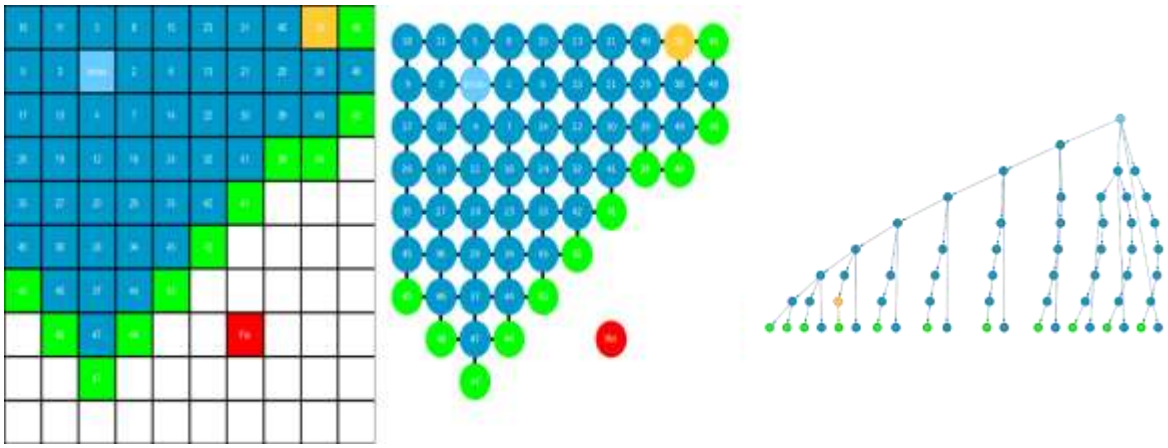

Figura 2.9 Paso 50 -BFS

Paso 60 - Nodo actual: (9, 0)

Cola: [(8, 3), (7, 4), (6, 5), (5, 6), (4, 7), (3, 8), (0, 7), (1, 8), (2, 9), (9, 3)]

Se agrega a Visitados

Se extrae de la cola (9, 0) y no se agregan vecinos nuevos.

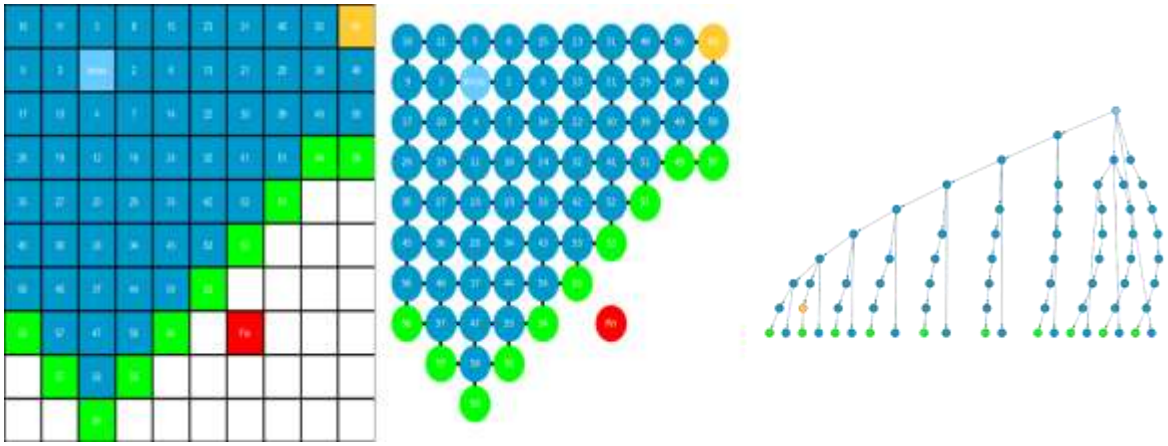

Figura 2.10 Paso 60 -BFS

Paso 70 - Nodo actual: (9, 3)

Cola: [(8, 4), (7, 5), (6, 6), (5, 7), (4, 8), (3, 9), (0, 8), (1, 9)]

Se agrega a Visitados

Se añade a Padres

Se extrae de la cola (9, 3) y se agrega a la cola sus vecinos: (9, 4)

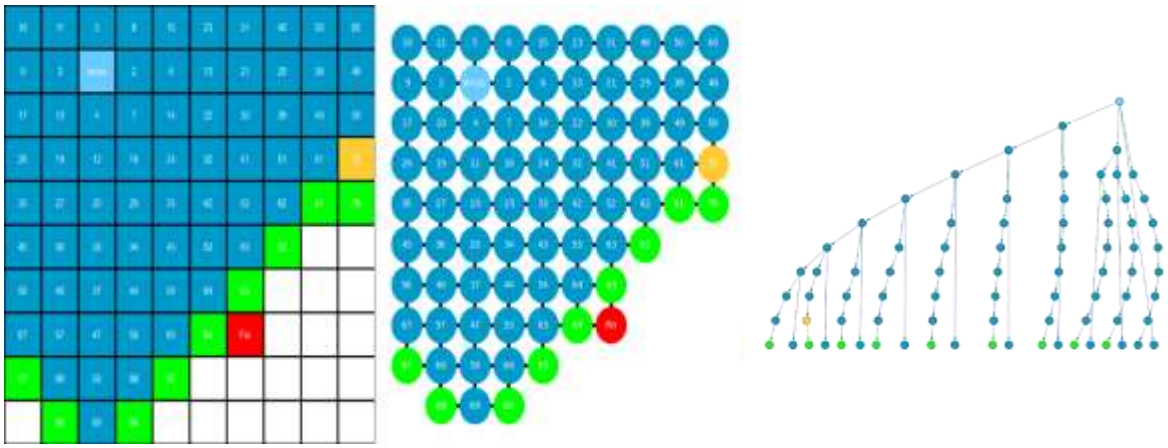

Figura 2.11 Paso 70 -BFS

Paso 80 - Nodo actual: (8, 5)

Cola: [(7, 6), (6, 7), (5, 8), (4, 9), (0, 9), (9, 5)]

Se agrega a Visitados

Se añade a Padres

Se extrae de la cola (8, 5) y se agrega a la cola sus vecinos: (8, 6)

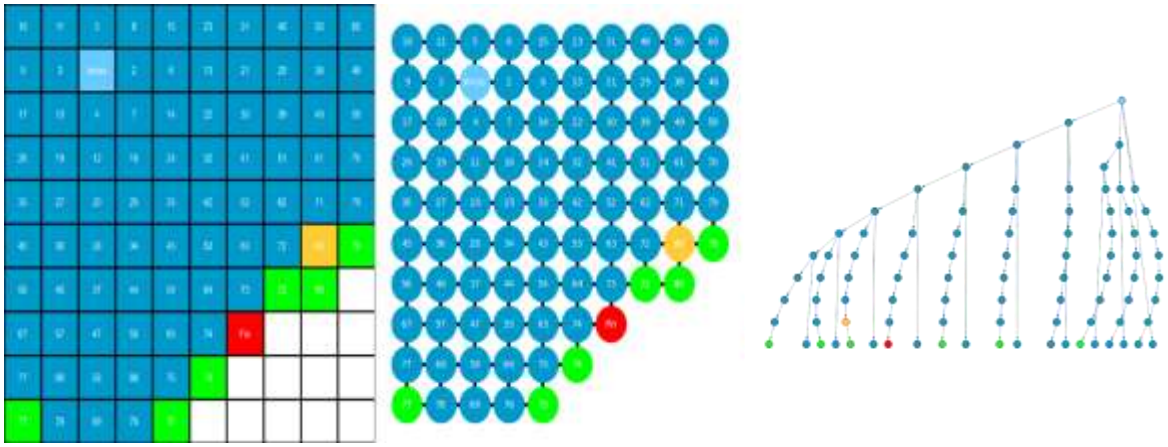

Figura 2.12 Paso 80 -BFS

Paso 82 - Nodo actual: (6, 7)

Cola: [(5, 8), (4, 9), (0, 9), (9, 5), (8, 6), (7, 7)]

Visitados: {(4, 0), (4, 9), (5, 1), (8, 0), (0, 5), (2, 2), (6, 2), (7, 1), (4, 2), (3, 6), (5, 3), (8, 2), (9, 1), (0, 7), (2, 4), (1, 8), (6, 4), (7, 3), (3, 8), (5, 5), (8, 4), (9, 3), (0, 0), (0, 9), (6, 6), (7, 5), (3, 1), (5, 7), (9, 5), (0, 2), (1, 3), (7, 7), (3, 3), (5, 0), (1, 5), (6, 1), (7, 0), (3, 5), (5, 2), (4, 4), (9, 0), (1, 7), (2, 6), (7, 2), (3, 7), (5, 4), (4, 6), (9, 2), (8, 6), (1, 0), (1, 9), (2, 8), (7, 4), (3, 0), (3, 9), (5, 6), (4, 8), (1, 2), (0, 4), (2, 1), (3, 2), (4, 1), (8, 1), (1, 4), (0, 6), (2, 3), (6, 3), (3, 4), (4, 3), (8, 3), (1, 6), (0, 8), (2, 5), (6, 5), (4, 5), (8, 5), (9, 4), (0, 1), (2, 7), (6, 7), (7, 6), (4, 7), (5, 8), (1, 1), (0, 3), (2, 0), (2, 9), (6, 0)}

Camino encontrado (Padres del nodo fin):

(2, 1) , (3, 1) , (4, 1) , (5, 1) , (6, 1) , (6, 2) , (6, 3) , (6, 4) , (6, 5) , (6, 6) , (6, 7)

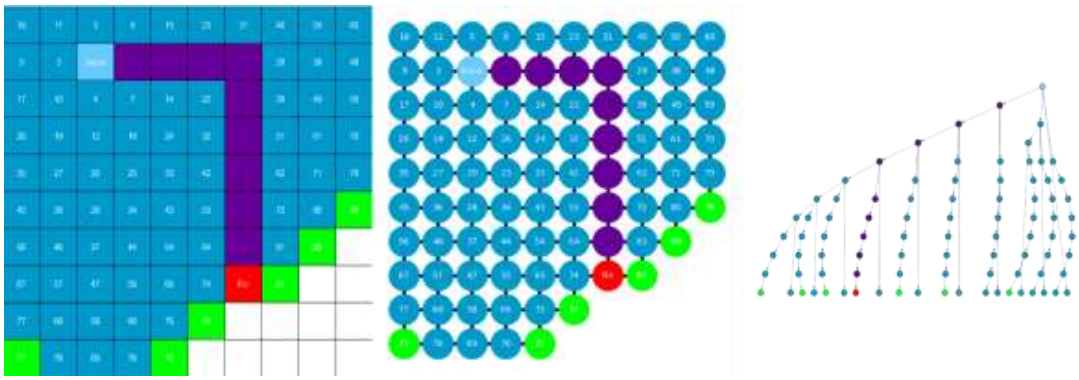

Figura 2.13 Paso 82 -BFS

2.1.2.2 DFS

Depth First Search o algoritmo de búsqueda en profundidad

A diferencia del BFS, el DFS explora primero un camino hasta su extremo antes de retroceder y continuar con otro camino.

El algoritmo DFS comienza visitando un nodo inicial y luego explora uno de sus vecinos no visitados elegido arbitrariamente. Una vez que llega a un nodo sin nodos vecinos que no han sido visitados, retrocede al nodo anterior y busca otros vecinos no visitados. Este proceso se repite hasta que se han visitado todos los nodos alcanzables desde el nodo inicial.

El DFS se puede implementar de manera recursiva o mediante el uso de una pila para guardar los nodos a visitar. En la implementación recursiva, el algoritmo utiliza la pila de llamadas del sistema para mantener el seguimiento de los nodos visitados y los nodos por visitar. En la pila se guardan los nodos por visitar y los nodos visitados.

El DFS es útil para encontrar componentes conexas en grafos no dirigidos y también para encontrar árboles de expansión en grafos dirigidos o no dirigidos. Además, se utiliza en la resolución de problemas de búsqueda de trayectorias y en la evaluación de expresiones en la construcción de compiladores.

La diferencia clave es que, al implementar este algoritmo para un grafo, se debe verificar si el nodo ha sido visitado. Si no se hace, se corre el riesgo de quedar atrapados en un bucle infinito [17].

A continuación, se presenta el grafo del funcionamiento del algoritmo DFS (Algoritmo 2).

Algoritmo 2: DFS

```
1. Funcion DFS(Grafo, Inicio, Fin):
2.     # Obtener el número de filas y columnas de la malla
3.     filas = len(malla)
4.     columnas = len(malla)
5.
6.     # Definir los movimientos posibles
7.     movimientos = [(0, 1), (0, -1), (1, 0), (-1, 0)]
8.
9.     # Inicializar estructuras
10.     # Crear una pila y poner ahí el punto de inicio
11.     pila = [inicio]
12.
13.     # Crear un conjunto llamado "visitados" y agregar el punto de inicio
14.     visitados = set([inicio])
15.
16.     # Crear un diccionario llamado "padres" para guardar el inicio
17.     padres = {inicio: None}
18.
19.     # Mientras haya lugares en la pila por visitar
20.     while pila:
21.         # Se extrae el último lugar de la pila y llamarlo "actual"
22.         actual = pila.pop()
23.
24.         # Si el lugar actual es el destino final
25.         if actual == fin:
26.             # Salir del ciclo
27.             break
28.
29.         # Obtener coordenadas del lugar actual
30.         x, y = actual
31.
32.         # Para cada dirección posible
33.         for dx, dy in movimientos:
34.             # Calcular posición del vecino
35.             vecino = (x + dx, y + dy)
36.             vx, vy = vecino
37.
38.             # Verificar que esté dentro de la malla y no visitado
39.             if 0 <= vx < filas and 0 <= vy < columnas and vecino not in visitados:
40.                 # Marcar el vecino como visitado
41.                 visitados.add(vecino)
42.                 # Guardar que su padre es el lugar actual
43.                 padres[vecino] = actual
```

```
44.             # Agregar el vecino a la pila
45.             pila.append(vecino)
46.
47.     return padres
48. Fin Función
```

Ejemplo:

Se tiene el siguiente escenario en el cual se parte de un punto inicio y se desea encontrar el punto fin, para esta solución se utilizará el algoritmo DFS, en la siguiente figura se muestra un escenario definido por una malla, el grafo compuesto por nodos y un grafo compuesto por nodos en árbol.

Paso 1 - Nodo actual: (2, 1)

Pila: []

Se extrae de la cola (2, 1) y se agrega a la pila sus vecinos: (3, 1), (1, 1), (2, 2), (2, 0)

Se añade Padre (2, 1)

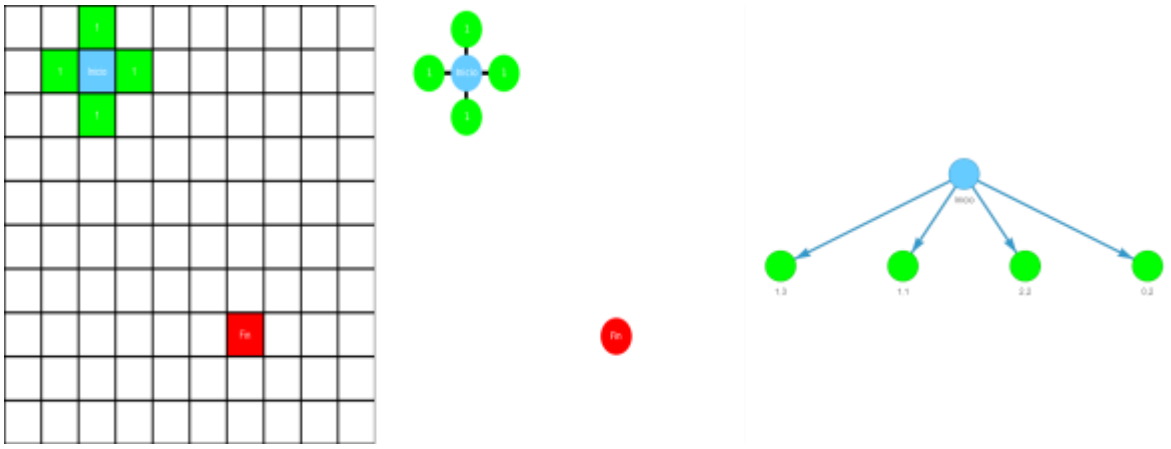

Figura 2.14 Paso 1- DFS

Paso 2 - Nodo actual: (2, 0)

Pila: [(3, 1), (1, 1), (2, 2)]

Se extrae de la cola (2, 0) y se agrega a la pila sus vecinos: (3, 0), (1, 0)

Se añade Padre (2, 0)

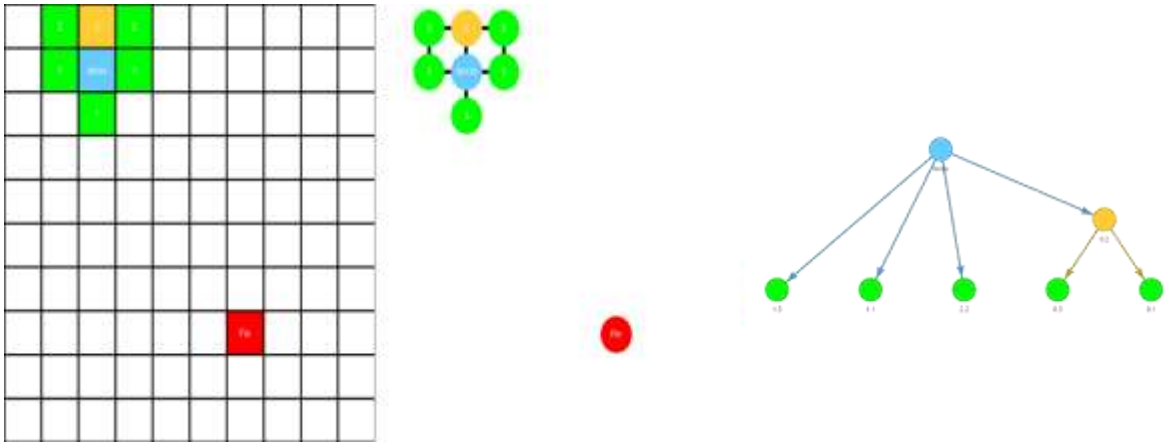

Figura 2.15 Paso 2- DFS

Paso 3 - Nodo actual: (1, 0)

Pila: [(3, 1), (1, 1), (2, 2), (3, 0)]

Se extrae de la cola (1, 0) y se agrega a la pila sus vecinos: (0, 0)

Se añade Padre (1, 0)

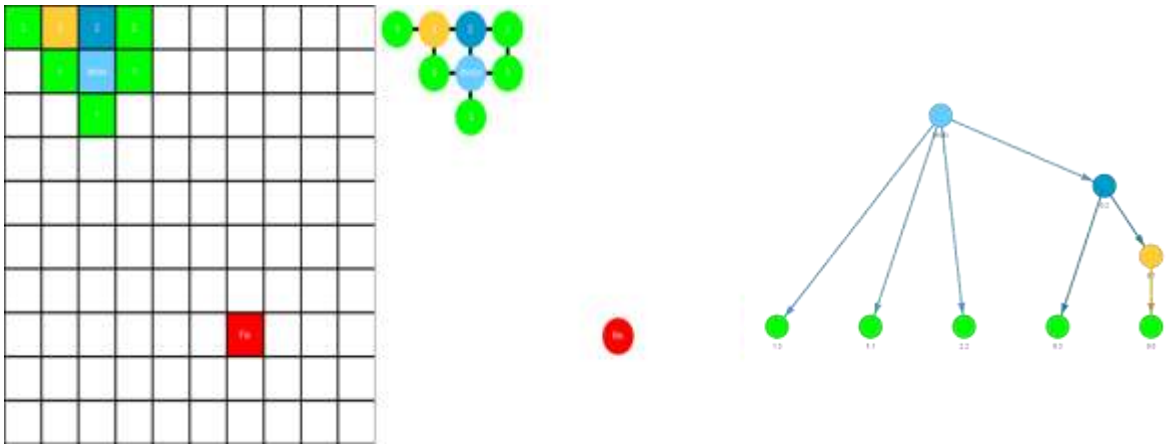

Figura 2.16 Paso 3- DFS

Paso 4 - Nodo actual: (0, 0)

Pila: [(3, 1), (1, 1), (2, 2), (3, 0)]

Se extrae de la cola (0, 0) y se agrega a la pila sus vecinos: (0, 1)

Se añade Padre (0, 0)

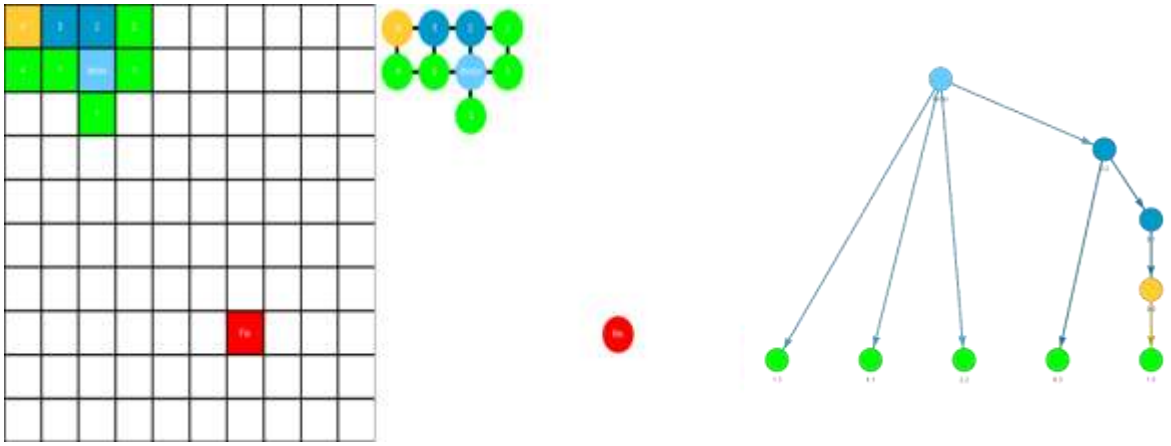

Figura 2.17 Paso 4- DFS

Paso 5 - Nodo actual: (0, 1)

Pila: [(3, 1), (1, 1), (2, 2), (3, 0)]

Se extrae de la cola (0, 1) y se agrega a la pila sus vecinos: (0, 2)

Se añade Padre (0, 1)

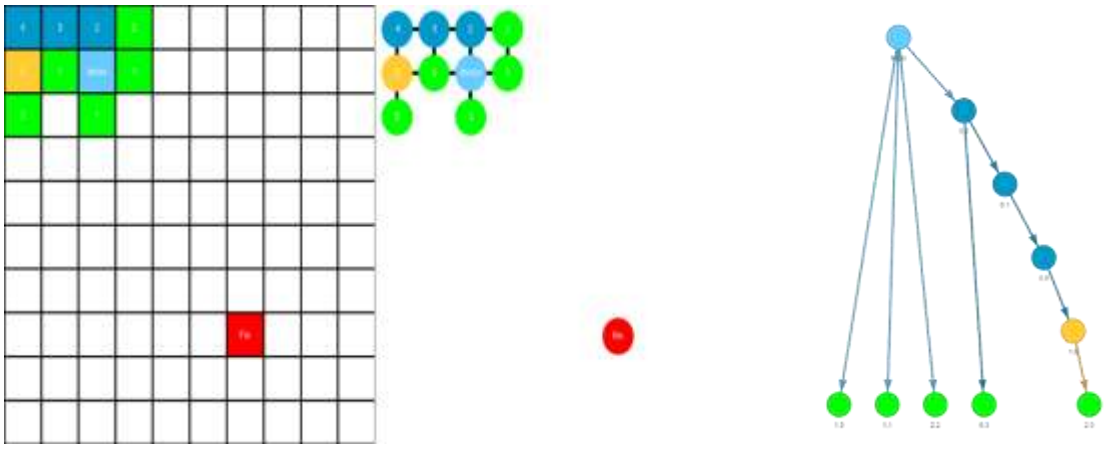

Figura 2.18 Paso 5- DFS

Paso 6 - Nodo actual: (0, 2)

Pila: [(3, 1), (1, 1), (2, 2), (3, 0)]

Se extrae de la cola (0, 2) y se agrega a la pila sus vecinos: (1, 2), (0, 3)

Se añade Padre (0, 2)

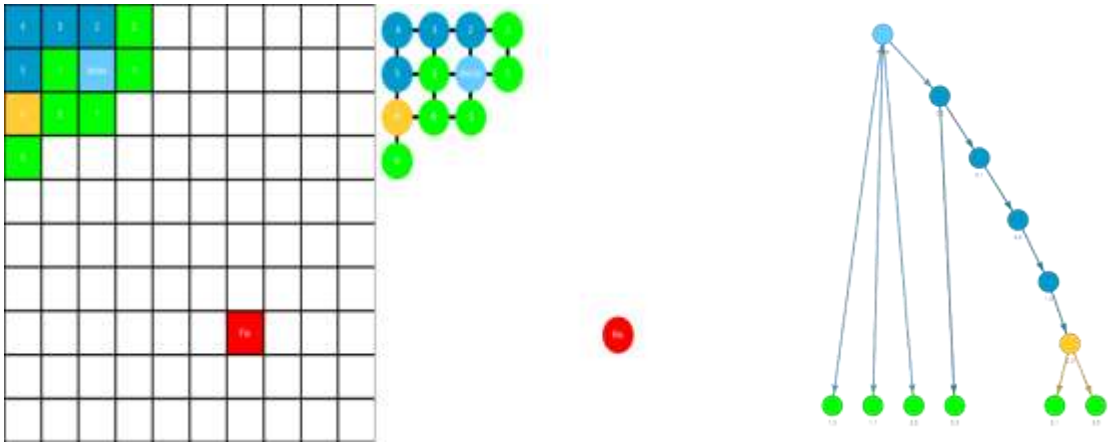

Figura 2.19 Paso 6- DFS

Paso 7 - Nodo actual: (0, 3)

Pila: [(3, 1), (1, 1), (2, 2), (3, 0), (1, 2)]

Se extrae de la cola (0, 3) y se agrega a la pila sus vecinos: (1, 3), (0, 4)

Se añade Padre (0, 3)

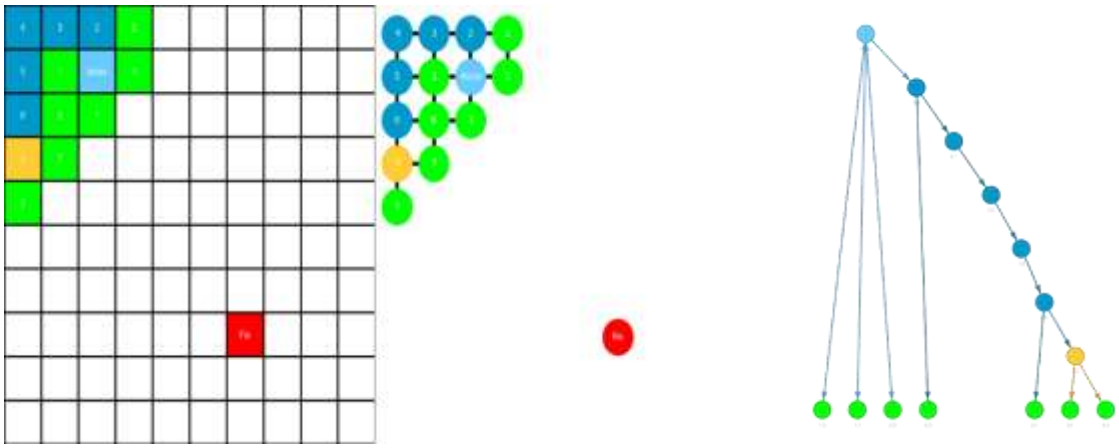

Figura 2.20 Paso 7- DFS

Paso 8 - Nodo actual: (0, 4)

Pila: [(3, 1), (1, 1), (2, 2), (3, 0), (1, 2), (1, 3)]

Se extrae de la cola (0, 4) y se agrega a la pila sus vecinos: (1, 4), (0, 5)

Se añade Padre (0, 4)

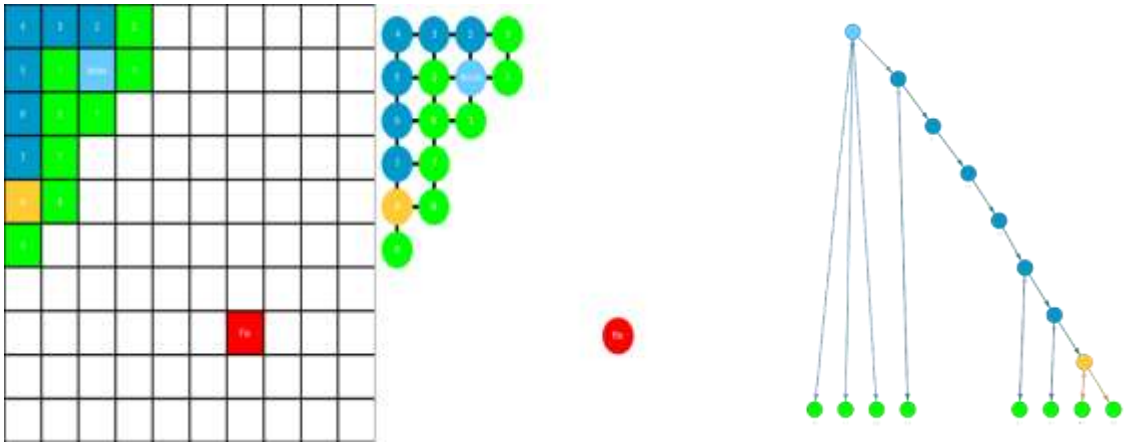

Figura 2.21 Paso 8- DFS

Paso 9 - Nodo actual: (0, 5)

Pila: [(3, 1), (1, 1), (2, 2), (3, 0), (1, 2), (1, 3), (1, 4)]

Se extrae de la cola (0, 5) y se agrega a la pila sus vecinos: (1, 5), (0, 6)

Se añade Padre (0, 5)

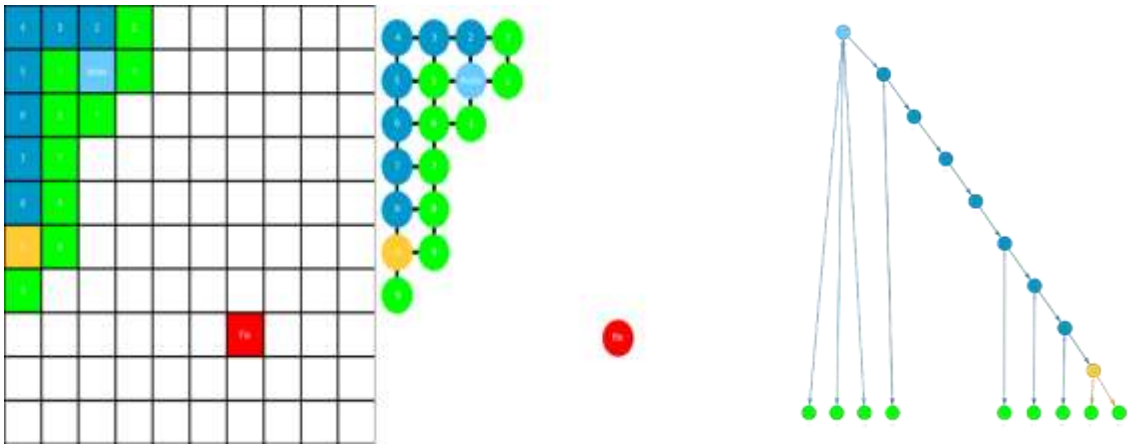

Figura 2.22 Paso 9- DFS

Paso 10 - Nodo actual: (0, 6)

Pila: [(3, 1), (1, 1), (2, 2), (3, 0), (1, 2), (1, 3), (1, 4), (1, 5)]

Se extrae de la cola (0, 6) y se agrega a la pila sus vecinos: (1, 6), (0, 7)

Se añade Padre (0, 6)

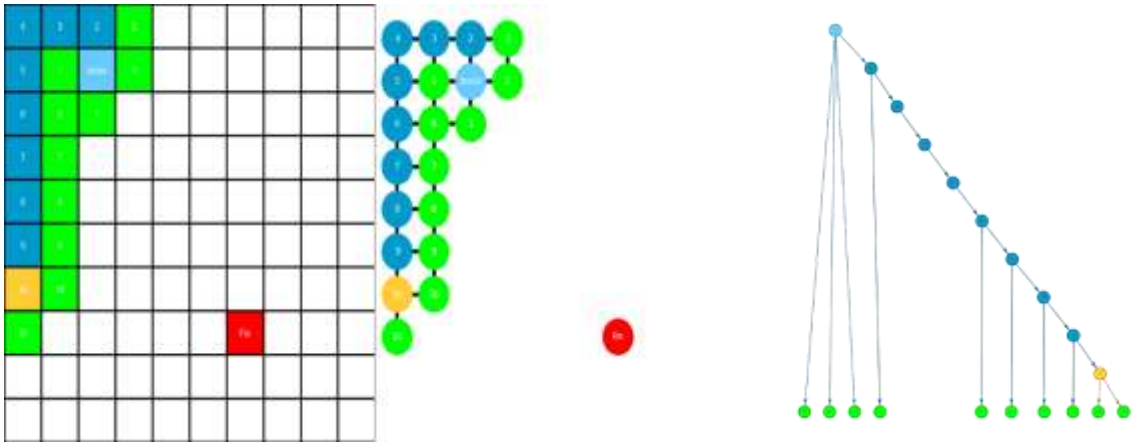

Figura 2.23 Paso 10- DFS

Paso 15 - Nodo actual: (2, 9)

Pila: [(3, 1), (1, 1), (2, 2), (3, 0), (1, 2), (1, 3), (1, 4), (1, 5), (1, 6), (1, 7), (1, 8)]

Se extrae de la cola (2, 9) y se agrega a la pila sus vecinos: (3, 9), (2, 8)

Se añade Padre (2, 9)

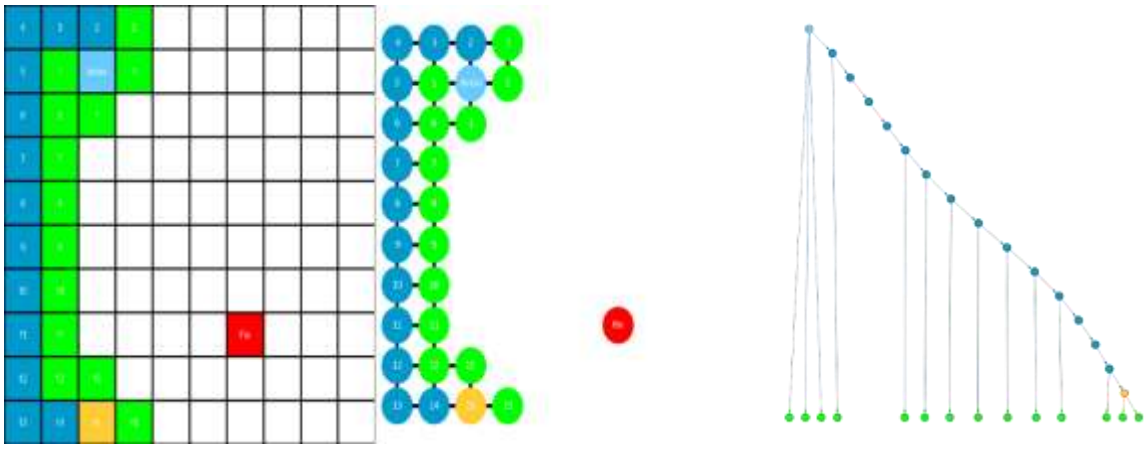

Figura 2.24 Paso 15- DFS

Paso 20 - Nodo actual: (2, 4)

Se extrae de la cola (2, 4) y se agrega a la pila sus vecinos: (3, 4), (2, 3)

Se añade Padre (2, 4)

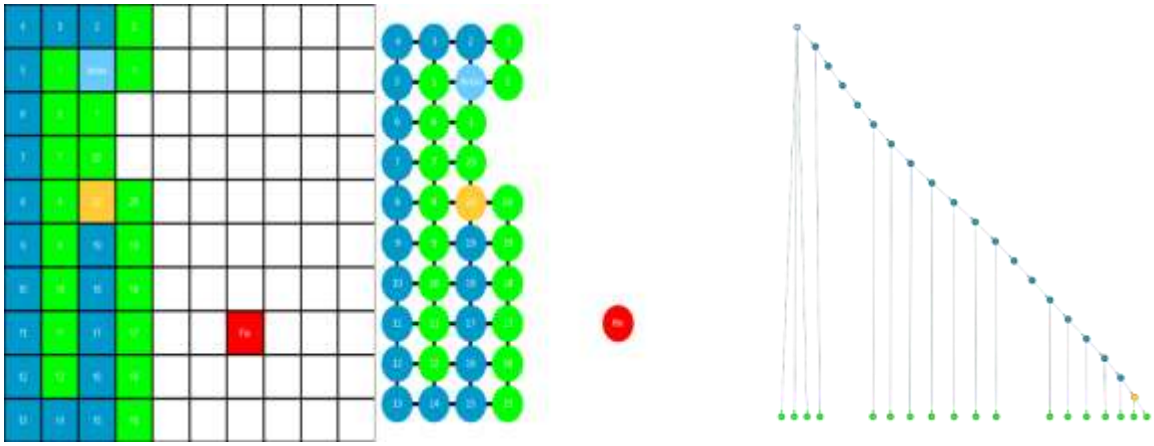

Figura 2.25 Paso 20- DFS

Paso 25 - Nodo actual: (4, 1)

Se extrae de la cola (4, 1) y se agrega a la pila sus vecinos: (5, 1), (4, 0)

Se añade Padre (4, 1)

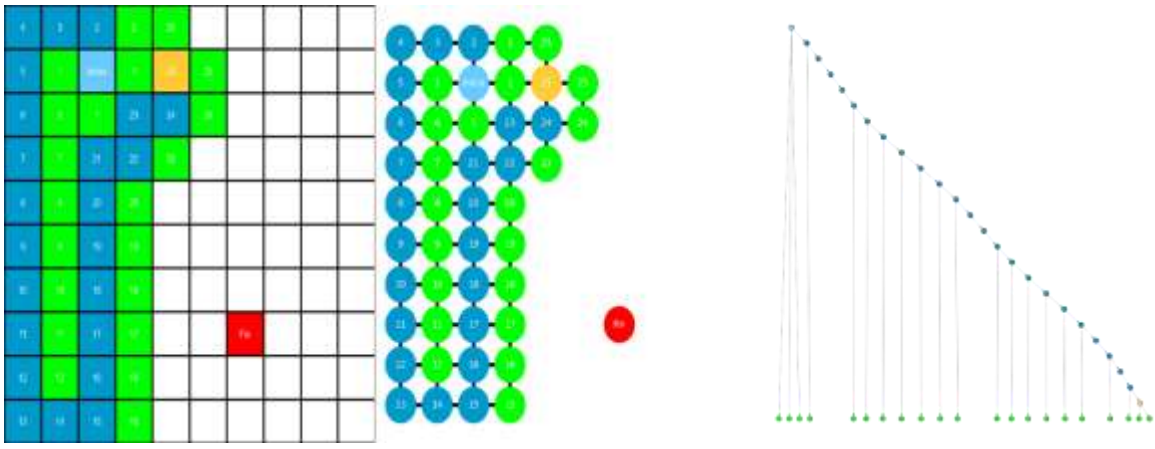

Figura 2.26 Paso 25- DFS

Paso 30 - Nodo actual: (6, 2)

Se extrae de la cola (6, 2) y se agrega a la pila sus vecinos: (7, 2), (6, 3)

Se añade Padre (6, 2)

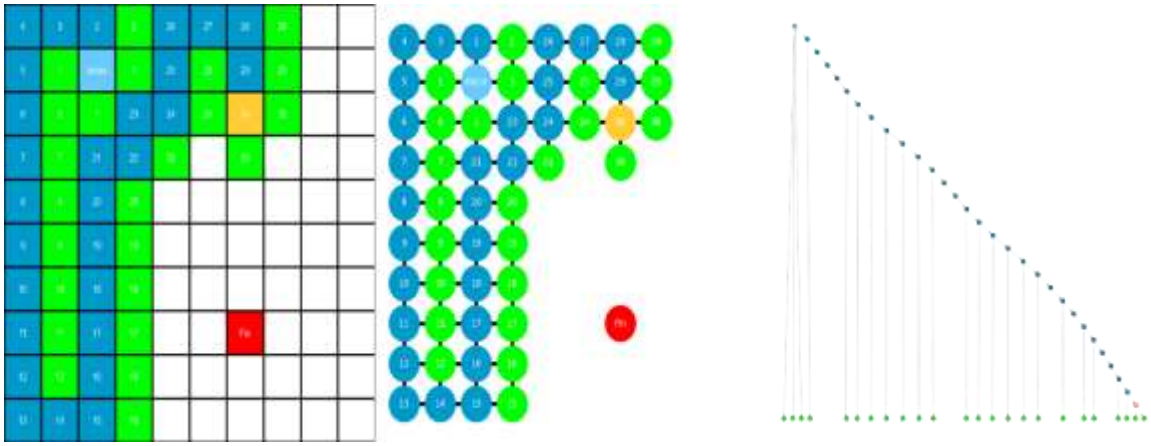

Figura 2.27 Paso 30- DFS

Paso 35 - Nodo actual: (6, 7)

Camino encontrado:

(2, 1) , (2, 0) , (1, 0) , (0, 0) , (0, 1) , (0, 2) , (0, 3) , (0, 4) , (0, 5) , (0, 6) , (0, 7) , (0, 8) , (0, 9) , (1, 9) , (2, 9) , (2, 8) , (2, 7) , (2, 6) , (2, 5) , (2, 4) , (2, 3) , (3, 3) , (3, 2) , (4, 2) , (4, 1) , (4, 0) , (5, 0) , (6, 0) , (6, 1) , (6, 2) , (6, 3) , (6, 4) , (6, 5) , (6, 6) , (6, 7)

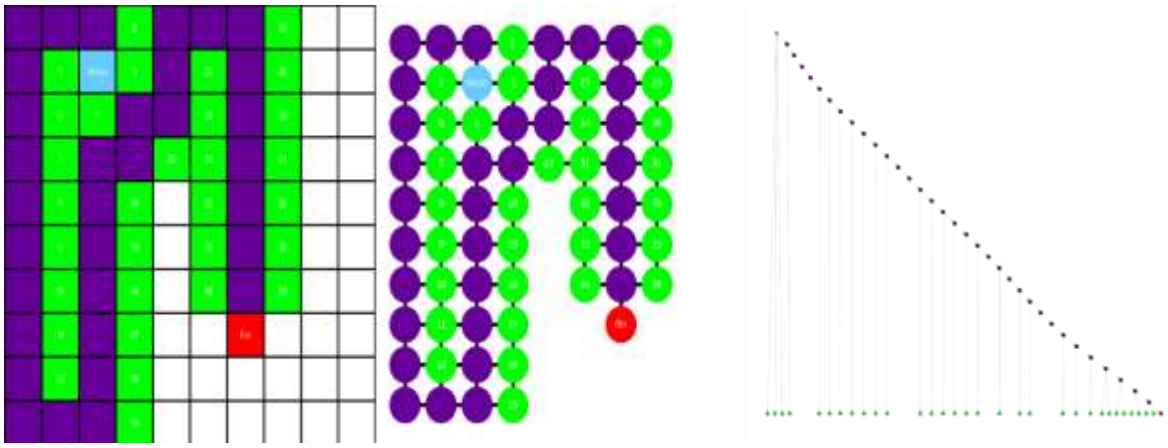

Figura 2.28 Paso 35- DFS

2.1.2.3 Algoritmo de Dijkstra

El algoritmo de Dijkstra es un algoritmo de búsqueda de camino más corto en grafos dirigidos ponderados con pesos no negativos. Fue diseñado por el científico de la computación holandés Edsger W. Dijkstra en 1956.

En el algoritmo de Dijkstra se va etiquetando cada nodo con la distancia acumulada respecto al padre, donde la distancia acumulada es la distancia mínima desde el nodo inicial y padre es el nodo predecesor en el camino mínimo que une el nodo inicial con el que se está calculando [21].

A continuación, se detalla el algoritmo paso a paso:

Inicialización: Se establece un conjunto de nodos cuyas distancias mínimas desde el nodo inicial se han determinado de manera definitiva. Al comienzo, este conjunto solo incluye el nodo inicial y su distancia mínima desde sí mismo se establece en 0. Los demás nodos se agregan a un conjunto de nodos no visitados con distancias mínimas tentativas establecidas en infinito.

Selección del nodo más cercano: Se selecciona el nodo no visitado con la distancia mínima más baja desde el nodo inicial. En la primera iteración, este será el nodo inicial.

Actualización de las distancias mínimas: Para cada nodo adyacente al nodo seleccionado, se calcula la distancia tentativa desde el nodo inicial pasando por el nodo seleccionado. Si esta distancia tentativa es menor que la distancia mínima conocida previamente para el nodo, se actualiza la distancia mínima y se marca el nodo seleccionado como el "padre" del nodo adyacente.

Marcado del nodo seleccionado como visitado: Después de actualizar todas las distancias mínimas adyacentes al nodo seleccionado, se marca como visitado y se elimina del conjunto de nodos no visitados.

Repetición del proceso: Se repiten los pasos 2 a 4 hasta que todos los nodos hayan sido visitados y sus distancias mínimas desde el nodo inicial se hayan determinado de manera definitiva.

Al finalizar el algoritmo, se tiene las distancias mínimas desde el nodo inicial a todos los demás nodos en el grafo, así como el camino más corto desde el nodo inicial a cada uno de estos nodos.

Algoritmo 3: Algoritmo de Dijkstra

```
1. Función Dijkstra(Grafo, Inicio, Fin):
2.     # Obtener el número de filas y columnas de la malla.
3.     filas = len(malla)
4.     columnas = len(malla[0])
5.
6.     # Definir los movimientos posibles:
7.     movimientos = [(0, 1), (0, -1), (1, 0), (-1, 0)]
8.
10       # Inicializar estructuras
11       # Crear una lista llamada "cola" y poner ahí el punto de inicio.
12       cola = {[inicio]}
13       # Crear un conjunto llamado "visitados" y agregar el punto de inicio.
14       visitados = set([inicio])
15
16       # Crear un diccionario llamado "padres" para guardar el inicio.
17       # El inicio no tiene padre.
18       padres = {inicio: None}
19
20       # Crear un diccionario llamado "distancia" para contar cuántos pasos desde el inicio.
21       distancia = {inicio: 0}
22.    # Mientras haya lugares en la cola por visitar:
23.    while cola:
24.        actual = cola.popleft()     # Se extrae el primero
25.
26.        # Si el lugar actual es el destino final:
27.        if actual == fin:
28.            break
29.
30.        x, y = actual
31.        # Explorar vecinos
32.        for dx, dy in movimientos:
33.            vecino = (x + dx, y + dy)
34.            vx, vy = vecino
35.
36.            # Verificar que esté dentro de la malla y no visitado
37.            if 0 <= vx < filas and 0 <= vy < columnas and vecino not in visitados:
38.                visitados.add(vecino)
39.                padres[vecino] = actual
40.                distancias[vecino] = distancias[actual] + 1  # Costo = 1
41.                cola.append(vecino)
42.
43.    return padres, distancias
44. Fin Función
```

Nodo inicial: (2,1)
Objetivo: (6,7)

Paso 1 Inicio:
Nodo actual: (2, 1)

Cola: []

Se añade a Padres

Se extrae de la cola (2, 1) y se agrega a la cola sus vecinos: (3, 1), (1, 1), (2, 2), (2, 0)

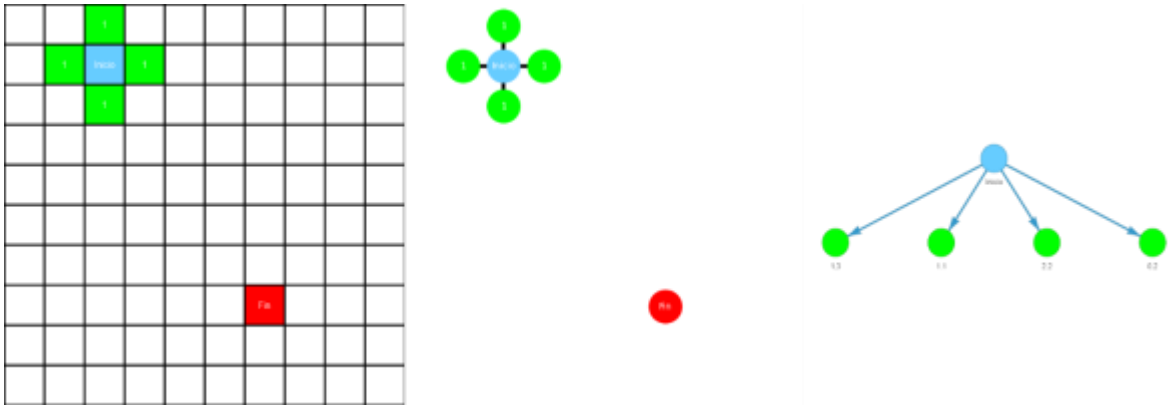

Figura .2.29 Paso 1 - Dijkstra

Paso 2 - Nodo actual: (3, 1)

Cola: [(1, 1), (2, 2), (2, 0)]

Se agrega a Visitados

Se añade a Padres

Se extrae de la cola (3, 1) y se agrega a la cola sus vecinos: (4, 1), (3, 2), (3, 0)

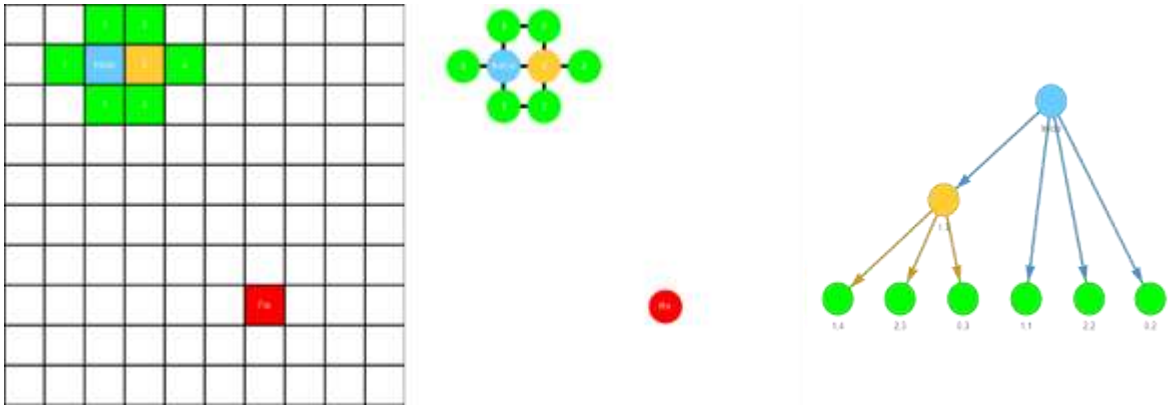

Figura 2.30 Paso 2 - Dijkstra

Paso 3 - Nodo actual: (1, 1)

Cola: [(2, 2), (2, 0), (4, 1), (3, 2), (3, 0)]

Se agrega a Visitados

Se añade a Padres

Se extrae de la cola (1, 1) y se agrega a la cola sus vecinos: (0, 1), (1, 2), (1, 0)

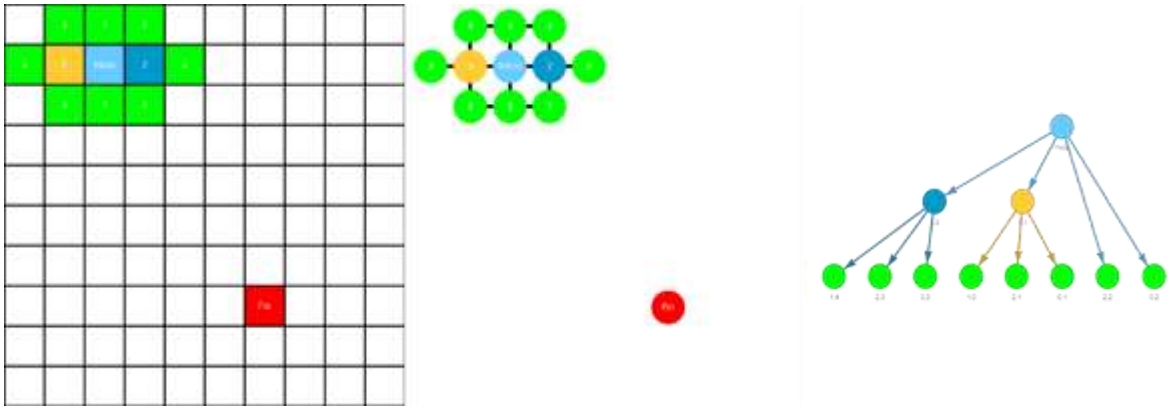

Figura 2.31 Paso 3 - Dijkstra

Paso 4 - Nodo actual: (2, 2)

Cola: [(2, 0), (4, 1), (3, 2), (3, 0), (0, 1), (1, 2), (1, 0)]

Se agrega a Visitados

Se añade a Padres

Se extrae de la cola (2, 2) y se agrega a la cola sus vecinos: (2, 3)

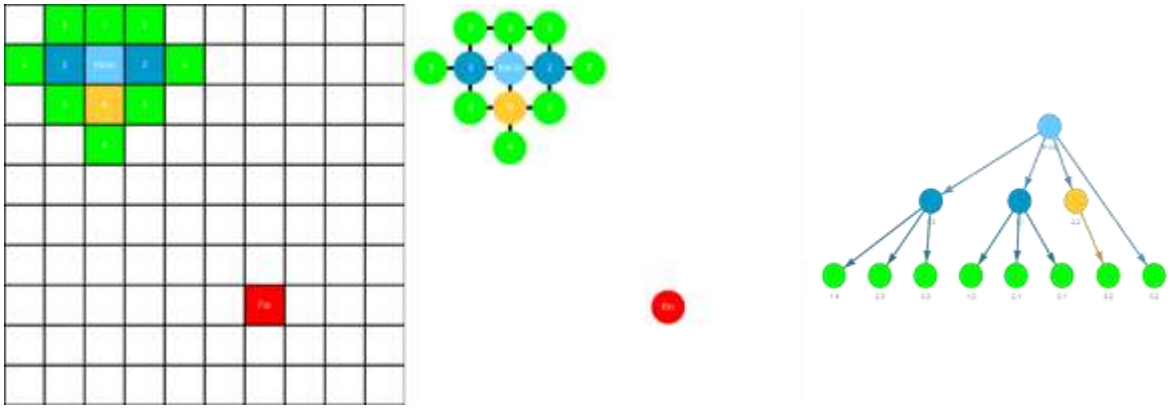

Figura 2.32 Paso 4 - Dijkstra

Paso 5 - Nodo actual: (2, 0)

Cola: [(4, 1), (3, 2), (3, 0), (0, 1), (1, 2), (1, 0), (2, 3)]

Se agrega a Visitados

Se extrae de la cola (2, 0) y no se agregan vecinos nuevos.

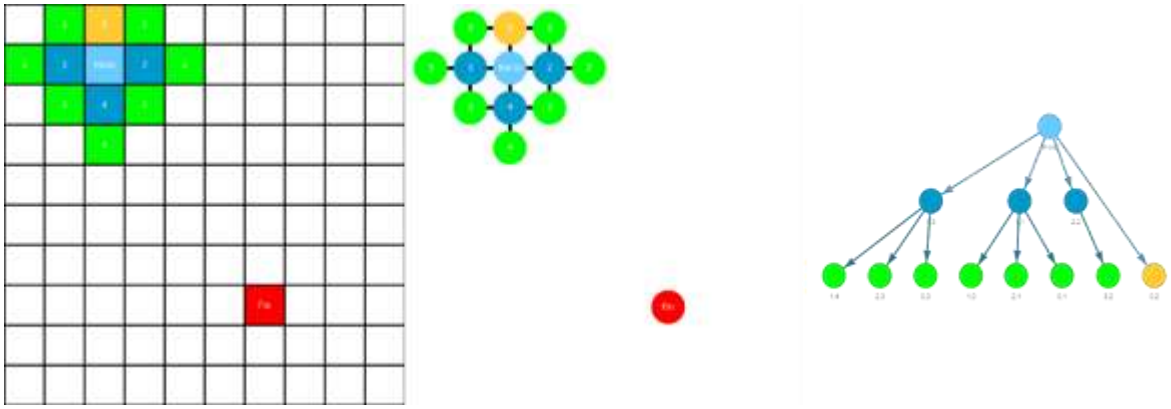

Figura 2.33 Paso 5 - Dijkstra

Paso 6 - Nodo actual: (4, 1)

Cola: [(3, 2), (3, 0), (0, 1), (1, 2), (1, 0), (2, 3)]

Se agrega a Visitados

Se añade a Padres

Se extrae de la cola (4, 1) y se agrega a la cola sus vecinos: (5, 1), (4, 2), (4, 0)

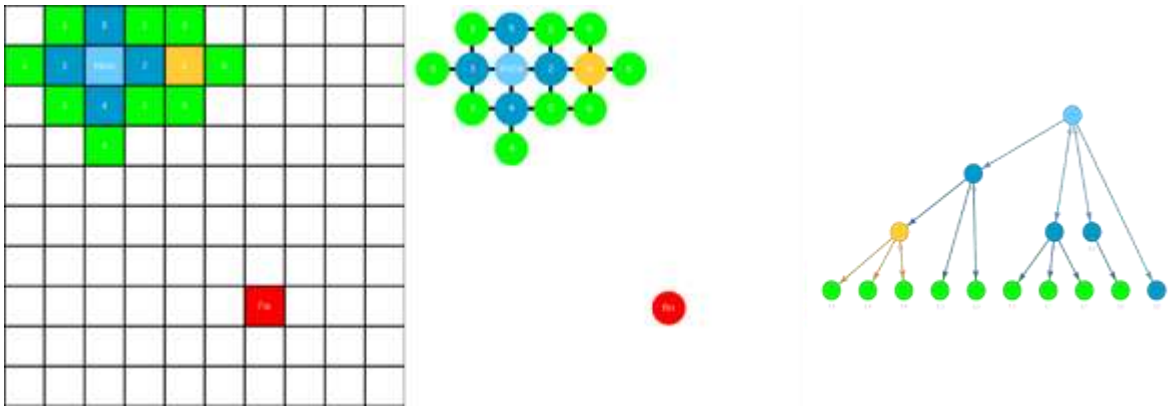

Figura 2.34 Paso 6 - Dijkstra

Paso 7 - Nodo actual: (3, 2)

Cola: [(3, 0), (0, 1), (1, 2), (1, 0), (2, 3), (5, 1), (4, 2), (4, 0)]

Se agrega a Visitados

Se añade a Padres

Se extrae de la cola (3, 2) y se agrega a la cola sus vecinos: (3, 3)

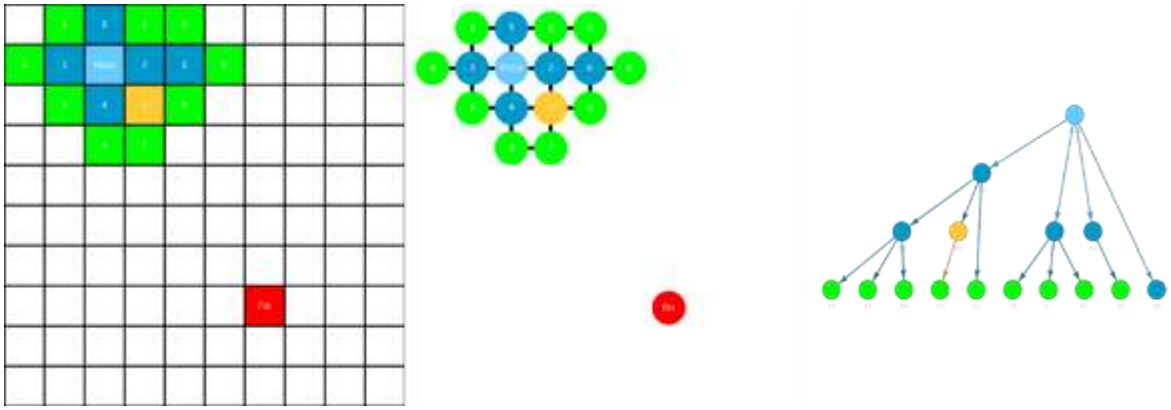

Figura 2.35 Paso 7 - Dijkstra

Paso 8 - Nodo actual: (3, 0)

Cola: [(0, 1), (1, 2), (1, 0), (2, 3), (5, 1), (4, 2), (4, 0), (3, 3)]

Se agrega a Visitados

Se extrae de la cola (3, 0) y no se agregan vecinos nuevos.

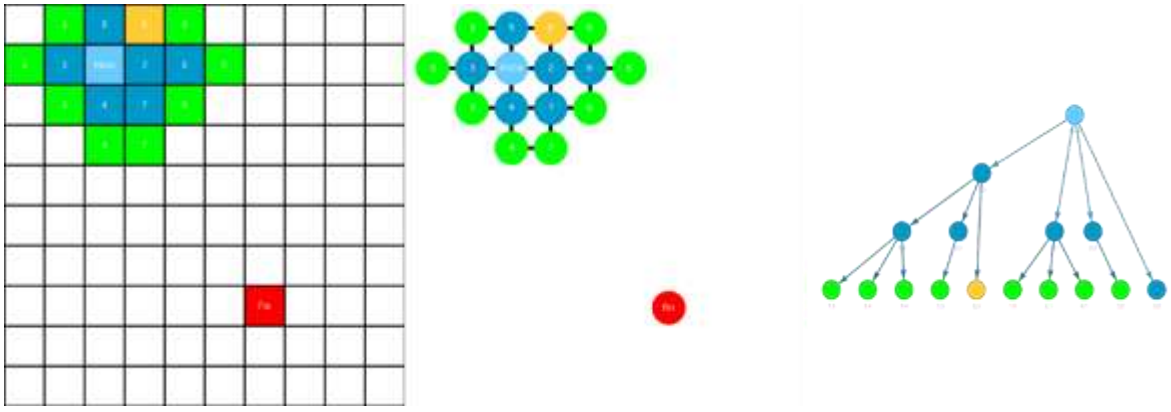

Figura 2.36 Paso 8 - Dijkstra

Paso 9 - Nodo actual: (0, 1)

Cola: [(1, 2), (1, 0), (2, 3), (5, 1), (4, 2), (4, 0), (3, 3)]

Se agrega a Visitados

Se añade a Padres

Se extrae de la cola (0, 1) y se agrega a la cola sus vecinos: (0, 2), (0, 0)

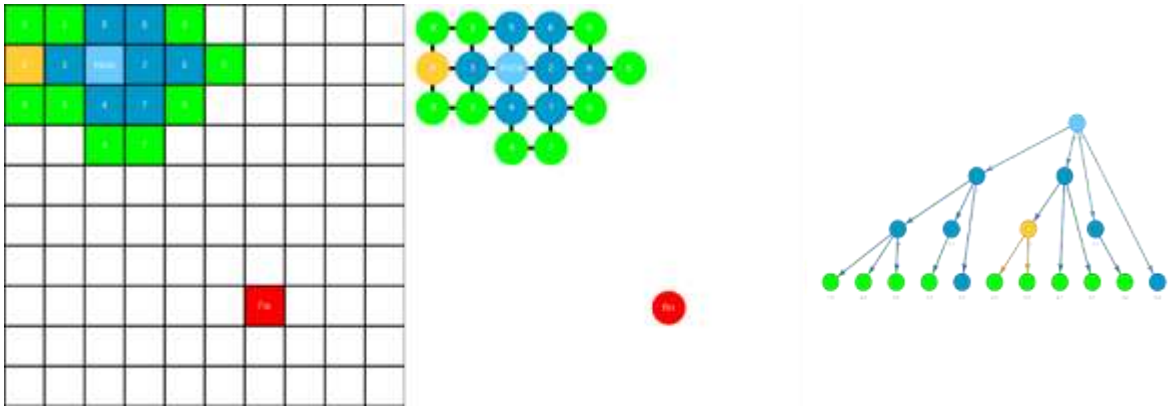

Figura 2.37 Paso 9 - Dijkstra

Paso 10 - Nodo actual: (1, 2)

Cola: [(1, 0), (2, 3), (5, 1), (4, 2), (4, 0), (3, 3), (0, 2), (0, 0)]

Se agrega a Visitados

Se añade a Padres

Se extrae de la cola (1, 2) y se agrega a la cola sus vecinos: (1, 3)

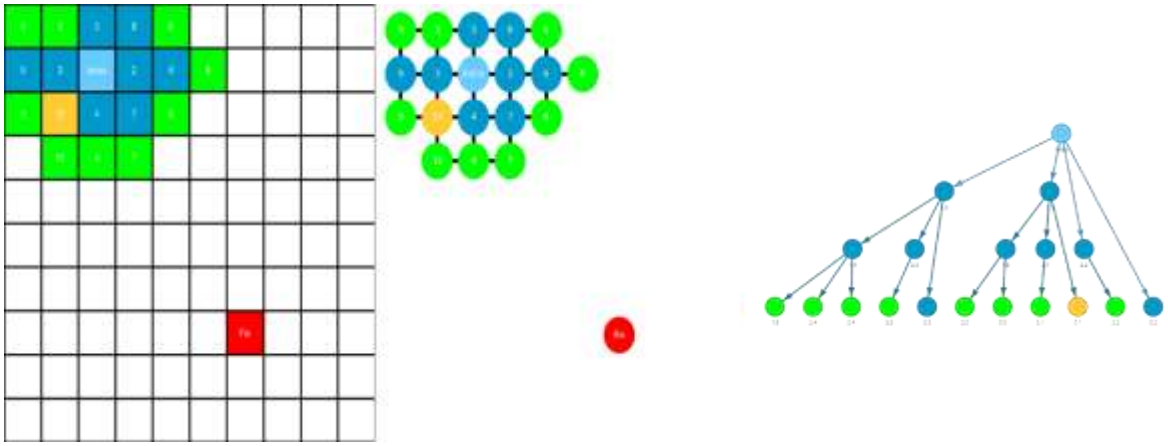

Figura 2.38 Paso 10 - Dijkstra

Paso 20 - Nodo actual: (2, 4)

Cola: [(6, 1), (5, 2), (5, 0), (4, 3), (3, 4), (0, 3), (1, 4)]

Se agrega a Visitados

Se añade a Padres

Se extrae de la cola (2, 4) y se agrega a la cola sus vecinos: (2, 5)

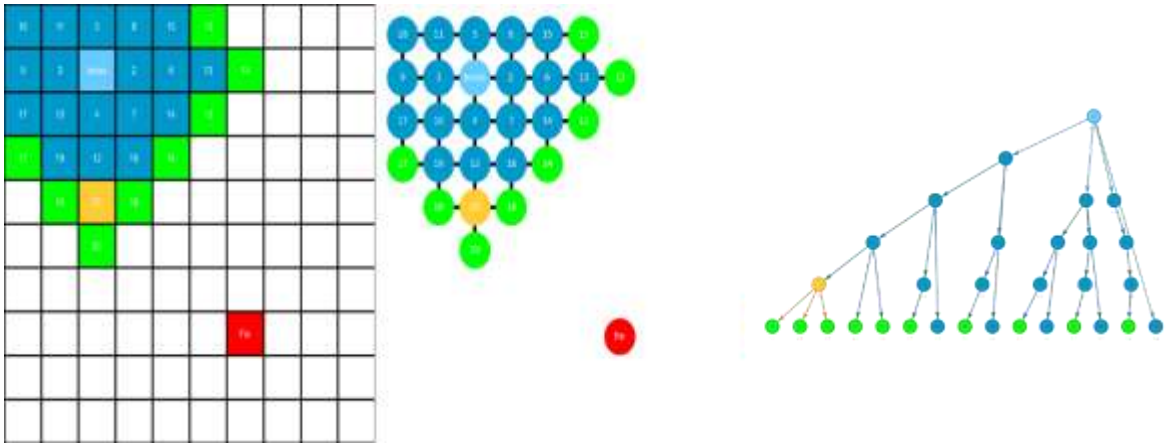

Figura 2.39 Paso 20 - Dijkstra

Paso 30 - Nodo actual: (6, 2)

Cola: [(6, 0), (5, 3), (4, 4), (3, 5), (0, 4), (1, 5), (2, 6), (8, 1), (7, 2), (7, 0)]

Se agrega a Visitados

Se añade a Padres

Se extrae de la cola (6, 2) y se agrega a la cola sus vecinos: (6, 3)

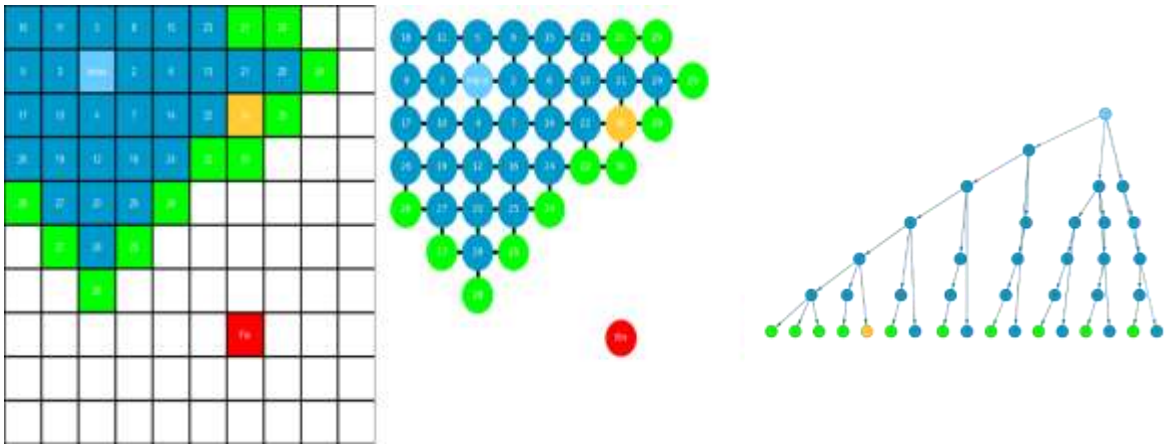

Figura 2.40 Paso 30 - Dijkstra

Paso 40 - Nodo actual: (7, 0)

Cola: [(6, 3), (5, 4), (4, 5), (3, 6), (0, 5), (1, 6), (2, 7), (9, 1), (8, 2), (8, 0), (7, 3)]

Se agrega a Visitados

Se extrae de la cola (7, 0) y no se agregan vecinos nuevos.

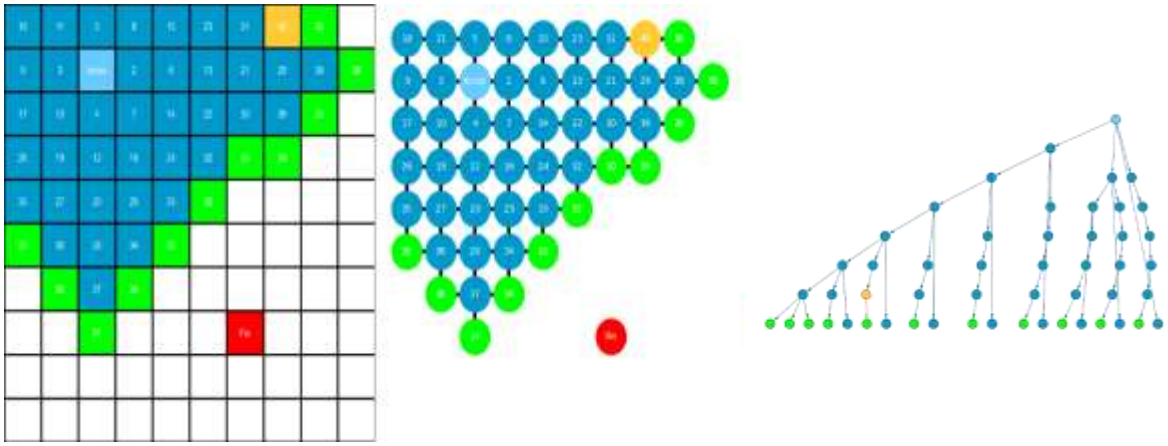

Figura 2.41 Paso 40 - Dijkstra

Paso 50 - Nodo actual: (8, 0)

Cola: [(7, 3), (6, 4), (5, 5), (4, 6), (3, 7), (0, 6), (1, 7), (2, 8), (9, 2), (9, 0), (8, 3)]

Se agrega a Visitados

Se extrae de la cola (8, 0) y no se agregan vecinos nuevos.

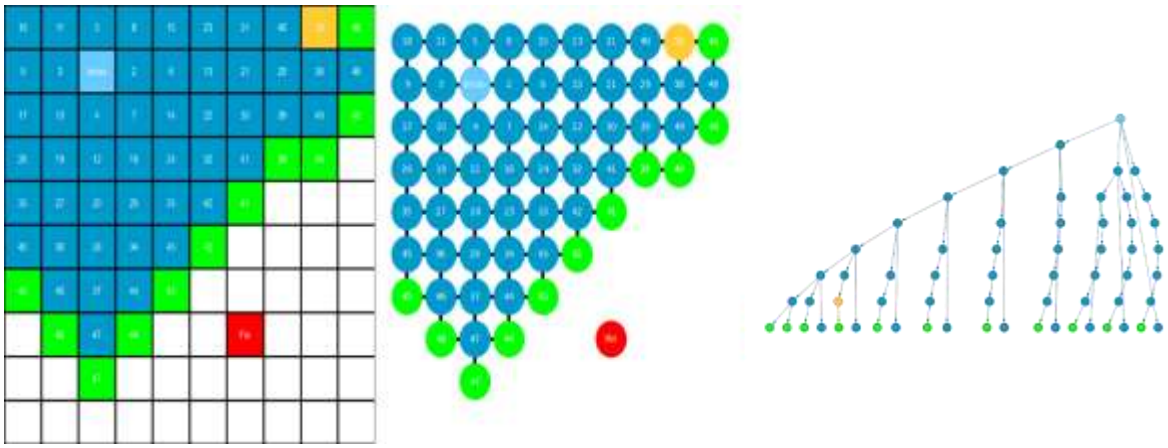

Figura 2.42 Paso 50 - Dijkstra

Paso 60 - Nodo actual: (9, 0)

Cola: [(8, 3), (7, 4), (6, 5), (5, 6), (4, 7), (3, 8), (0, 7), (1, 8), (2, 9), (9, 3)]

Se agrega a Visitados

Se extrae de la cola (9, 0) y no se agregan vecinos nuevos.

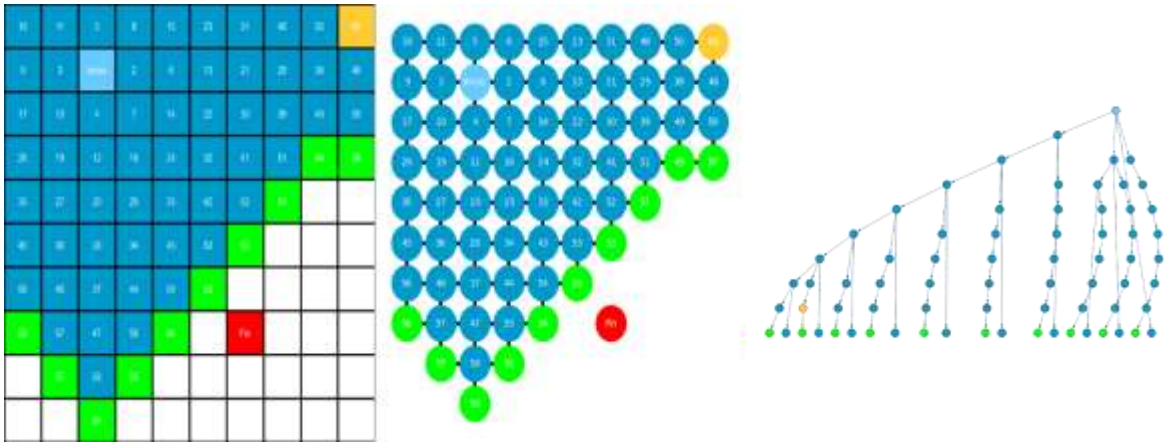

Figura 2.43 Paso 60 - Dijkstra

Paso 70 - Nodo actual: (9, 3)

Cola: [(8, 4), (7, 5), (6, 6), (5, 7), (4, 8), (3, 9), (0, 8), (1, 9)]

Se agrega a Visitados

Se añade a Padres

Se extrae de la cola (9, 3) y se agrega a la cola sus vecinos: (9, 4)

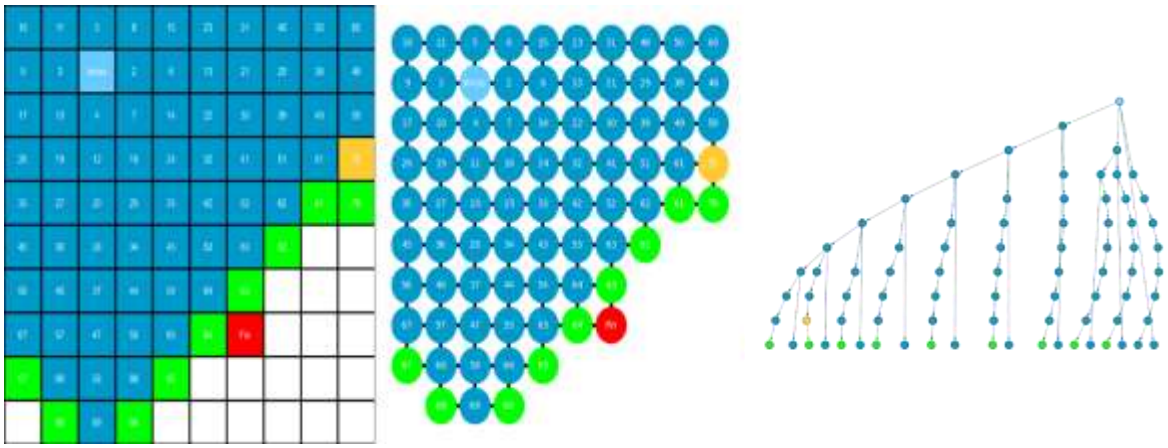

Figura 2.44 Paso 70 - Dijkstra

Paso 80 - Nodo actual: (8, 5)

Cola: [(7, 6), (6, 7), (5, 8), (4, 9), (0, 9), (9, 5)]

Se agrega a Visitados

Se añade a Padres

Se extrae de la cola (8, 5) y se agrega a la cola sus vecinos: (8, 6)

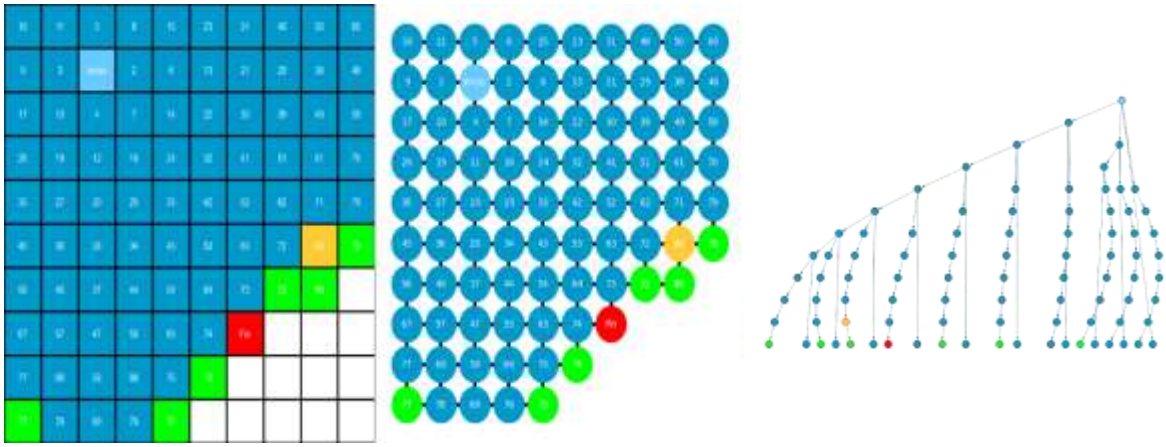

Figura 2.45 Paso 80 - Dijkstra

Paso 82 - Nodo actual: (6, 7)

Cola: [(5, 8), (4, 9), (0, 9), (9, 5), (8, 6), (7, 7)]

Visitados: {(4, 0), (4, 9), (5, 1), (8, 0), (0, 5), (2, 2), (6, 2), (7, 1), (4, 2), (3, 6), (5, 3), (8, 2), (9, 1), (0, 7), (2, 4), (1, 8), (6, 4), (7, 3), (3, 8), (5, 5), (8, 4), (9, 3), (0, 0), (0, 9), (6, 6), (7, 5), (3, 1), (5, 7), (9, 5), (0, 2), (1, 3), (7, 7), (3, 3), (5, 0), (1, 5), (6, 1), (7, 0), (3, 5), (5, 2), (4, 4), (9, 0), (1, 7), (2, 6), (7, 2), (3, 7), (5, 4), (4, 6), (9, 2), (8, 6), (1, 0), (1, 9), (2, 8), (7, 4), (3, 0), (3, 9), (5, 6), (4, 8), (1, 2), (0, 4), (2, 1), (3, 2), (4, 1), (8, 1), (1, 4), (0, 6), (2, 3), (6, 3), (3, 4), (4, 3), (8, 3), (1, 6), (0, 8), (2, 5), (6, 5), (4, 5), (8, 5), (9, 4), (0, 1), (2, 7), (6, 7), (7, 6), (4, 7), (5, 8), (1, 1), (0, 3), (2, 0), (2, 9), (6, 0)}

Camino encontrado (Padres del nodo fin):

(2, 1) , (3, 1) , (4, 1) , (5, 1) , (6, 1) , (6, 2) , (6, 3) , (6, 4) , (6, 5) , (6, 6) , (6, 7)

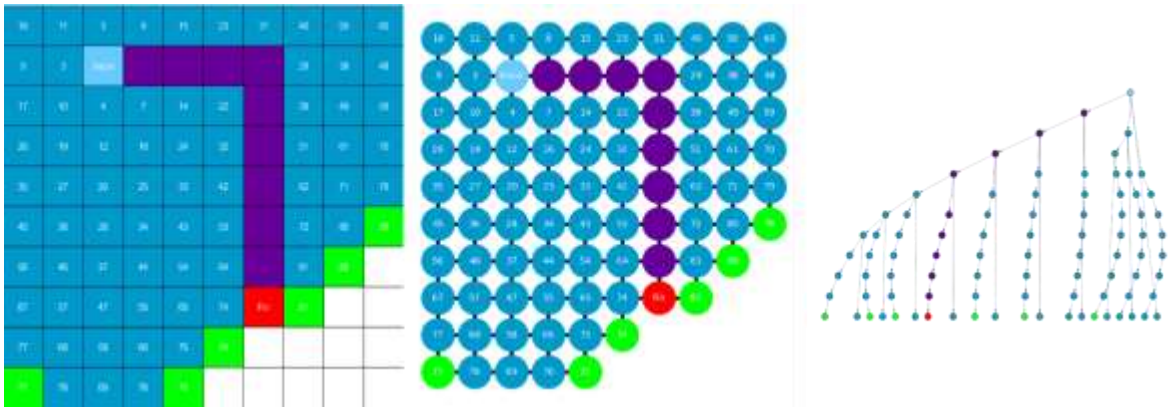

Figura 2.46 Paso 82 - Dijkstra

El algoritmo de Dijkstra fue muy utilizado hasta la aparición del algoritmo A* en la búsqueda de trayectorias. El algoritmo de Dijkstra no está optimizado para que la solución se encuentre rápidamente ya que toma decisiones basadas en la información local (los pesos de las aristas), y a veces podría no seleccionar la mejor opción global en términos de encontrar la solución más rápida. A continuación, se presenta el grafo del funcionamiento del algoritmo de Dijkstra (Algoritmo 3).

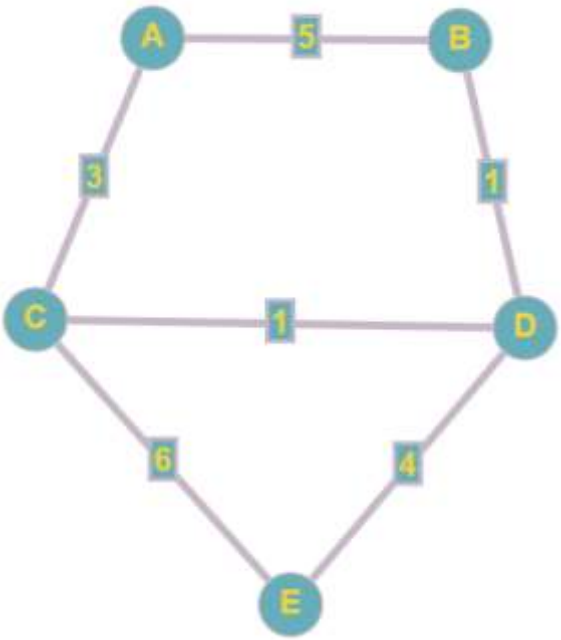

Figura 2.47 Grafo de algoritmo de Dijkstra y sus pesos

El algoritmo de Dijkstra implementa el uso de pesos, en el ejemplo que se presenta en las siguientes tablas, se indica la trayectoria hacía el Nodo **A** desde los nodos que se encuentran en el grafo, evaluando las trayectorias encontradas con los pesos en sus aristas.

Tabla 2.1 Trayectoria hacía el nodo A.

Nodo	**Trayectoria desde origen al Nodo meta**	**Peso de trayectoria**
A	A Desde el nodo A hacia A	0

En la Tabla 1 se puede apreciar que la trayectoria hacia el nodo A desde el nodo A, en este ejemplo, al ser el nodo A el nodo inicio y el nodo final el coste de la trayectoria sería 0.

Tabla 2.2 Trayectoria hacía el nodo A desde el nodo B.

Nodo	Desde el origen al Nodo	Peso de trayectoria
B	B-D-E-C-A Primer nodo visitado D Segundo nodo visitado E Tercer nodo visitado C Cuarto nodo visitado A	1+4+6+3 = 14
B	B-D-C-A Primer nodo visitado D Segundo nodo visitado C Tercer nodo visitado A	1+1+3 = 5
B	B-A Primer nodo visitado A	5

En la Tabla 2 se muestran las diferentes trayectorias que se utilizan para llegar del nodo B hasta el nodo A. La primera trayectoria es la trayectoria B-D-E-C-A siendo esta trayectoria la más pesada para llegar al nodo A ya que la suma de los pesos es 14, y las trayectorias más ligeras serían las trayectorias B-D-C-A y la trayectoria BA ambas con un peso total de 5 sin importar los nodos visitados en esta ocasión.

Tabla 2.3 Trayectoria hacía el nodo A desde el nodo C.

Nodo	Desde el origen al Nodo	Peso de trayectoria
C	C-E-D-B-A Primer nodo visitado E Segundo nodo visitado D Tercer nodo visitado B Cuarto nodo visitado A	6+4+1+5 = 16
C	C-D-B-A Primer nodo visitado D Segundo nodo visitado B Tercer nodo visitado A	1+1+5 = 7
C	C-A Primer nodo visitado A	3

En la Tabla 3 se puede observar la trayectoria más pesada dentro del grafo, esta trayectoria es la C-E-D-B-A que cuenta con un peso total de 16, siendo que esta recorre todos los nodos dentro del grafo, y por otra parte se puede también apreciar la trayectoria más ligera que es la trayectoria C-A con un peso total de 3.

Tabla 2.4 Trayectoria hacía el nodo A desde el nodo D y sus pesos.

Nodo	Desde el origen al Nodo	Peso de trayectoria
D	D-E-C-A Primer nodo visitado E Segundo nodo visitado C Tercer nodo visitado A	4+6+3 = 13
D	D-C-A Primer nodo visitado C Segundo nodo visitado A	1+3 = 4
D	D-B-A Primer nodo visitado B Segundo nodo visitado A	1+5 = 6

En la tabla 4 se presenta las trayectorias para llegar al nodo A desde el nodo D, teniendo su trayectoria más pesada un total de 13 y su trayectoria más ligera 4.

Tabla 2.5 Trayectoria hacía el nodo A desde el nodo E y sus pesos.

Nodo	Desde el origen al Nodo	Peso de trayectoria
E	E-C-D-B-A Primer nodo visitado C Segundo nodo visitado D Tercer nodo visitado B Cuarto nodo visitado A	6+1+1+5 = 13
E	E-D-B-A Primer nodo visitado D Segundo nodo visitado B Tercer nodo visitado A	4+1+5 = 10
E	E-D-C-A Primer nodo visitado D Segundo nodo visitado C Tercer nodo visitado A	4+1+3 = 8
E	E-C-A Primer nodo visitado C Segundo nodo visitado A	6+3 = 9

En la Tabla 5 se puede visualizar qué el hecho de que una trayectoria visite más nodos no siempre quiere decir que es la más pesada, así como que el hecho de que una trayectoria viste menos nodos no significa que esta sea la más ligera.

2.1.2.4 Algoritmo A*

El algoritmo A* es un método de búsqueda de camino que encuentra la ruta más corta entre un nodo de inicio y un nodo objetivo en un grafo o red ponderada. El algoritmo utiliza una función heurística para estimar la distancia entre el nodo actual y el nodo objetivo, lo que lo hace más eficiente que los algoritmos de búsqueda no informados, como BFS y DFS.

En el algoritmo A*, se utilizan tres funciones clave para guiar la búsqueda del camino más corto entre dos puntos en un grafo o una malla:

- ***g(n)***: Esta función representa el costo acumulado para llegar al nodo *n* desde el nodo inicial. Es decir, *g(n)* mide el costo real o la distancia recorrida para llegar a *n*.
- ***h(n)***: Esta función estima el costo restante para llegar al nodo objetivo desde el nodo *n*. Se basa en una heurística, que es una función que proporciona una estimación del costo. Idealmente, *h(n)* debe ser una estimación optimista (nunca debe sobreestimar el costo real) para asegurar que el algoritmo A* sea eficiente y correcto.
- ***f(n)***: Esta función combina las dos anteriores y se utiliza para priorizar los nodos durante la búsqueda. *f(n)* se calcula como la suma de *g(n)* y *h(n)*:

$$f(n)=g(n)+h(n)$$

Donde *g(n)* es el costo real para llegar a *n* desde el inicio, y *h(n)* es la estimación del costo para llegar al objetivo desde *n*. La función *f(n)* permite al algoritmo A* seleccionar los nodos que parecen más prometedores para alcanzar el objetivo de manera eficiente, considerando tanto el costo actual como la estimación del costo futuro.

Algoritmo 4: Algoritmo A*

```
1. Funcion A_Estrella(Grafo, Inicio, Meta):
2.     # Obtener el número de filas y columnas de la malla
3.     filas = len(malla)
4.     columnas = len(malla[0])
5.
6.     # Definir los movimientos posibles
7.     movimientos = [(0, 1), (0, -1), (1, 0), (-1, 0)]
8.
9.     # Función heurística (distancia Manhattan en este caso)
10.    def heuristica(nodo):
11.        x, y = nodo
12.        fx, fy = fin
13.        return abs(x - fx) + abs(y - fy)
14.
15.    # Inicializar estructuras
16.    # Cola de prioridad con (f_score, g_score, nodo)
17.    heap = [(heuristica(inicio), 0, inicio)]
18.
19.    # Diccionario para guardar los padres
20.    padres = {inicio: None}
21.
22.    # Diccionario para guardar los g_scores (costo real desde inicio)
23.    g_scores = {inicio: 0}
24.
25.    # Mientras haya nodos en el heap
26.    while heap:
27.        # Extraer el nodo con menor f_score
28.        f_score, g_score, actual = heappop(heap)
29.
30.        # Si llegamos al objetivo
31.        if actual == fin:
32.            break
33.
34.        # Si encontramos un g_score mayor al almacenado, ignorar
35.        if g_score > g_scores.get(actual, float('inf')):
36.            continue
37.
38.        # Obtener coordenadas del nodo actual
39.        x, y = actual
40.
41.        # Explorar vecinos
```

```
42.         for dx, dy in movimientos:
43.             vecino = (x + dx, y + dy)
44.             vx, vy = vecino
45.
46.             # Verificar que esté dentro de la malla
47.             if 0 <= vx < filas and 0 <= vy < columnas:
48.                 # Calcular costo tentativo (asumimos 1 para movimientos básicos)
49.                 costo_tentativo = g_score + 1
50.
51.                 # Si encontramos un camino mejor al vecino
52.                 if costo_tentativo < g_scores.get(vecino, float('inf')):
53.                     # Actualizar padre
54.                     padres[vecino] = actual
55.                     # Actualizar g_score
56.                     g_scores[vecino] = costo_tentativo
57.                     # Calcular f_score (g_score + heurística)
58.                     f_score = costo_tentativo + heuristica(vecino)
59.                     # Agregar a la cola de prioridad
60.                     heappush(heap, (f_score, costo_tentativo, vecino))
61.
62.     return padres, g_scores
63. Fin Funcion
```

El algoritmo selecciona el nodo con la menor función de costo *f(n)* y lo marca como "visitado". Luego, se expanden los nodos vecinos del nodo actual y se calculan sus funciones de costo *f(n)*. Si un nodo vecino ya ha sido visitado, el algoritmo compara el costo actual con el costo anterior y selecciona el camino con el menor costo. Si un nodo vecino no ha sido visitado, se lo agrega a la lista de nodos a visitar.

El algoritmo continúa expandiendo los nodos y seleccionando el nodo con la función de costo más baja hasta que se alcance el nodo objetivo o hasta que no queden más nodos por visitar. Si se alcanza el nodo objetivo, el algoritmo ha encontrado la ruta óptima desde el nodo inicial hasta el nodo objetivo. Si no se alcanza el nodo objetivo, no existe un camino entre el nodo inicial y el nodo objetivo.

Ejemplo:

Paso 1 - Nodo actual: (2, 1)

Se extrae de la cola de prioridad (2, 1) y se agregan vecinos: (3, 1), (1, 1), (2, 2), (2, 0)

Se añade Padre (2, 1)

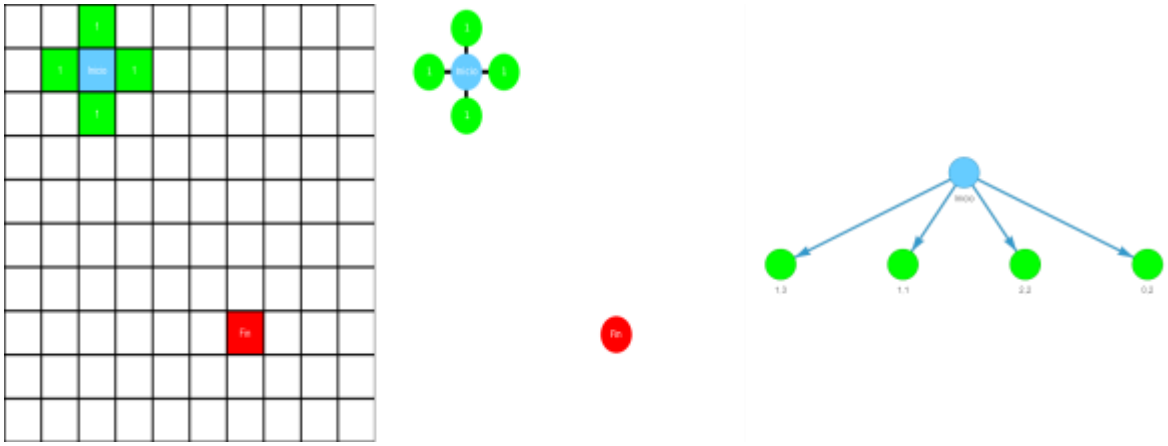

*Figura 2.48 Paso 1 – A**

Paso 2 - Nodo actual: (2, 2)

Se extrae de la cola de prioridad (2, 2) y se agregan vecinos: (3, 2), (1, 2), (2, 3)

Se añade Padre (2, 2)

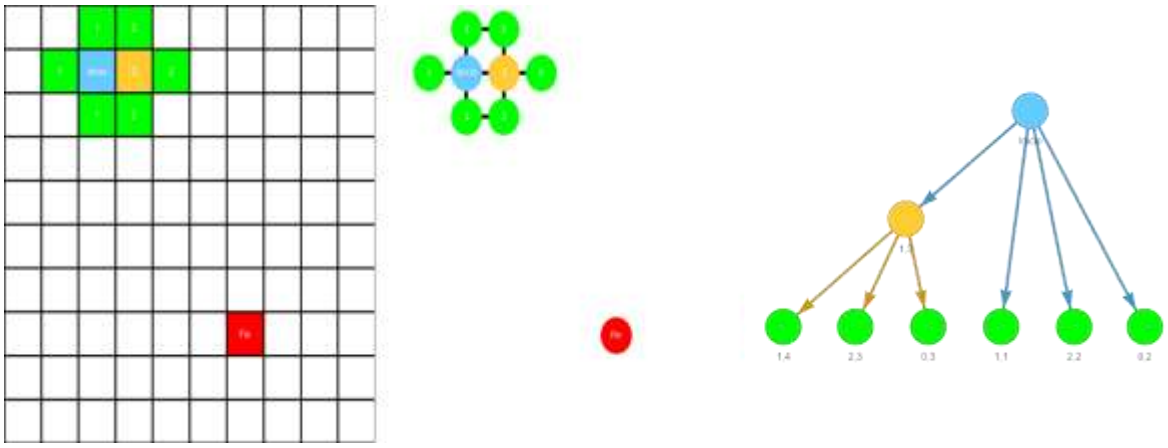

*Figura 2.49 Paso 2 – A**

Paso 3 - Nodo actual: (3, 1)

Se extrae de la cola de prioridad (3, 1) y se agregan vecinos: (4, 1), (3, 0)

Se añade Padre (3, 1)

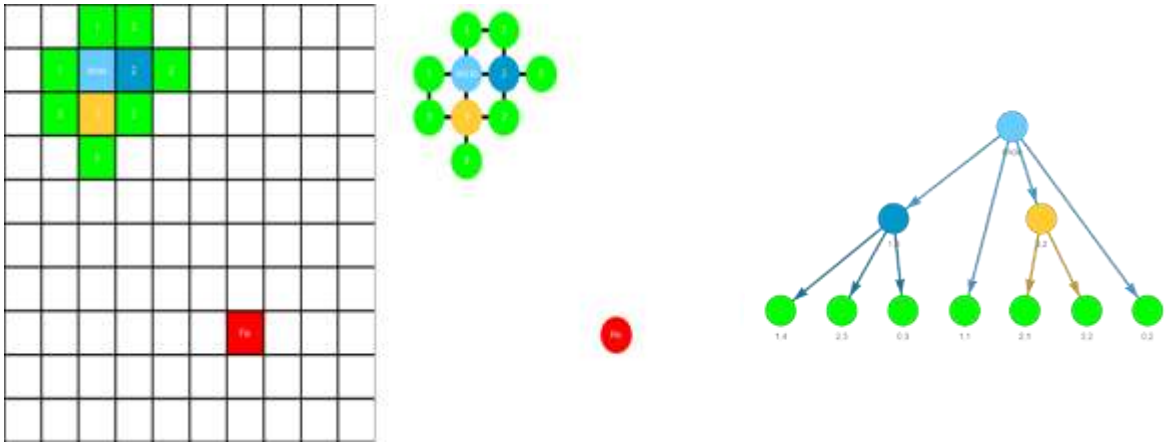

*Figura 2.50 Paso 3 – A**

Paso 4 - Nodo actual: (2, 3)

Se extrae de la cola de prioridad (2, 3) y se agregan vecinos: (3, 3), (1, 3), (2, 4)

Se añade Padre (2, 3)

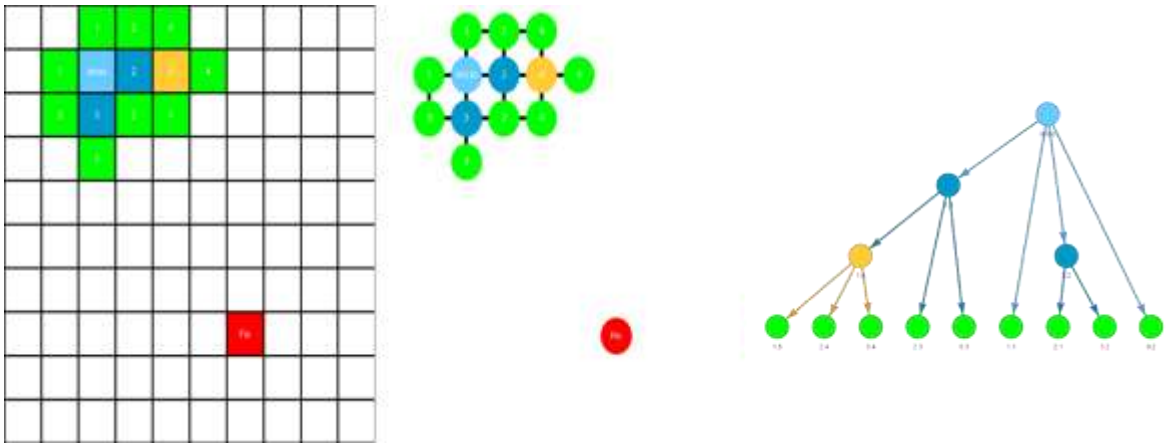

*Figura 2.51 Paso41 – A**

Paso 5 - Nodo actual: (3, 2)

Se extrae de la cola de prioridad (3, 2) y se agregan vecinos: (4, 2)

Se añade Padre (3, 2)

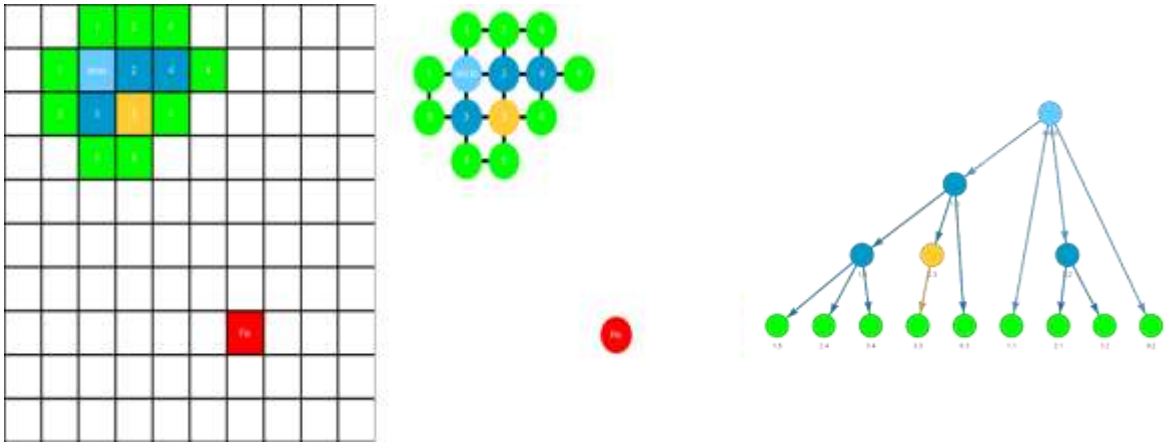

*Figura 2.52 Paso 5 – A**

Paso 6 - Nodo actual: (4, 1)

Se extrae de la cola de prioridad (4, 1) y se agregan vecinos: (5, 1), (4, 0)

Se añade Padre (4, 1)

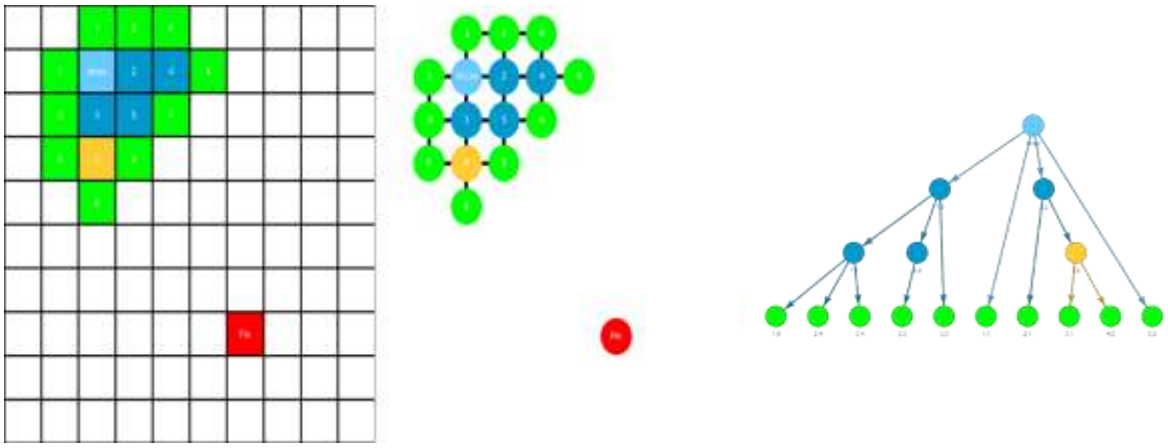

*Figura 2.53 Paso 6 – A**

Paso 7 - Nodo actual: (2, 4)

Se extrae de la cola de prioridad (2, 4) y se agregan vecinos: (3, 4), (1, 4), (2, 5)

Se añade Padre (2, 4)

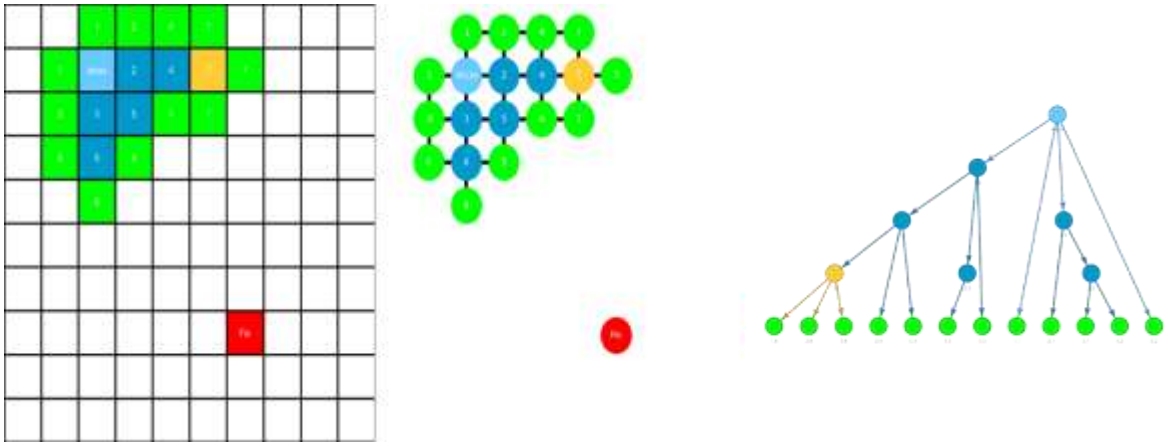

*Figura 2.54 Paso 7 – A**

Paso 8 - Nodo actual: (3, 3)

Se extrae de la cola de prioridad (3, 3) y se agregan vecinos: (4, 3)

Se añade Padre (3, 3)

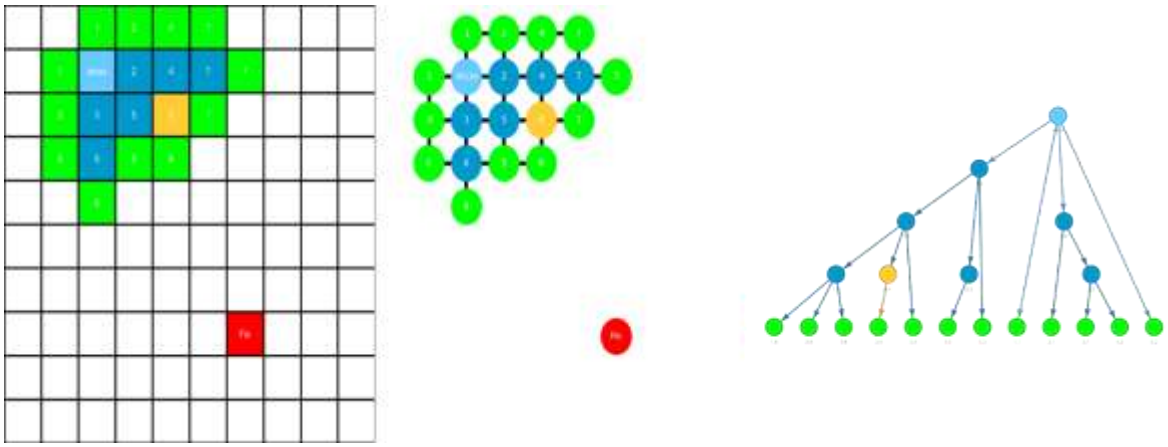

*Figura 2.55 Paso 8 – A**

Paso 9 - Nodo actual: (4, 2)

Se extrae de la cola de prioridad (4, 2) y se agregan vecinos: (5, 2)

Se añade Padre (4, 2)

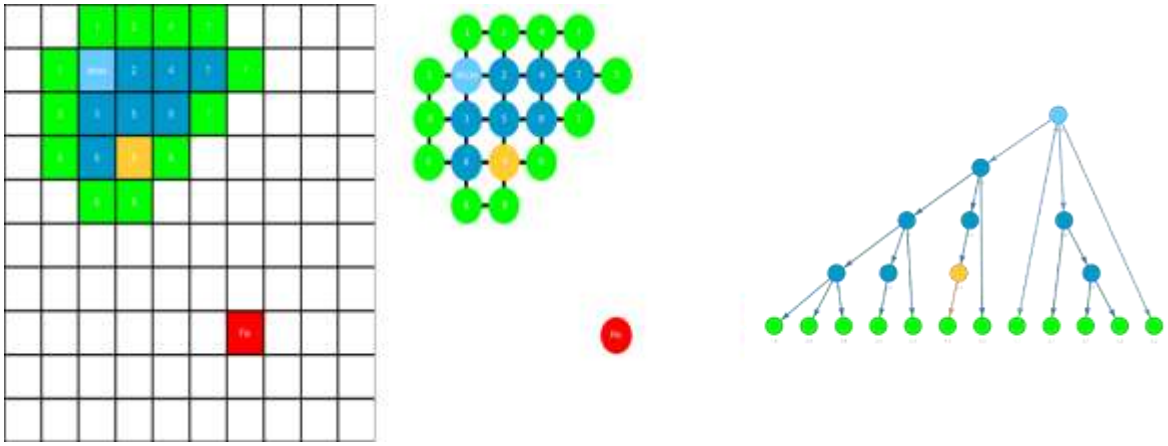

*Figura 2.56 Paso 9 – A**

Paso 10 - Nodo actual: (5, 1)

Se extrae de la cola de prioridad (5, 1) y se agregan vecinos: (6, 1), (5, 0)

Se añade Padre (5, 1)

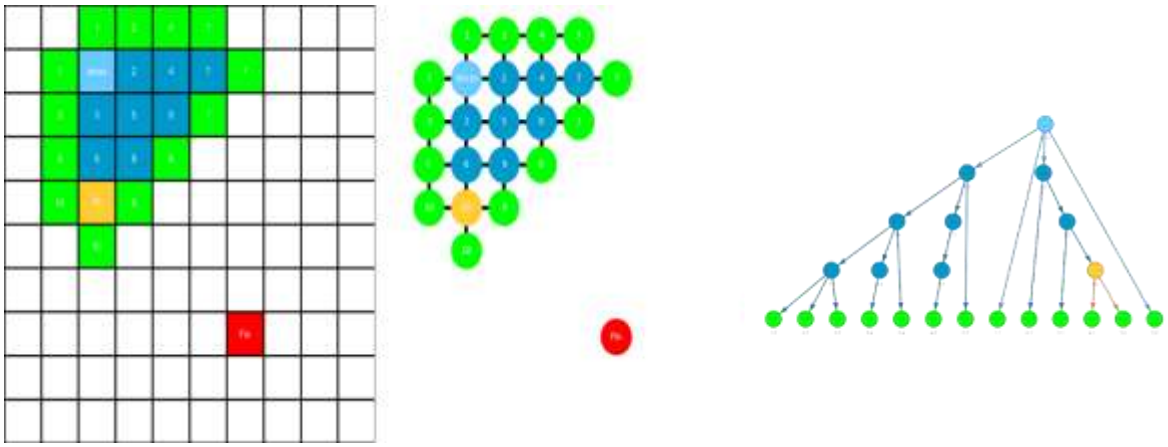

*Figura 2.57 Paso 10 – A**

Paso 15 - Nodo actual: (6, 1)

Se extrae de la cola de prioridad (6, 1) y se agregan vecinos: (7, 1), (6, 0)

Se añade Padre (6, 1)

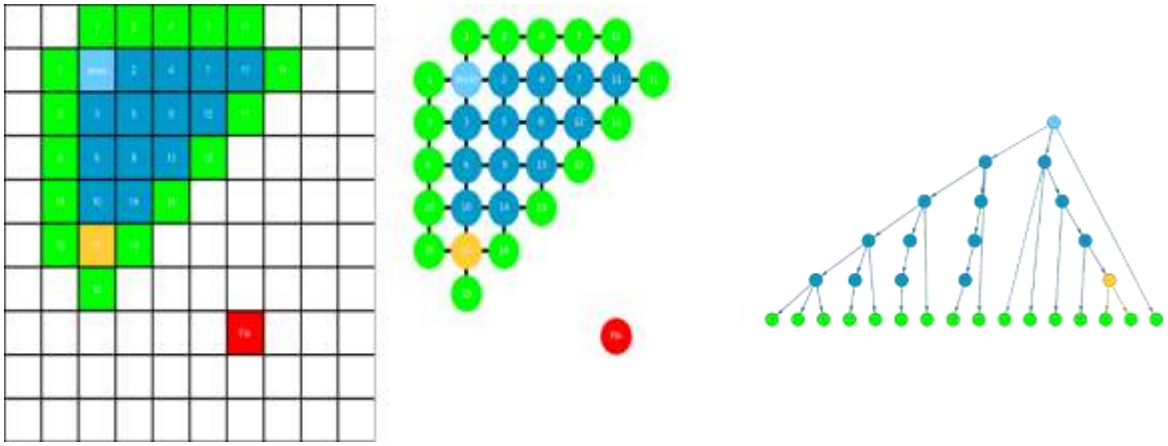

*Figura 2.58 Paso 15 – A**

Paso 20 - Nodo actual: (6, 2)

Se extrae de la cola de prioridad (6, 2) y se agregan vecinos: (7, 2)

Se añade Padre (6, 2)

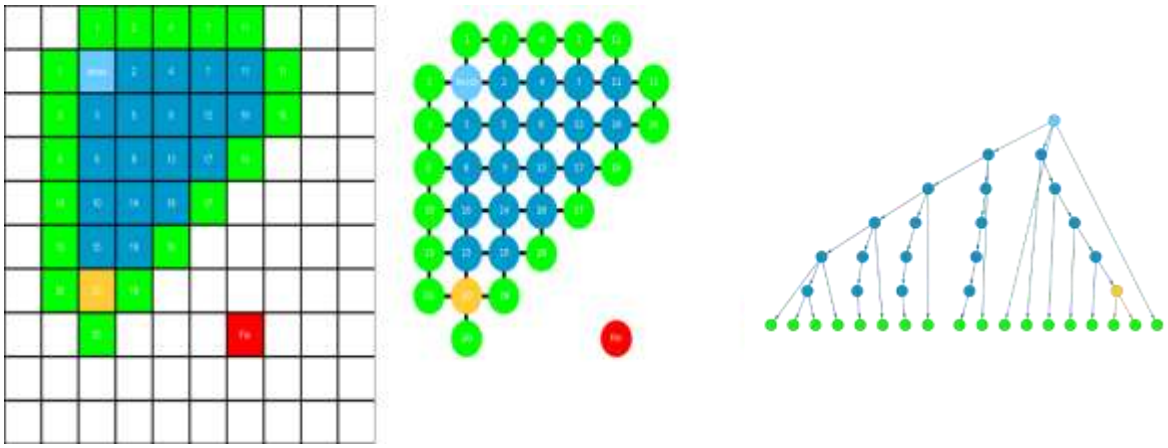

*Figura 2.59 Paso 20 – A**

Paso 25 - Nodo actual: (6, 3)

Se extrae de la cola de prioridad (6, 3) y se agregan vecinos: (7, 3)

Se añade Padre (6, 3)

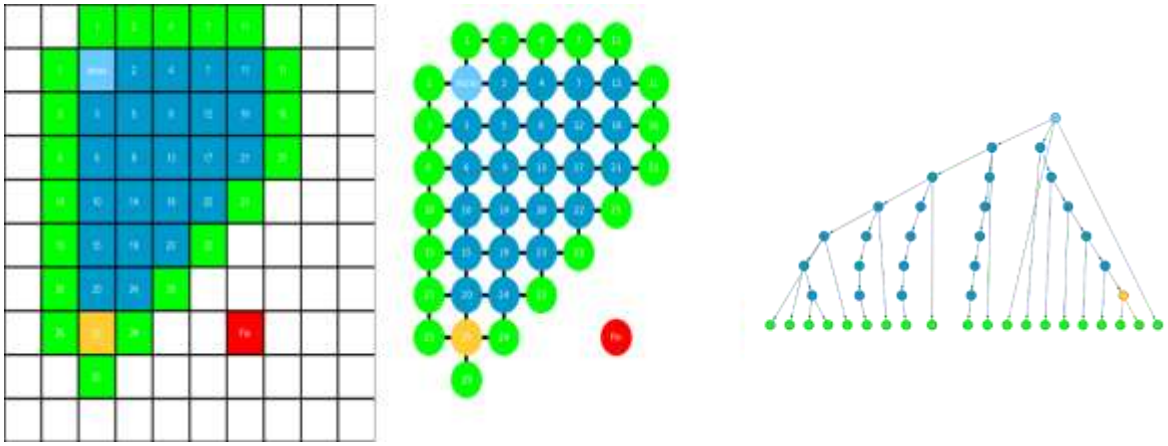

*Figura 2.60 Paso 25 – A**

Paso 30 - Nodo actual: (4, 7)

Se extrae de la cola de prioridad (4, 7) y se agregan vecinos: (5, 7), (4, 8)

Se añade Padre (4, 7)

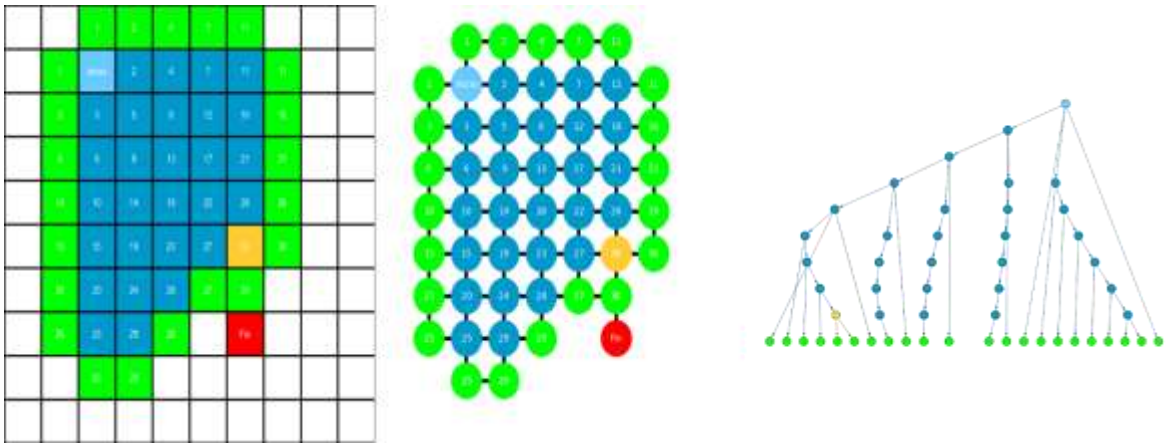

*Figura 2.61 Paso 30 – A**

Paso 35 - Nodo actual: (6, 7)

Camino encontrado:

(2, 1) , (2, 2) , (2, 3) , (2, 4) , (2, 5) , (2, 6) , (2, 7) , (3, 7) , (4, 7) , (5, 7) , (6, 7)

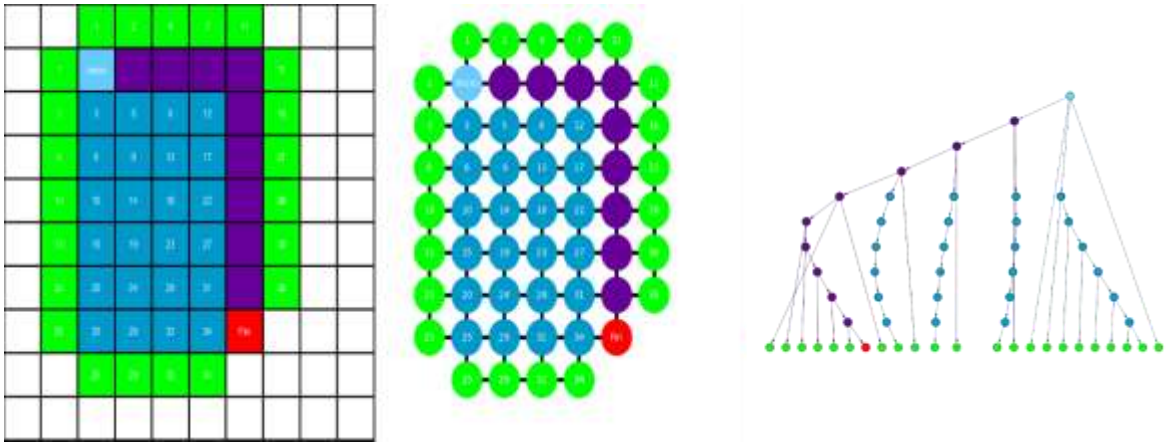

*Figura 2.62 Paso 35 – A**

El algoritmo A* se considera una mejora sobre el algoritmo de Dijkstra, ya que la función heurística le permite estimar el costo del camino desde el nodo actual hasta el nodo objetivo y priorizar la exploración de nodos que están más cerca del objetivo. Esto reduce la cantidad de nodos que deben ser explorados y hace que el algoritmo sea más rápido y eficiente.

Según el libro *Cracking the Coding Interview* el algoritmo A* encuentra la trayectoria de menor costo entre un nodo de origen y un nodo de destino (o uno de varios nodos de destino). Eso amplía el algoritmo de Dijkstra y logra un mejor rendimiento mediante el uso de heurística [17]. A continuación, se presenta el grafo del funcionamiento del algoritmo A* (Algoritmo 4).

A continuación, se muestra un ejemplo de un grafo con pesos (Figura 2.70).

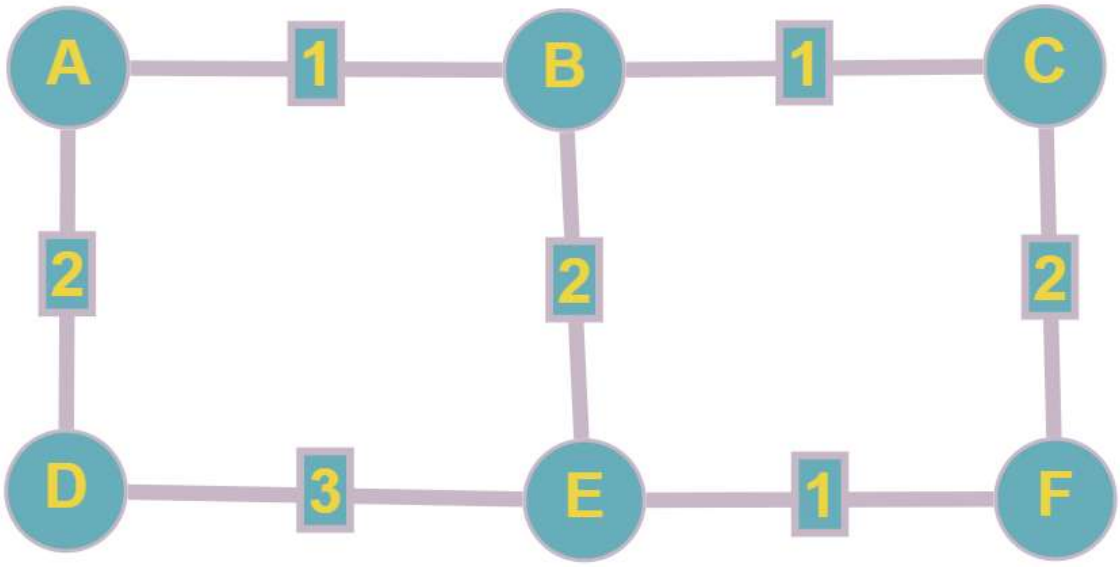

*Figura 2.63 Grafo con pesos para el algoritmo A**

Nodo inicial: A
Nodo objetivo: G
Heurística h(n):

- h(A) = 7
- h(B) = 5
- h(C) = 6
- h(D) = 4
- h(E) = 3
- h(F) = 2
- h(G) = 0

Corrida de Escritorio

1. **Inicio en el nodo A**:
    - **Distancias**: {A: 0, B: ∞, C: ∞, D: ∞, E: ∞, F: ∞, G: ∞}
    - **Cola de Prioridad**: [(7, A)]

    Se extrae A y se explora B y C.

2. **Explorando nodo B y C**:
    - **Distancias**: {A: 0, B: 2, C: 1, D: ∞, E: ∞, F: ∞, G: ∞}
    - **Cola de Prioridad**: [(7, C), (9, B)]

Se extrae C y se explora F.

3. **Explorando nodo F**:
   - **Distancias**: {A: 0, B: 2, C: 1, D: ∞, E: ∞, F: 6, G: ∞}
   - **Cola de Prioridad**: [(8, F), (9, B)]

   Se extrae F y luego G (vecino de F).

4. **Finalización**:
   - **Distancias**: {A: 0, B: 2, C: 1, D: ∞, E: ∞, F: 6, G: 12}

Trayectoria más corta desde A a G con A*: A → C → F → G (Distancia = 12)

DFS y BFS son algoritmos de búsqueda no informada que exploran el espacio de búsqueda sin considerar el costo. Dijkstra y A* son algoritmos de búsqueda de caminos en grafos, pero A* utiliza una heurística adicional para dirigir la búsqueda de manera más eficiente. Dijkstra considera todos los nodos por igual, mientras que A* combina el costo acumulado con una heurística que estima el costo restante hasta el objetivo. La heurística permite a A* priorizar nodos más prometedores, lo que puede ser más eficiente en términos de tiempo de ejecución, especialmente en grafos grandes o complejos.

2.1.2 Animación

Crear un entorno bidimensional con elementos gráficos tridimensionales implica combinar varios elementos fundamentales para construir y renderizar el espacio y los objetos dentro de él. A continuación, se detallan los componentes esenciales que se usan durante el desarrollo de esta investigación:

2.1.2.1. Geometría (Modelado 3D)

- Vértices, Aristas y Caras: Son los componentes básicos de los modelos tridimensionales. Los vértices son puntos en el espacio tridimensional, las aristas son líneas que conectan los vértices, y las caras son superficies planas delimitadas por aristas.
- Mallas: Las mallas son colecciones de vértices, aristas y caras que forman la estructura del modelo tridimensional.
- Superficies y Volúmenes: Los objetos tridimensionales están definidos por superficies y, en algunos casos, por volúmenes que determinan su forma y tamaño en el espacio tridimensional.

2.1.2.2. Cámaras

- Punto de Vista: La cámara determina desde dónde se observa la escena tridimensional. La posición, orientación y los parámetros de la cámara (campo de visión, distancia focal) afectan cómo se renderiza la escena.
- Proyección: Hay dos tipos principales de proyección:
    - Proyección en Perspectiva: Simula la forma en que los objetos parecen más pequeños a medida que se alejan del espectador.
    - Proyección Ortográfica: No tiene perspectiva; los objetos mantienen su tamaño independientemente de la distancia.

2.1.2.3. Materiales y Texturas

- Materiales: Definen cómo la superficie de un objeto interactúa con la luz. Incluyen propiedades como la reflectancia, transparencia, y el índice de refracción.
- Texturas: Imágenes 2D aplicadas a las superficies de los modelos tridimensionales para agregar detalles como color, patrones, y relieve. Las texturas pueden ser mapas de color, mapas de normales, mapas de desplazamiento, entre otros.

2.1.2.5. Transformaciones

- Traslación: Mover objetos a diferentes posiciones en el espacio tridimensional.
- Rotación: Girar objetos alrededor de un eje en el espacio tridimensional.
- Escalado: Cambiar el tamaño de los objetos en una o más dimensiones.

2.1.2.6. Animación

- Simulación Física: Simula efectos como gravedad, colisiones, y dinamismo de fluidos o partículas para dar realismo al entorno.

2.1.2.7. Entorno y Fondos

- Cielos y Horizontes: Imágenes envolventes que simulan el cielo o el entorno distante.

2.1.2.8. Interacción

- Colisiones y Física: Gestionan cómo los objetos interactúan entre sí y con el entorno.
- Input del Usuario: Para entornos interactivos, es necesario gestionar las entradas del usuario, como el movimiento de la cámara o la manipulación de objetos.

2.1.2.9 NPC - Personaje No Jugador

Un NPC, o Non-Player Character (Personaje No Jugador en español), es un personaje en un videojuego o en una simulación que no es controlado por el jugador. En lugar de eso, los

NPCs son controlados por el sistema del juego o por la inteligencia artificial (IA). Su propósito puede variar dependiendo del diseño del escenario.

2.1.3 Fotogramas por segundo (FPS)

Es la tasa a la cual un sistema gráfico genera y despliega imágenes secuenciales, representando la frecuencia de actualización visual en una unidad de tiempo. Se calcula como el número de fotogramas individuales que se procesan, renderizan y presentan por segundo.

En el contexto de gráficos por computadora y animación en tiempo real los FPS son cruciales porque determinan la fluidez visual de la experiencia del usuario. Los FPS altos (como 60 FPS o más) generalmente proporcionan una experiencia más suave y visualmente agradable, mientras que FPS más bajos pueden resultar en una animación entrecortada o menos receptiva.

Donde el "Tiempo por fotograma" es el tiempo que tarda el sistema en completar el procesamiento y renderizado de un solo fotograma.

En sistemas gráficos, mantener altos FPS es importante para asegurar una experiencia de usuario óptima, especialmente en aplicaciones interactivas o de realidad virtual.

3 Hipótesis

La implementación de un algoritmo de búsqueda de trayectorias, utilizando un entorno bidimensional con elementos gráficos tridimensionales, el cuál sea eficiente en el uso de los recursos de la computadora. Para lograr esto, se realiza una evaluación continua de la memoria y del tiempo de CPU utilizados durante la ejecución del algoritmo, lo que evita desbordamientos de memoria y congelamiento de pantalla mientras se realizan los cálculos de búsqueda de trayectorias.

Se resolverán escenarios complejos, utilizando los algoritmos de búsqueda de trayectorias, tomando en cuenta variables como los puntos a recorrer, personajes jugadores, personajes autónomos y obstáculos. La optimización de los algoritmos de búsqueda de trayectorias permitirá que, al aumentar la complejidad del escenario, se pueda consumir de manera eficiente los recursos de la máquina en la que se está ejecutando este algoritmo.

4 Objetivos

La implementación de un algoritmo de optimización de búsqueda de trayectorias permitirá descubrir una solución eficiente que optimice los recursos necesarios para encontrar la mejor trayectoria según los criterios definidos. Este algoritmo permitirá actualizar constantemente la mejor trayectoria, considerando el movimiento de personajes, obstáculos y elementos autónomos donde la longitud de la trayectoria sea la mínima en un escenario establecido.

4.1 Objetivos Específicos

- Implementar un algoritmo que sea capaz de disminuir el consumo de recursos computacionales, para que tenga una mejor respuesta ante el movimiento de personajes autónomos. De esta manera será eficiente aun cuando existan escenarios complejos.

- Implementar el algoritmo en un escenario bidimensional con elementos gráficos tridimensionales, que permita la evaluación y verificación de las variables involucradas en la búsqueda de trayectorias, como son el tiempo de respuesta, el consumo de recursos ante el desplazamiento de personajes jugadores y personajes autónomos y la evasión de obstáculos.

- Desarrollar un entorno qué sea escalable tomando en cuenta las variables cómo puntos a recorrer, personajes jugadores, personajes autónomos y obstáculos.

5 Material y Métodos

En esta investigación los recursos materiales a utilizar son:

5.1 Hardware

Los experimentos que se muestran en esta investigación se implementan en una computadora con un procesador Intel Core i5-3570 a 3,40 GHz de 64 bits y 32 GB de RAM a 1660 MHz.

5.2 Software

Los algoritmos se implementan en un sistema operativo Windows en su versión 10.0.018363, utilizando el lenguaje de programación Python versión 3.9.7, la biblioteca psutil versión 5.9.0, la biblioteca tracemalloc versión 0.9, la biblioteca Pygame versión 2.0.2 y para la creación del entorno bidimensional con elementos gráficos tridimensionales, la librería Ursina en su versión 4.1.1.

Durante esta investigación, se implementan algoritmos en escenarios sin obstáculos con medidas de 10x10 bloques, 20x20 bloques, 30x30 bloques y 50x50 bloques. Posteriormente, los experimentos se realizan con escenarios del mismo tamaño, agregando obstáculos para medir el desempeño de los algoritmos.

5.2.1 Librería Psutil

Psutil es una biblioteca en Python que proporciona una interfaz para acceder a la información sobre el uso del sistema, incluyendo la CPU y la memoria.

Porcentaje de CPU: El método psutil.cpu_percent mide el uso de la CPU como un porcentaje del tiempo total de CPU que está siendo utilizado por todos los procesos en el sistema[24].

Virtual Memory: El método *psutil.virtual_memory()* proporciona un resumen del uso de la memoria RAM en el sistema. Devuelve un objeto con varios atributos, como total, disponible, porcentaje, usado, y libre.

Para medir el uso específico de la memoria y la CPU de un proceso en particular, como un script de Python, psutil ofrece funciones que permiten monitorear un proceso en tiempo real o realizar mediciones precisas en momentos determinados. Esto se puede hacer usando su ID de proceso (PID).

5.2.2 Tracemalloc

Tracemalloc es una biblioteca de Python diseñada para rastrear la asignación de memoria en el programa. Permite a los desarrolladores identificar cuánta memoria se está utilizando, dónde se realizan las asignaciones y encontrar posibles fugas de memoria.[25]

Tracemalloc rastrea las asignaciones de memoria y mantiene un historial de los tamaños de bloques de memoria asignados, así como de dónde se realizaron estas asignaciones en el código (las "trazas" o "snapshots"). Esto es especialmente útil para identificar problemas de memoria en aplicaciones grandes o complejas.
Snapshots: tracemalloc permite capturar instantáneas (snapshots) del estado de la memoria en un momento dado. Estas instantáneas pueden compararse para ver cómo ha cambiado el uso de memoria entre dos puntos del programa.

5.3 Criterios de Evaluación

Para evaluar el rendimiento del algoritmo, se ejecutan pruebas en diversos escenarios. Durante estas pruebas, se analizan varias métricas clave:

1. **Número de Bloques Computados**: Representa la cantidad de nodos o celdas en la malla para los cuales se calculó una métrica, como la distancia o el coste, durante la búsqueda de la trayectoria. Los bloques se dividen en dos categorías:
    - **Lista Abierta**: Conjunto de bloques que están pendientes de ser evaluados en términos de coste o heurística.
    - **Lista Cerrada**: Conjunto de bloques para los cuales ya se ha completado el cálculo y no serán revisados nuevamente.
2. **Pasos en la Trayectoria Encontrada**: Esta métrica contabiliza el número de bloques o celdas que conforman el camino final desde el punto de inicio hasta el objetivo. Cada bloque en la trayectoria representa un "paso" hacia la meta.
3. **Tiempo de Ejecución**: Tiempo total, medido en segundos, que el algoritmo tarda en finalizar la búsqueda de la trayectoria óptima desde el inicio hasta el punto objetivo.
4. **Uso de Memoria**: Consumo de memoria RAM, medido en megabytes (MB), durante la ejecución del algoritmo. Este valor se promedia para reflejar el uso típico de memoria a lo largo de la ejecución.
5. **Uso de CPU**: Porcentaje promedio de utilización de la CPU durante la ejecución del algoritmo. Esta métrica refleja la carga de trabajo que el proceso impone sobre el procesador.

Las pruebas se realizan inicialmente en escenarios sin obstáculos para establecer una línea base del comportamiento de los algoritmos. Posteriormente, se introducen escenarios con obstáculos para evaluar el impacto de las condiciones más complejas en el rendimiento del algoritmo.

Para realizar una evaluación exhaustiva del rendimiento de los algoritmos de búsqueda de trayectorias, se diseñó un conjunto de pruebas que comprende un total de 300 iteraciones por escenario. Se realizan pruebas considerando 3 escenarios diferentes, cada prueba se repite 100 veces, para obtener un promedio representativo de la ejecución, lo que permite minimizar la variabilidad en los resultados y asegurar una mayor precisión en la evaluación. Cuando existe un escenario en el cual no existe una solución se genera un nuevo escenario hasta encontrar uno que si tenga solución.

Las pruebas se estructuran en tres escenarios distintos:

1. **Escenario Fijo**: En este escenario, tanto el punto de inicio como el punto de destino están predefinidos. El punto de inicio se ubica en la celda (0,0) y el punto de destino en la celda (0,9). Este escenario sirve como referencia para analizar el rendimiento en una trayectoria con parámetros completamente controlados.

2. **Escenario de Esquinas Opuestas**: Aquí, el punto de inicio permanece en (0,0), mientras que el punto de destino se ubica en la esquina opuesta de la malla, es decir, en (n-1, n-1). Esta prueba tiene como objetivo evaluar el rendimiento de los algoritmos en la búsqueda de trayectorias que cubren la máxima distancia posible dentro de la malla.

3. **Escenario Aleatorio**: En este caso, tanto el punto de inicio como el punto de destino se seleccionan de manera aleatoria dentro de la malla. Este escenario simula condiciones menos predecibles, donde los puntos de interés pueden variar de manera no determinística.

Las pruebas iniciales se ejecutan en mallas sin obstáculos, con tamaños que varían desde 10x10 hasta 500x500 bloques. Posteriormente, se realizan pruebas con obstáculos generados de dos maneras, la primera, definidas por el usuario al crear el escenario; la segunda donde se generan de forma aleatoria, siempre tomando en cuenta que exista una trayectoria desde el punto de inicio hasta el punto final, donde el porcentaje de ocupación de estos se incrementa de 10 en 10, comenzando en un 10% hasta alcanzar un 50% del área total de la malla.

Durante todas estas pruebas, se recopilan los datos que incluyen el número de bloques computados, los pasos necesarios para completar la trayectoria encontrada, el tiempo de ejecución, el uso de memoria (medido en megabytes) y el porcentaje de utilización de la CPU. Este conjunto de métricas proporciona una visión detallada y cuantificable del comportamiento de los algoritmos en diversas condiciones, permitiendo comparaciones rigurosas entre ellos.

5.4 Diagrama de metodología

En esta sección se presenta el diagrama de metodología a utilizar durante esta investigación, que describe las etapas en las cuales se divide la búsqueda, selección e implementación de los algoritmos de búsqueda de trayectorias.

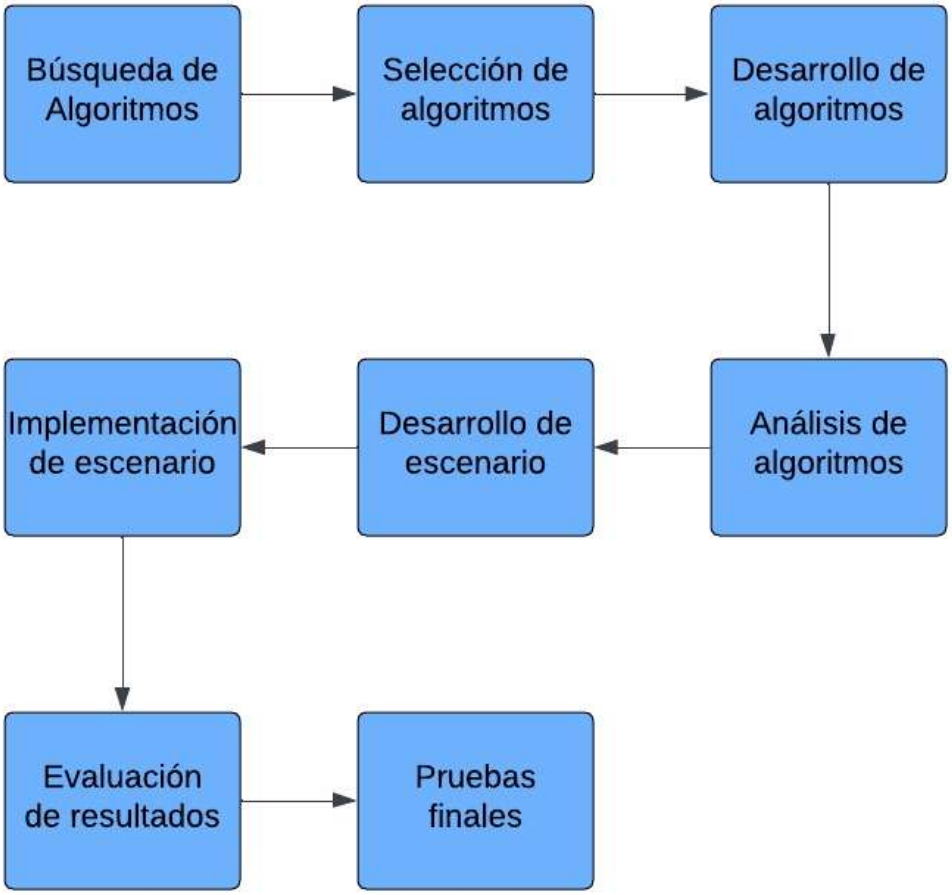

Figura 5.1 Diagrama de flujo de la metodología a seguir.

1. Realizar una investigación sobre los algoritmos utilizados para la búsqueda de trayectorias:

Se llevará a cabo una investigación exhaustiva de artículos, libros e investigaciones donde se hayan implementado algoritmos de búsqueda de trayectorias.

2. Selección de los algoritmos a utilizar:

Se realizará una selección de los algoritmos que han sido implementados para la búsqueda de trayectorias, DFS, BFS, Dijkstra y A*, teniendo en cuenta sus ventajas específicas para resolver problemas en grafos, como la exploración exhaustiva, la garantía de encontrar caminos más cortos, y la eficiencia en la búsqueda guiada por heurísticas.

3. Desarrollo e implementación de los algoritmos:

Una vez seleccionados los algoritmos que han presentado mejores resultados, se procederá a implementarlos en la plataforma de desarrollo tridimensional.

4. Análisis de los algoritmos implementados:

Una vez optimizados, adaptados e implementados los algoritmos, se llevará a cabo un análisis de su desempeño, teniendo en cuenta el tiempo de respuesta, el uso de CPU, la longitud de la trayectoria, el número de personajes, el tamaño del escenario y los puntos a recorrer.

5. Desarrollo de un escenario en un entorno 3D con los algoritmos implementados:

Se desarrollarán escenarios que serán resueltos por los algoritmos, los cuales contarán con diferentes tamaños, puntos a recorrer, personajes involucrados y diversos obstáculos.

6. Implementación de los algoritmos en los escenarios 3D:

Se implementarán los algoritmos desarrollados en los escenarios creados, realizando un análisis de los resultados de los algoritmos.

7. Evaluación de los algoritmos y su desempeño en el escenario:

Se realizará una evaluación de los resultados obtenidos por cada algoritmo implementado, después de su optimización.

8. Optimización de los algoritmos:

Se seleccionará el algoritmo que haya obtenido los mejores resultados durante las pruebas.

9. Pruebas finales:

Se evaluará el algoritmo optimizado en diferentes escenarios. Si es necesario, se regresará al punto 8 para realizar los ajustes necesarios.

5.5 Creación de escenarios en dos dimensiones

En el proceso de generación de escenarios en dos dimensiones, se busca que los algoritmos sean evaluados en diferentes situaciones. Se utilizan los mismos escenarios para los cuatro algoritmos evaluados, y se toman en cuenta los criterios de evaluación desde el inicio del programa hasta que se muestra en pantalla la trayectoria encontrada.

La figura siguiente, figura 5.2, muestra algunos escenarios para la evaluación de los algoritmos utilizados. El escenario de 10x10 bloques es el más pequeño, mientras que el de 80x80 bloques es el más grande. Sin embargo, los escenarios de 60x60 bloques y 80x80 bloques no se utilizan para la evaluación debido a que se dificulta la visualización de los escenarios y de las trayectorias encontradas.

Cabe destacar que, al realizar la generación del escenario con elementos gráficos tridimensionales, se pueden implementar escenarios en los cuales no sea necesaria la visualización de las trayectorias. De esta forma, se pueden evaluar los algoritmos de manera más precisa y eficiente.

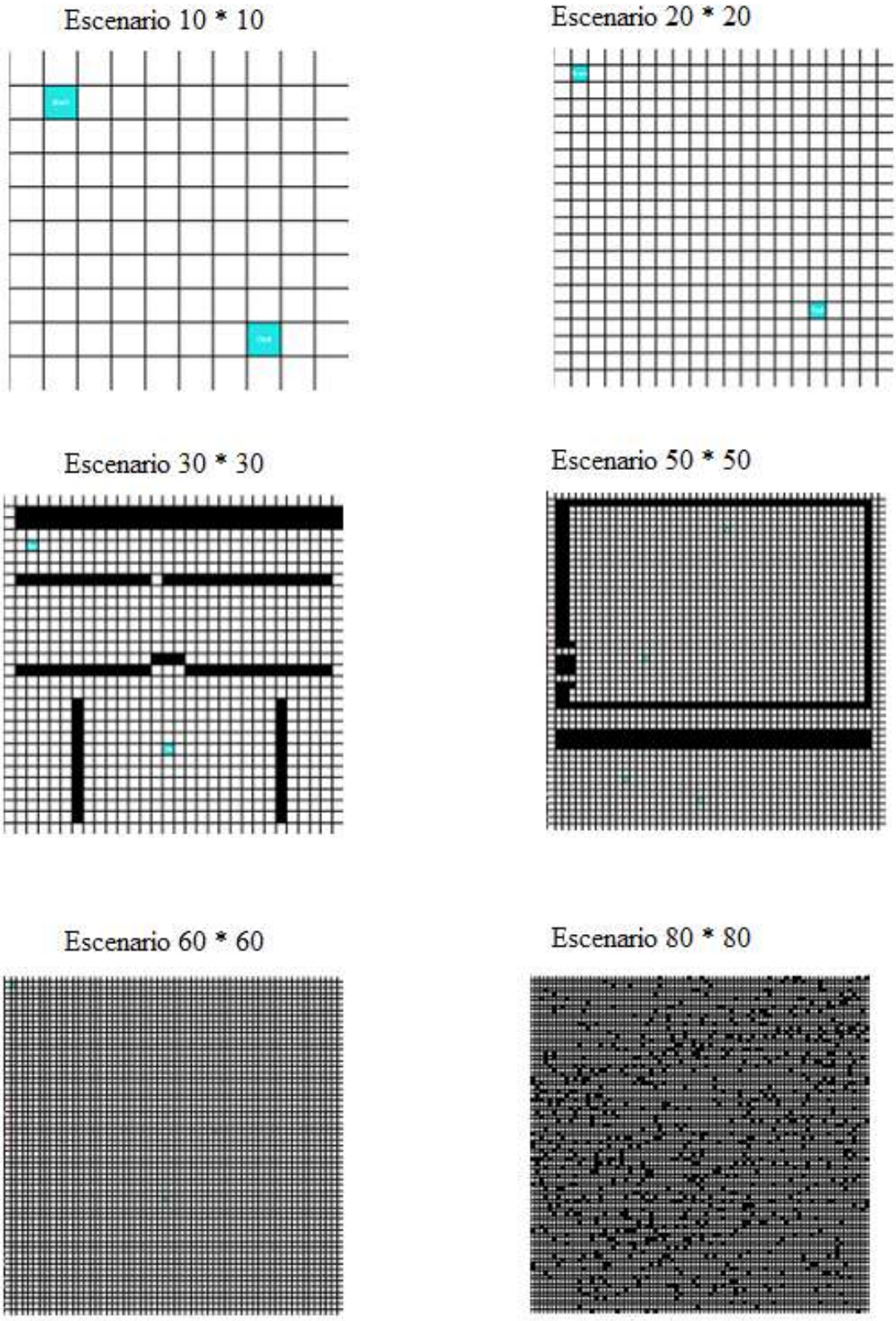

Figura 5.2 Ejemplos de tamaño de los escenarios

5.6 Mapeo de malla a un grafo

El mapeo de una malla a un grafo implica representar cada celda de la malla como un nodo en el grafo y establecer aristas entre nodos vecinos según las conexiones en la malla.

Nodos: Cada bloque en la malla se convierte en un nodo del grafo.

Aristas: Las aristas son las conexiones entre nodos. En la malla, cada bloque está conectado a sus bloques vecinos (arriba, abajo, izquierda, derecha), si no están en los bordes.

Conexiones: Si se toma un bloque que no está en el borde, por ejemplo, el bloque en la posición (5, 5), se conectará a los bloques en (4, 5), (6, 5), (5, 4), y (5, 6).
Para un bloque en un borde, como (0, 0), solo se conectará a los bloques (0, 1) y (1, 0).

Propósito: Este mapeo es útil para problemas de búsqueda de caminos, simulaciones, o cualquier problema donde necesites representar la relación entre puntos cercanos en una cuadrícula.

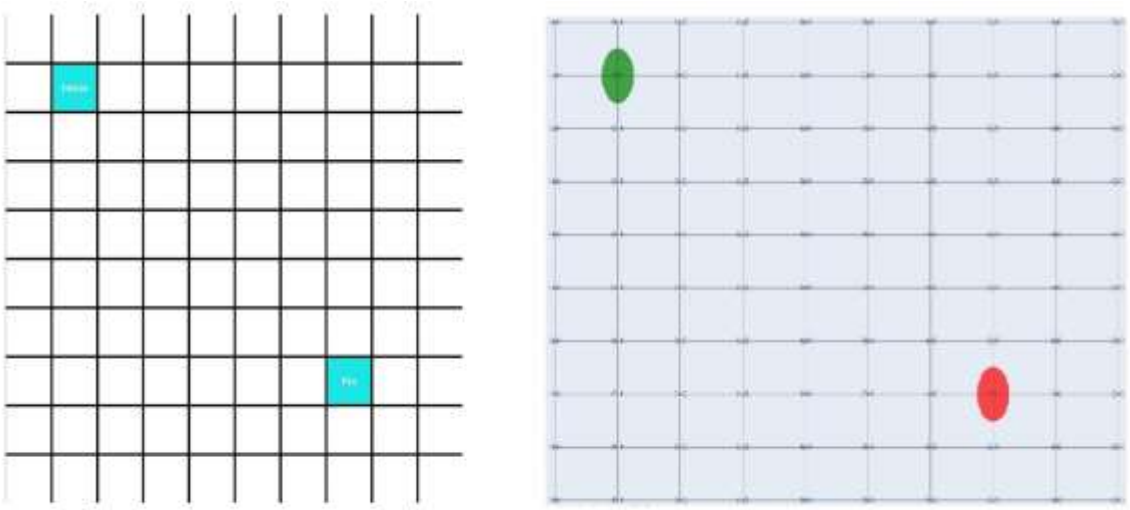

Figura 5.3 Malla y grafo 10x10 sin obstáculos

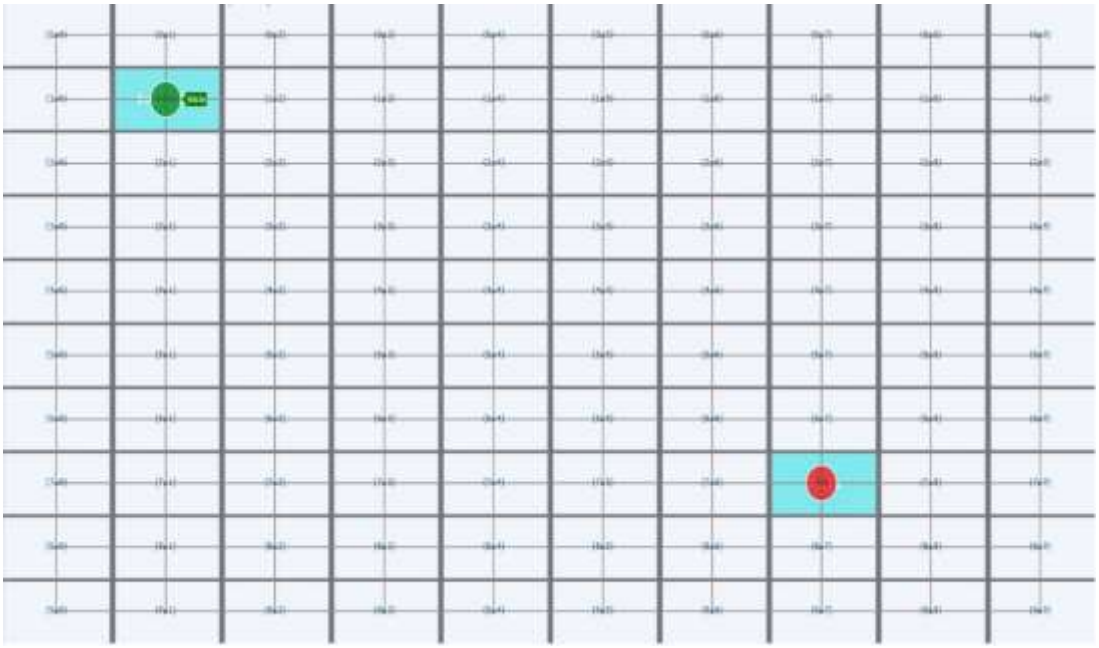

Figura 5.4 Grafo superposicionado de malla 10x10 sin obstáculos

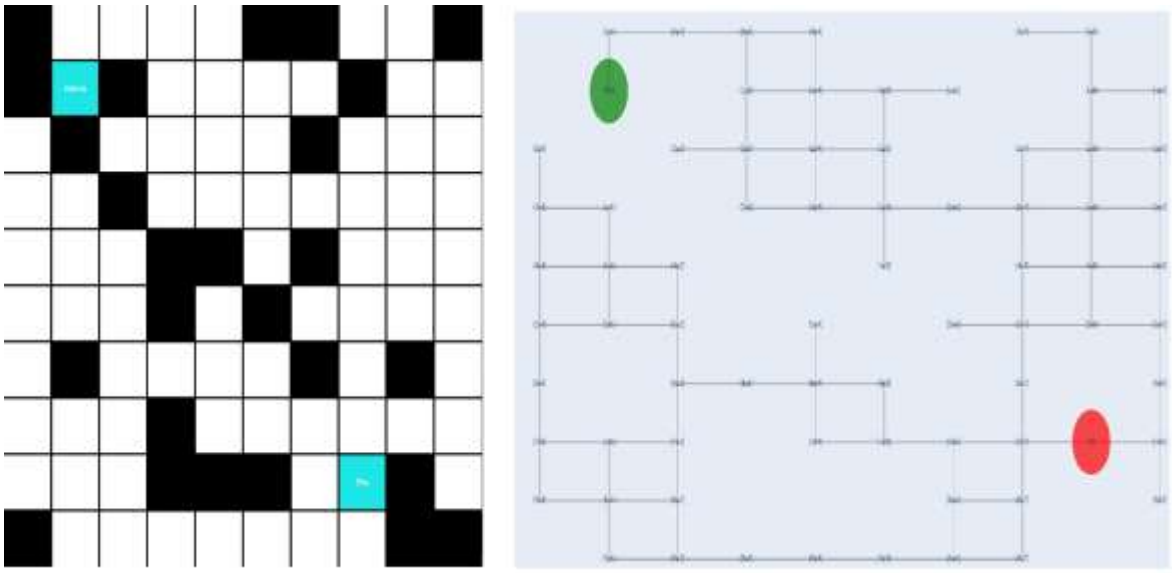

Figura 5.5 Malla y grafo 10x10 con obstáculos

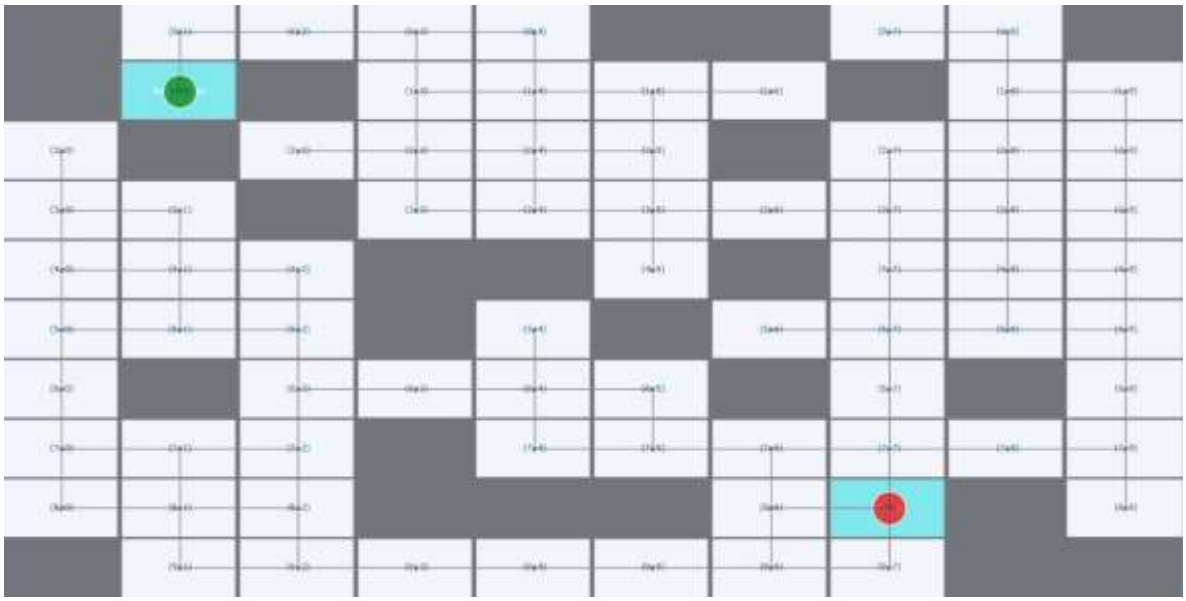

Figura 5.6 Grafo superposicionado de malla 10x10 con obstáculos

6 Resultados y discusión

Los algoritmos DFS, BFS, el algoritmo de Dijkstra y A* están implementados en el lenguaje Python con el apoyo de la librería Pygame en un entorno 2D. Posteriormente, los experimentos se realizaron utilizando el motor gráfico Ursina. Para todos los experimentos, los criterios de evaluación utilizados incluyen el número de bloques computados, los pasos requeridos en la trayectoria encontrada, el tiempo de ejecución, el uso de memoria en megabytes y el porcentaje de uso de la CPU.

En la primera etapa de los experimentos, se construyen escenarios sin obstáculos utilizando mallas cuadradas de diferentes tamaños, aumentando el tamaño de los escenarios hasta llegar a una malla de 500x500 bloques. El objetivo es analizar el comportamiento de los algoritmos y el consumo de recursos computacionales en escenarios sin obstáculos.

En la segunda etapa de los experimentos, se introduce la presencia de obstáculos en los escenarios previamente construidos. Al incorporar obstáculos en el entorno, se pretende evaluar cómo los algoritmos se comportan y adaptan ante situaciones donde la ruta hacia el destino está obstruida. La inclusión de obstáculos añade una capa adicional de complejidad al análisis, permitiendo una evaluación más completa del desempeño y la eficacia de los algoritmos en escenarios más complejos.

En cada prueba realizada, se presenta una tabla comparativa que muestra el rendimiento de cada algoritmo en el escenario. Estas tablas nos permiten analizar los algoritmos y tomar decisiones sobre cuál sería el más adecuado para su implementación en un entorno con elementos gráficos tridimensionales. Además, se incluye una imagen que representa un ejemplo de los caminos encontrados por cada algoritmo. Esta representación visual nos ayuda a comprender mejor los bloques computados, los bloques presentes en las listas abierta y cerrada, así como el camino encontrado por cada algoritmo durante su ejecución.

6.1 Escenarios sin obstáculos

En los escenarios sin obstáculos, se presentan entornos abiertos y libres de cualquier obstrucción o impedimento. Estos escenarios proporcionan un espacio amplio y claro donde los algoritmos pueden operar sin restricciones, permitiendo una evaluación más directa de su rendimiento y eficacia en la búsqueda de caminos. Los escenarios sin obstáculos ofrecen la oportunidad de analizar cómo los algoritmos navegan a través del espacio disponible para encontrar la ruta más corta desde el punto de inicio hasta el punto de destino.

6.1.1 Escenario 10x10

En el marco del primer experimento, correspondiente a la fase inicial de la investigación, se realizó un análisis detallado de un escenario compuesto por una malla de 10x10 bloques. Se realizaron 100 iteraciones considerando 3 escenarios diferentes. Los promedios de los resultados obtenidos en esta implementación se presentan en la Tabla 6, donde se detallan las métricas de desempeño de los algoritmos evaluados en este contexto.

Tabla 6.1 Consumo de recursos computacionales de los algoritmos DFS, BFS, Dijkstra y A en un escenario de 10x10 bloques.*

Escenario 10 x 10	Algoritmos			
	DFS	BFS	Dijkstra	A*
Bloques Computados	75	98	99	22
Pasos realizados	34	7	7	7
Tiempo de ejecución (Segundos)	0.0163	0.0228	0.0147	0.0114
Uso de memoria (Megabits)	0.0072	0.0079	0.0086	0.0047
Porcentaje de CPU	5.0775 %	2.9553 %	3.17 %	13.5653 %

Durante la evaluación de los algoritmos implementados, la presencia de dos listas, la lista abierta y la lista cerrada, ocasiona que el recuento de bloques pueda llegar a ser el doble del número presente en el escenario, ya que el conteo de bloques computados suma los bloques en la lista cerrada y la lista abierta. El algoritmo A* es el que obtiene la menor cantidad de bloques computados, con un total de 22 bloques, mientras que los algoritmos BFS y Dijkstra son los que tienen la mayor cantidad de bloques computados, con un total de 98 y 99 respectivamente.

Cabe destacar que, a pesar de que los algoritmos BFS, Dijkstra y A* obtienen el mismo número de pasos para llegar al punto de interés, el algoritmo DFS es el que consume más tiempo en encontrar la trayectoria; mientras que el algoritmo A* es el que menos tarda en encontrar el punto de interés. Por su parte, el algoritmo BFS es el que consume el porcentaje más bajo de CPU, siendo el DFS el algoritmo que realiza la mayor cantidad de pasos para llegar al punto de interés.

En cuanto a la comparación de uso de memoria, se puede observar que los algoritmos de Dijkstra y BFS son los que consumen más memoria. Es importante destacar que estos resultados pueden variar dependiendo del escenario utilizado. Por ello, es necesario realizar diferentes pruebas en diferentes escenarios para evaluar el rendimiento de cada algoritmo.

A continuación, se muestra un ejemplo de una trayectoria encontrada por los algoritmos de búsqueda durante las pruebas para alcanzar el punto de interés:

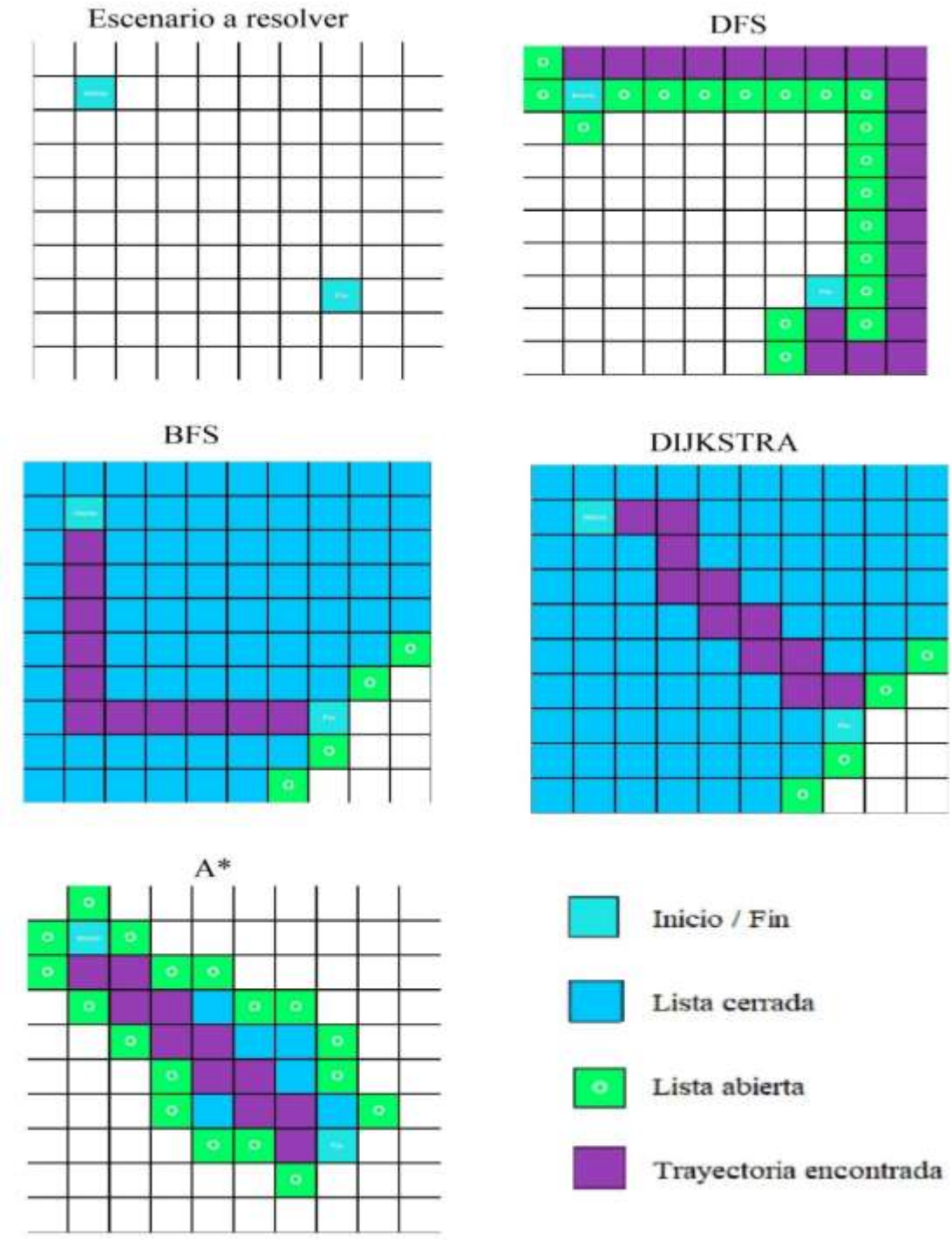

Figura 6.1 Bloques calculados y trayectorias encontradas para un escenario de 10x10 bloques sin obstáculos.

En la figura 6.1, se muestra como cada uno de los algoritmos que se evalúan en esta investigación, el DFS, BFS, Dijkstra y A*, encuentran su camino en el escenario de 10x10 bloques. El algoritmo DFS, comienza su recorrido por subir hacia los bloques superiores y posteriormente hacia la derecha, mientras que el BFS se expande entre nodos adyacentes al punto de partida, de manera similar al algoritmo de Dijkstra. Por otro lado, el algoritmo A* demuestra ser el más eficiente al mostrar el camino más corto con menos bloques en su lista cerrada. La reducción en el número de bloques computados tiene un impacto significativo en el tiempo de ejecución, en este caso, resultando en el menor tiempo de ejecución en comparación con los otros algoritmos.

6.1.2 Escenario 20x20

Para dar continuidad a la evaluación de los algoritmos y preparar su implementación, llevamos a cabo el segundo experimento de la primera etapa, esta vez utilizando un escenario en una malla de 20x20 en lugar de 10x10. Esta ampliación nos brindará una perspectiva más detallada y completa del desempeño de los algoritmos en un entorno de mayor tamaño, lo que proporciona datos adicionales que podrían influir en la selección final del algoritmo para su implementación.

Tabla 6.2 Comparación del consumo de recursos computacionales de los algoritmos DFS, BFS, Dijkstra y A en un escenario de 20x20 bloques.*

Escenario 20 x 20	Algoritmos			
	DFS	BFS	Dijkstra	A*
Bloques Computados	573	536	540	142
Pasos realizados	245	21	21	21
Tiempo de ejecución (Segundos)	0.0593	0.1087	0.0622	0.0202
Uso de memoria (Megabits)	0.0421	0.0385	0.04361	0.0123
Porcentaje de CPU	1.4351 %	1.7622 %	1.8050 %	5.1161 %

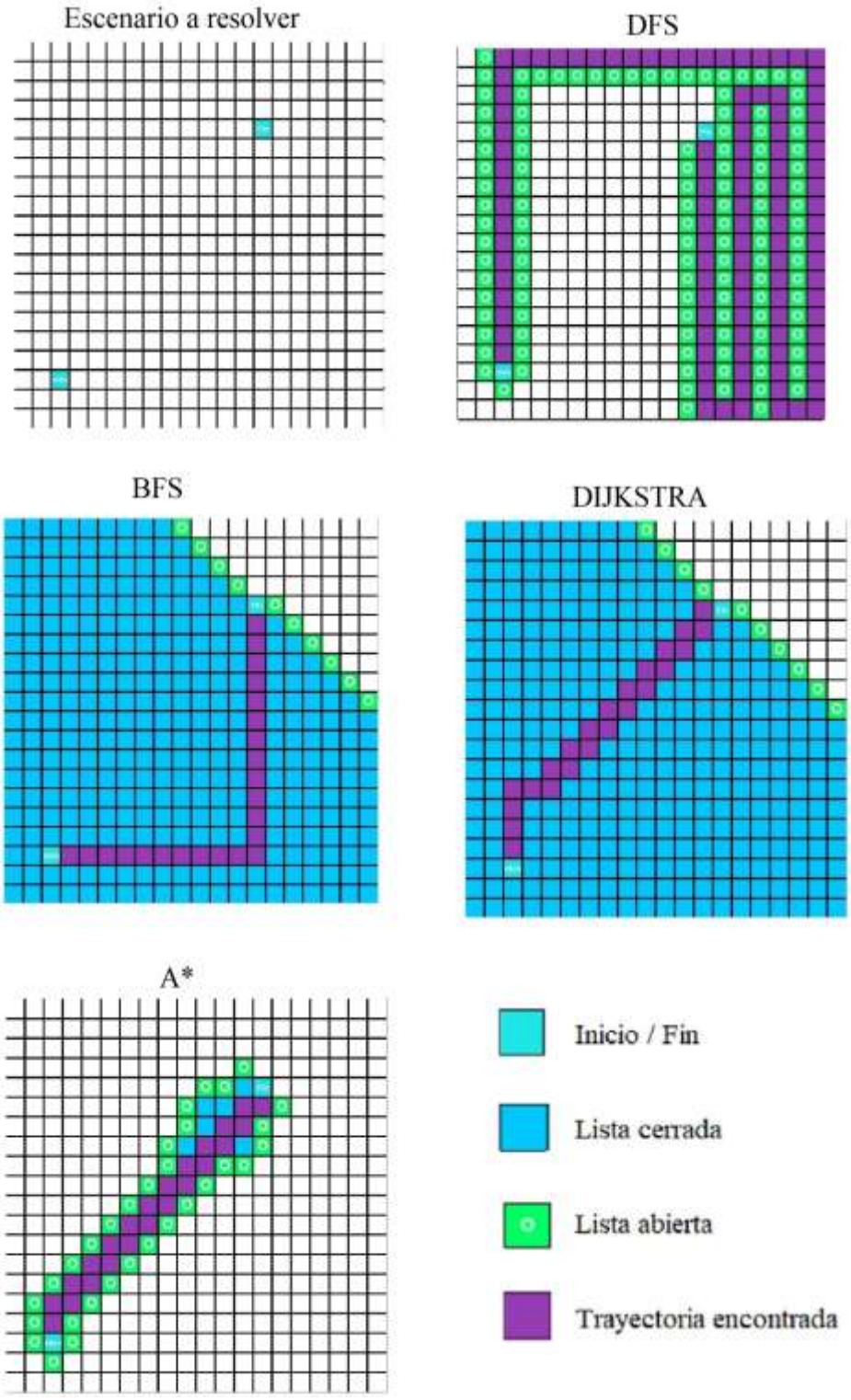

Figura 6.2 Bloques calculados y trayectorias encontradas para un escenario de 20x20 bloques sin obstáculos.

En este escenario, se observa que el algoritmo DFS computa una cantidad significativamente mayor de bloques (573) en comparación con los pasos realizados (245) para alcanzar el punto de interés, lo que indica una mayor exploración innecesaria. Por otro lado, los algoritmos BFS y Dijkstra computan un número similar de bloques (536 y 540, respectivamente) y realizan la misma cantidad de pasos (21) para llegar al destino, aunque a través de caminos ligeramente diferentes. El algoritmo A*, aunque computa muchos menos bloques (142), logra encontrar la misma trayectoria óptima en términos de pasos (21), destacándose por su eficiencia.

En cuanto al tiempo de ejecución, A* demuestra ser el más rápido con 0.0202 segundos, requiriendo una fracción del tiempo comparado con DFS (0.0593 segundos), BFS (0.1087 segundos) y Dijkstra (0.0622 segundos).

Respecto al uso de memoria, todos los algoritmos muestran un consumo bajo, con A* destacándose nuevamente como el más eficiente con solo 0.0123 MB, en contraste con DFS, que utiliza 0.0421 MB. Sin embargo, el algoritmo A* registra el mayor porcentaje de uso de CPU (5.1161 %), superando significativamente a los demás algoritmos, que mantienen un uso de CPU alrededor del 1.7 %.

6.1.3 Escenario 30x30

Para el tercer experimento se utiliza una malla de 30x30 bloques. En este escenario se espera observar un aumento en los recursos computacionales utilizados por cada algoritmo, así como analizar si existe algún patrón de comportamiento entre el tamaño del escenario y los recursos computacionales utilizados por los algoritmos. En la siguiente tabla se muestran los resultados del tercer experimento.

Tabla 6.3 Comparación del consumo de recursos computacionales de los algoritmos DFS, BFS, Dijkstra y A en un escenario de 30x30 bloques.*

Escenario 30 x 30	Algoritmos			
	DFS	BFS	Dijkstra	A*
Bloques Computados	992	1032	1027	213
Pasos realizados	360	23	23	23
Tiempo de ejecución (Segundos)	0.1153	0.2154	0.1197	0.0279
Uso de memoria (Megabits)	0.0548	0.0587	0.0653	0.0188
Porcentaje de CPU	2.0110 %	1.9668 %	2.3244 %	4.9931 %

En cuanto a los algoritmos de Dijkstra, DFS y BFS, se puede observar que el número de bloques calculados es considerablemente alto en comparación con el algoritmo A*. Además, en este tercer escenario, se puede notar un aumento en el uso de memoria en los algoritmos en comparación con los otros escenarios. El algoritmo A* sigue siendo el que consume menos recursos, y sus bloques computados siguen siendo menores.

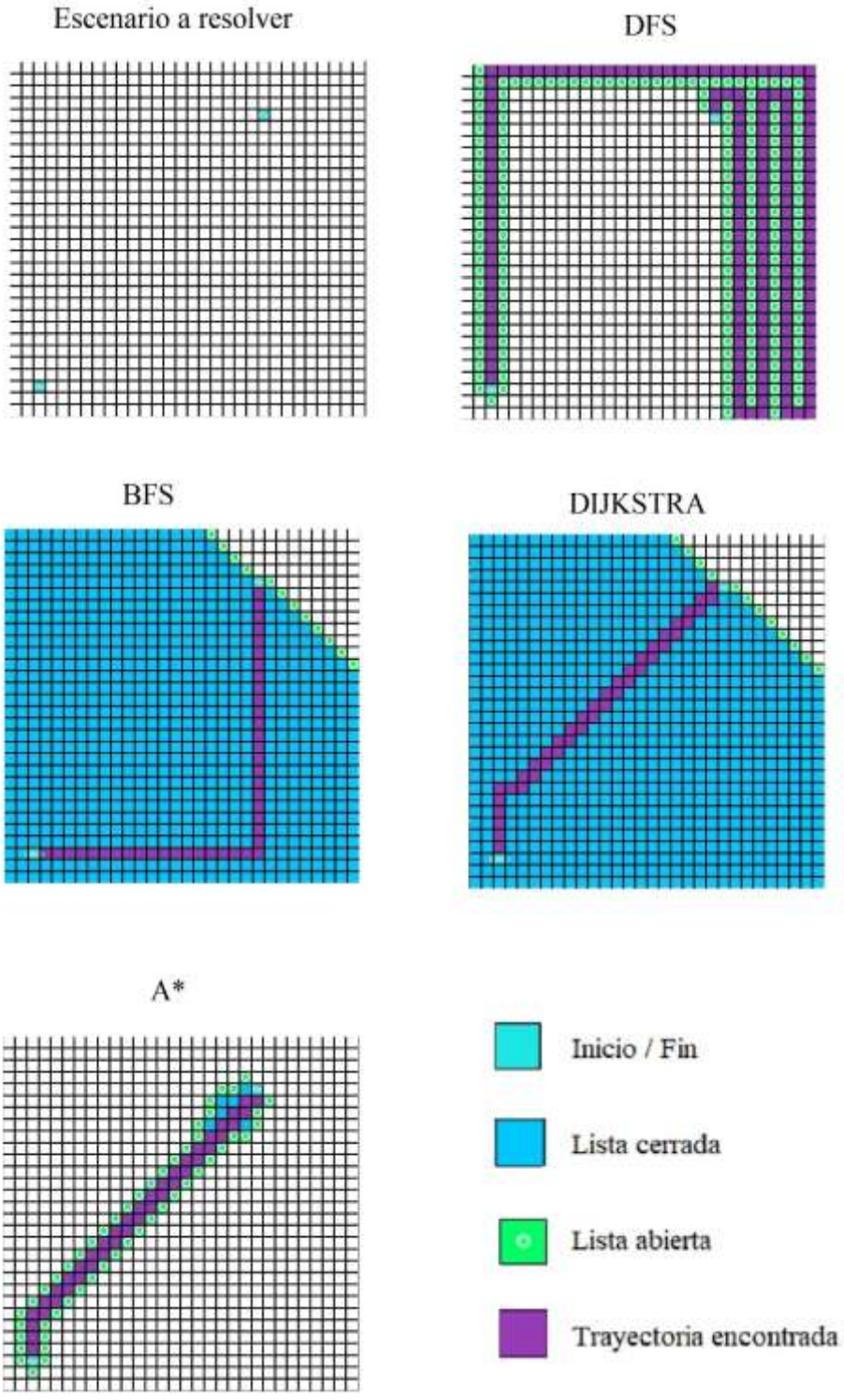

Figura 6.3 Bloques calculados y trayectorias encontradas para un escenario de 30x30 bloques sin obstáculos.

En la figura 6.3, se presenta un ejemplo de los caminos encontrados por los algoritmos de búsqueda de trayectorias, en esta figura es perceptible que el algoritmo A* es el que menos bloques tiene en su lista cerrada, y que los algoritmos de Dijkstra y BFS mantienen un comportamiento similar con los mismos bloques computados, aunque teniendo diferentes caminos encontrados.

6.1.4 Escenario 50x50

En el cuarto escenario, se pronostica que, al ser un entorno más extenso, esto genere nuevamente un aumento del uso de los recursos computacionales. Los resultados se presentan en la siguiente tabla:

Tabla 6.4 Comparación del consumo de recursos computacionales de los algoritmos DFS, BFS, Dijkstra y A en un escenario de 50x50 bloques.*

Escenario 50 x 50	Algoritmos			
	DFS	BFS	Dijkstra	A*
Bloques Computados	1911	2250	2239	529
Pasos realizados	758	29	29	29
Tiempo de ejecución (Segundos)	0.2137	0.4838	0.2615	0.0659
Uso de memoria (Megabits)	0.1248	0.1368	0.1513	0.0431
Porcentaje de CPU	1.5428 %	1.7795 %	1.8096 %	4.7552 %

Aunque los cuatro algoritmos muestran un uso de CPU y memoria relativamente similar, se observan diferencias notables en los bloques computados, los pasos realizados y el tiempo de ejecución. En particular, el algoritmo A* se destaca por su eficiencia en términos de tiempo de ejecución, requiriendo solo 0.0659 segundos, lo que lo convierte en el más rápido entre los algoritmos evaluados. A pesar de que A* también presenta un uso de memoria algo mayor (0.0431 MB) y un porcentaje de CPU (4.7552 %) superior al de DFS, BFS y Dijkstra, el tiempo de ejecución reducido resalta su eficacia en el procesamiento.

Por otro lado, el algoritmo DFS computa la mayor cantidad de bloques (1911) y realiza un número considerable de pasos (758), lo que resulta en un tiempo de ejecución de 0.2137 segundos. BFS y Dijkstra, aunque computan una cantidad mayor de bloques en comparación

con A*, realizan el mismo número de pasos (29) y tienen tiempos de ejecución intermedios, con BFS requiriendo 0.4838 segundos y Dijkstra 0.2615 segundos.

En resumen, a pesar de que A* tiene un mayor uso de memoria y CPU en comparación con los otros algoritmos, su capacidad para reducir significativamente el tiempo de ejecución lo hace destacar en escenarios con mallas de 50x50 bloques. Los algoritmos DFS, BFS y Dijkstra presentan un rendimiento más uniforme en términos de memoria y CPU, pero con tiempos de ejecución más largos y, en el caso de DFS, una mayor cantidad de bloques computados.

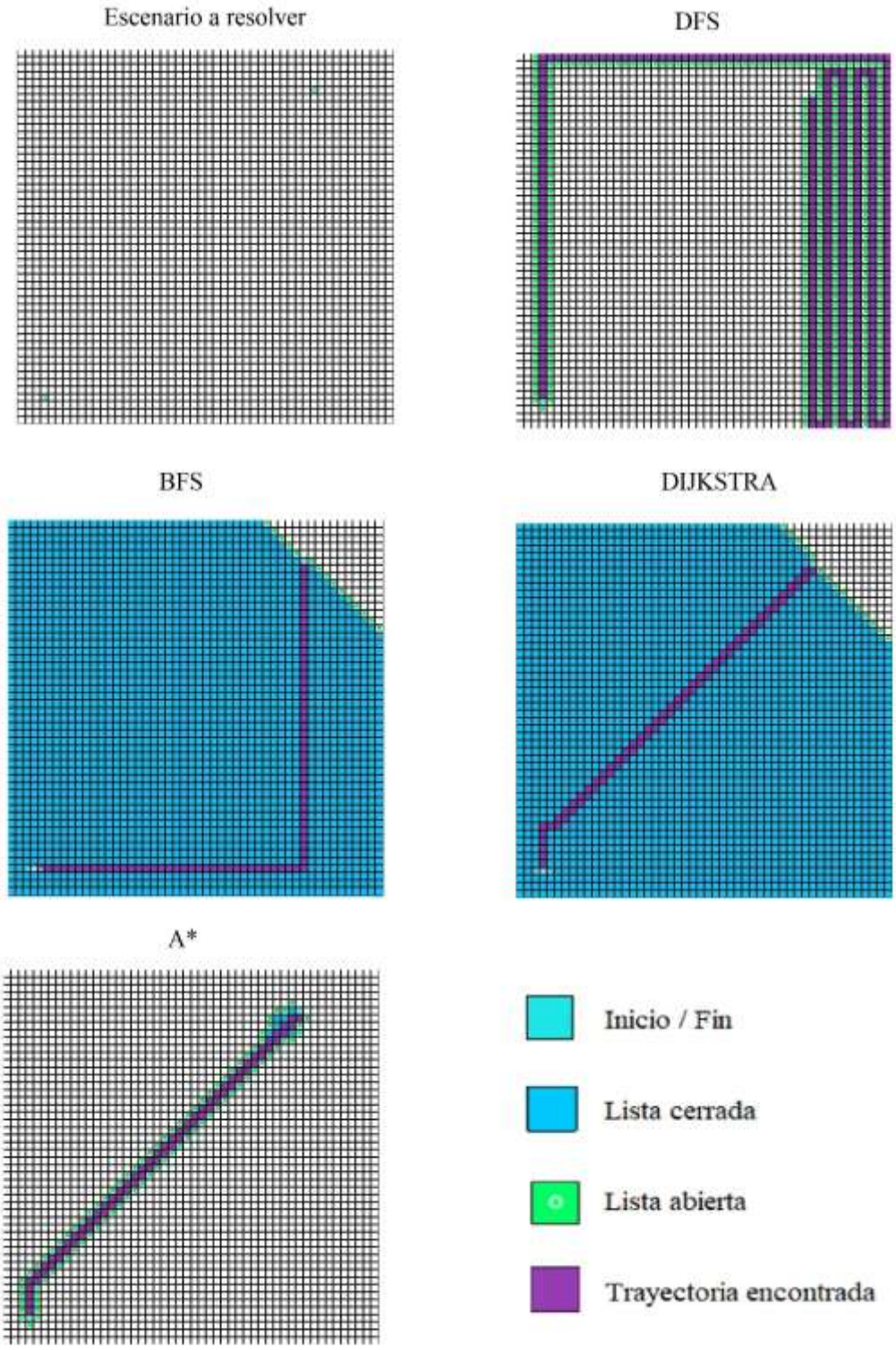

Figura 6.4 Bloques calculados y trayectorias encontradas para un escenario de 50x50 bloques sin obstáculos.

6.2 Escenarios con obstáculos

En esta etapa de la investigación, se analiza el impacto de la presencia de obstáculos en la búsqueda de trayectorias por los algoritmos de búsqueda. Los bloques negros en la malla representan obstáculos que bloquean el camino directo hacia el punto final, forzando a los algoritmos a encontrar rutas alternativas.

Los obstáculos pueden ser agregados de dos formas distintas según la intención del usuario y la fase de investigación. Durante las etapas iniciales, cuando se trabaja con mallas de dimensiones reducidas, resulta sencillo y práctico ubicar los obstáculos directamente mediante una interfaz gráfica, ya que la visualización en pantalla permite posicionarlos de manera intuitiva casilla por casilla. Sin embargo, a medida que aumenta el tamaño de la malla, este método manual presenta limitaciones como son: la representación gráfica pierde claridad al disminuir el tamaño visible de cada casilla, y el replicar patrones complejos se vuelve difícil y propenso a errores.

Posteriormente, se implementó un sistema de importación mediante archivos CSV que permite indicar las coordenadas cartesianas de cada obstáculo. Este enfoque no solo resuelve los problemas de escalabilidad, sino que aporta ventajas técnicas adicionales. Al cargar los obstáculos desde un archivo estructurado, se pueden replicar configuraciones exactas entre las diferentes pruebas, y lo que es más importante desde el punto de vista computacional, poder consolidar todos los elementos en un único objeto durante el proceso de generación de obstáculos, optimizando así el uso de memoria. El formato del archivo CSV sigue una estructura sencilla donde cada línea representa un obstáculo con sus coordenadas (x, y) y propiedades específicas, permitiendo una implementación rápida incluso en mallas de grandes.

Se espera que tanto el aumento en el tamaño de la cuadrícula como la introducción de obstáculos incrementen el consumo de los recursos computacionales necesarios para encontrar la trayectoria óptima. En las pruebas realizadas, se inició con un 10% de los bloques de la malla designados como obstáculos y se incrementó gradualmente hasta alcanzar un

50%. Este límite del 50% fue elegido para asegurar que los escenarios mantuvieran una complejidad considerable sin volverse insuperables, permitiendo así una evaluación efectiva del rendimiento de los algoritmos bajo diferentes condiciones. Cuando existe un escenario en el cual no existe una solución se genera un nuevo escenario hasta encontrar uno que si tenga solución.

El agregar obstáculos a los escenarios tiene varios propósitos, algunos de estos son:

- Evaluación del rendimiento:

  La presencia de obstáculos permite evaluar la capacidad de los algoritmos para sortear situaciones complejas. Esto ayuda a determinar qué tan eficientes y efectivos son los algoritmos en entornos más complejos.

- Pruebas de robustez:

  Al introducir obstáculos, se puede probar la robustez de los algoritmos frente a condiciones adversas. Esto es crucial para sistemas que deben operar en entornos dinámicos o desconocidos.

- Validación del comportamiento:

  Al agregar obstáculos, se valida el comportamiento de los algoritmos en escenarios más complejos, asegurando que respondan de manera adecuada y eficiente ante condiciones no ideales.

6.2.1 Escenario 10x10

Para este primer escenario, que consiste en una cuadrícula de 10x10 bloques, se asegura que cada algoritmo pueda encontrar al menos una trayectoria para llegar al punto final.

Tabla 6.5Comparación del consumo de recursos computacionales de los algoritmos DFS, BFS, Dijkstra y A en un escenario de 10x10 bloques con obstáculos.*

Algoritmo DFS					
Porcentaje de Obstáculos	Bloques Computados	Pasos realizados	Tiempo de ejecución (Segundos)	Uso de memoria (Megabits)	Porcentaje de CPU
10 %	107	33	0.0192	0.0093	3.3052 %
20 %	100	27	0.0183	0.0078	3.1279 %
30 %	80	16	0.0161	0.0065	3.9558 %
40 %	55	8	0.0138	0.0058	6.9820 %
50 %	28	4	0.0144	0.0049	16.2973 %

Algoritmo BFS					
Porcentaje de Obstáculos	Bloques Computados	Pasos realizados	Tiempo de ejecución (Segundos)	Uso de memoria (Megabits)	Porcentaje de CPU
10 %	88	7	0.0186	0.0075	3.1356 %
20 %	84	7	0.0185	0.0074	3.7424 %
30 %	73	8	0.0156	0.0065	4.2406 %
40 %	54	5	0.0156	0.0062	6.6798 %
50 %	33	3	0.0150	0.0053	10.7089 %

Algoritmo Dijkstra					
Porcentaje de Obstáculos	Bloques Computados	Pasos realizados	Tiempo de ejecución (Segundos)	Uso de memoria (Megabits)	Porcentaje de CPU
10 %	88	7	0.0151	0.0087	3.5566 %
20 %	78	7	0.0143	0.0083	4.2687 %
30 %	71	8	0.0143	0.0073	5.5915 %
40 %	61	5	0.0141	0.0068	5.4256 %
50 %	29	3	0.0145	0.0055	13.4450 %

Algoritmo A*					
Porcentaje de Obstáculos	Bloques Computados	Pasos realizados	Tiempo de ejecución (Segundos)	Uso de memoria (Megabits)	Porcentaje de CPU
10 %	27	7	0.0121	0.0053	13.2100 %
20 %	32	8	0.0133	0.0056	10.9302 %
30 %	55	7	0.0146	0.0062	8.8055 %
40 %	45	4	0.0149	0.0061	9.2414 %
50 %	28	2	0.0141	0.0053	15.4065 %

En este escenario de 10x10 con la inclusión de obstáculos, se observa un cambio significativo en el comportamiento de los algoritmos en comparación con el escenario sin obstáculos. En particular, el porcentaje de uso de CPU muestra un aumento notable en el algoritmo DFS, llegando hasta un 16.2973% con un 50% de obstáculos, mientras que, en los demás algoritmos, como BFS y Dijkstra, también se registra un incremento, aunque menos pronunciado. Por ejemplo, el algoritmo A* presenta un uso de CPU del 15.4065% con un 50% de obstáculos, indicando un mayor esfuerzo computacional en estos escenarios más complejos.

El uso de memoria, aunque similar entre los distintos algoritmos y escenarios, refleja pequeñas variaciones a medida que aumenta el porcentaje de obstáculos. Por ejemplo, en DFS, el uso de memoria disminuye ligeramente a medida que aumentan los obstáculos, pasando de 0.0093 MB con un 10% de obstáculos a 0.0049 MB con un 50% de obstáculos. Un patrón similar se observa en otros algoritmos como BFS y Dijkstra, aunque estos cambios son menos pronunciados.

La cantidad de bloques computados se reduce notablemente con la presencia de obstáculos, ya que estos limitan el espacio de búsqueda disponible. Sin embargo, en el caso de DFS, se observa un aumento en los bloques computados a medida que se incrementa el porcentaje de obstáculos, lo que se debe a la naturaleza del algoritmo que explora en profundidad antes de retroceder. Con un 50% de obstáculos, DFS computa 28 bloques, al igual que A* computa 28 bloques, pero con una notable diferencia en tiempo de ejecución (0.0141 segundos para A* frente a 0.0144 segundos para DFS).

En cuanto a la eficiencia en tiempo de ejecución, el algoritmo A* sigue siendo el más rápido, manteniendo su ventaja incluso en escenarios con hasta un 50% de obstáculos. Su tiempo de ejecución, que varía desde 0.0121 segundos con un 10% de obstáculos hasta 0.0141 segundos con un 50% de obstáculos, demuestra su capacidad para manejar situaciones más complejas con un tiempo mínimo.

En resumen, la introducción de obstáculos en el escenario de 10x10 aumenta los requerimientos computacionales, especialmente en términos de uso de CPU. A pesar de esto, A* continúa destacándose como el algoritmo más eficiente, tanto en términos de bloques computados como en tiempo de ejecución, reafirmando su eficacia en escenarios desafiantes.

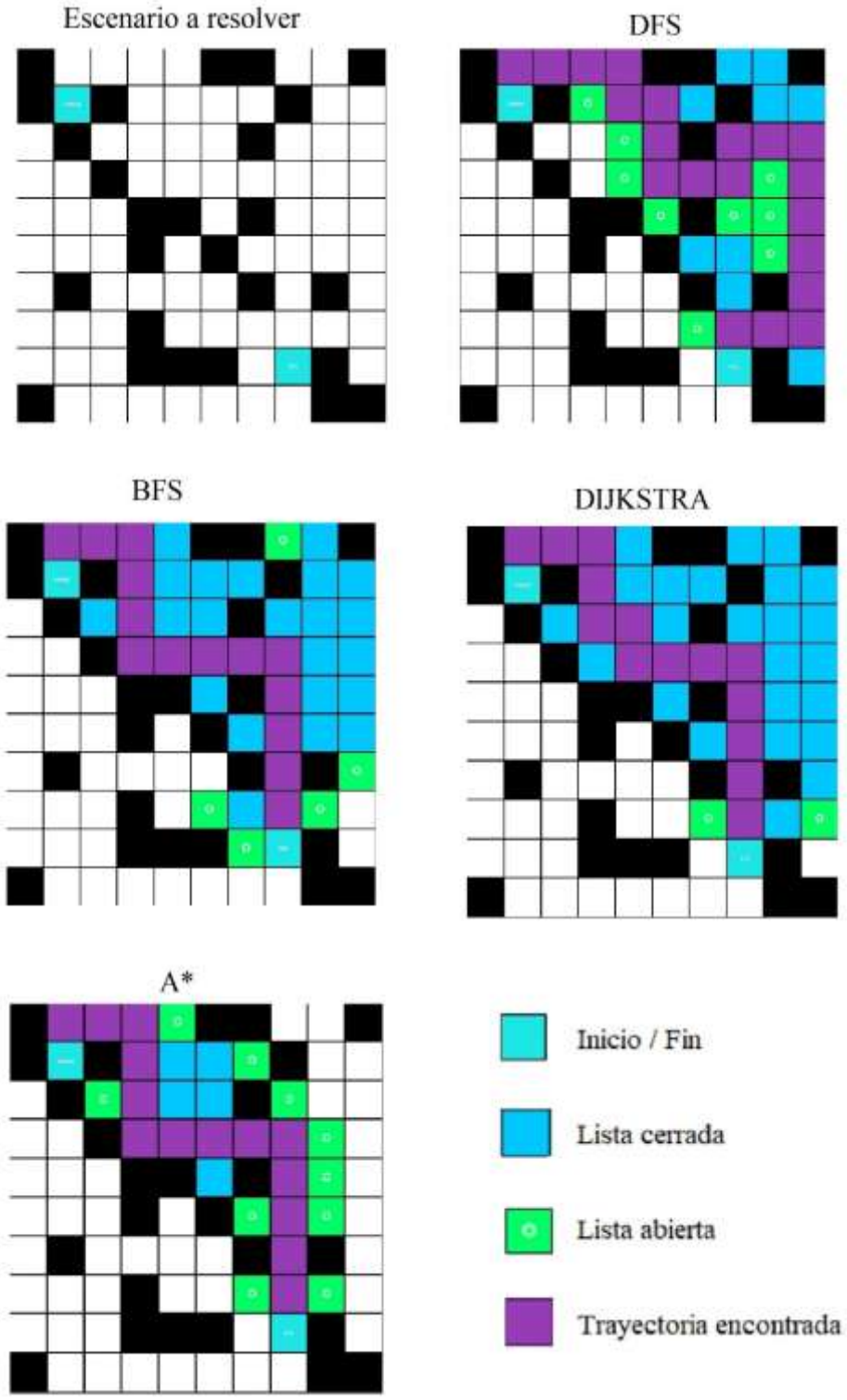

Figura 6.5 Ejemplo de caminos encontrados para un escenario de 10x10 bloques con obstáculos.

6.2.2 Escenario 20x20

En el segundo escenario de esta etapa, se utiliza una malla de 20x20 bloques donde los obstáculos se colocan al azar. Los resultados del segundo escenario se muestran en la siguiente tabla.

Tabla 6.6 Comparación del consumo de recursos computacionales de los algoritmos DFS, BFS, Dijkstra y A en un escenario de 20x20 bloques con obstáculos.*

Algoritmo DFS					
Porcentaje de Obstáculos	Bloques Computados	Pasos realizados	Tiempo de ejecución (Segundos)	Uso de memoria (Megabits)	Porcentaje de CPU
10 %	400	141	0.0419	0.0249	1.5774 %
20 %	364	104	0.0394	0.0178	1.5035 %
30 %	327	57	0.0371	0.0173	1.8269 %
40 %	180	16	0.0226	0.0121	4.3480 %

Algoritmo BFS					
Porcentaje de Obstáculos	Bloques Computados	Pasos realizados	Tiempo de ejecución (Segundos)	Uso de memoria (Megabits)	Porcentaje de CPU
10 %	486	21	0.0900	0.0350	1.7897 %
20 %	436	21	0.0724	0.0232	1.7756 %
30 %	337	19	0.0508	0.0171	3.6461 %
40 %	191	11	0.0289	0.0129	6.6589 %

Algoritmo Dijkstra					
Porcentaje de Obstáculos	Bloques Computados	Pasos realizados	Tiempo de ejecución (Segundos)	Uso de memoria (Megabits)	Porcentaje de CPU
10 %	483	21	0.0556	0.0383	1.8169 %
20 %	441	22	0.0514	0.0264	1.9019 %
30 %	343	21	0.0397	0.0201	1.9944 %
40 %	196	7	0.0244	0.0149	6.2659 %

Algoritmo A*					
Porcentaje de Obstáculos	Bloques Computados	Pasos realizados	Tiempo de ejecución (Segundos)	Uso de memoria (Megabits)	Porcentaje de CPU
10 %	127	21	0.0186	0.0115	4.5454 %
20 %	123	21	0.0179	0.0123	4.2379 %
30 %	170	20	0.0222	0.0131	4.1401 %
40 %	143	9	0.0203	0.0126	3.6283 %

En el escenario de 20x20 con obstáculos, se observa una tendencia similar a la del escenario de 10x10 con obstáculos. Los bloques computados presentan una reducción significativa, mientras que el número de pasos realizados aumenta, lo que refleja la complejidad añadida por los obstáculos que impiden un camino directo al punto de interés. En términos de uso de CPU, se experimenta un aumento en comparación con escenarios sin obstáculos, mientras que el uso de memoria muestra una disminución.

El tiempo de ejecución varía según el algoritmo: tanto DFS como A* experimentan un ligero incremento en el tiempo de ejecución, mientras que los algoritmos de Dijkstra y BFS muestran una disminución. Es destacable que, aunque el algoritmo DFS computa menos bloques en comparación con otros escenarios, su tiempo de ejecución sigue siendo mayor que el de los demás algoritmos. Por otro lado, el algoritmo A* continúa siendo eficiente en términos de bloques computados y tiempo de ejecución, aunque muestra un mayor uso de CPU en comparación con otros algoritmos.

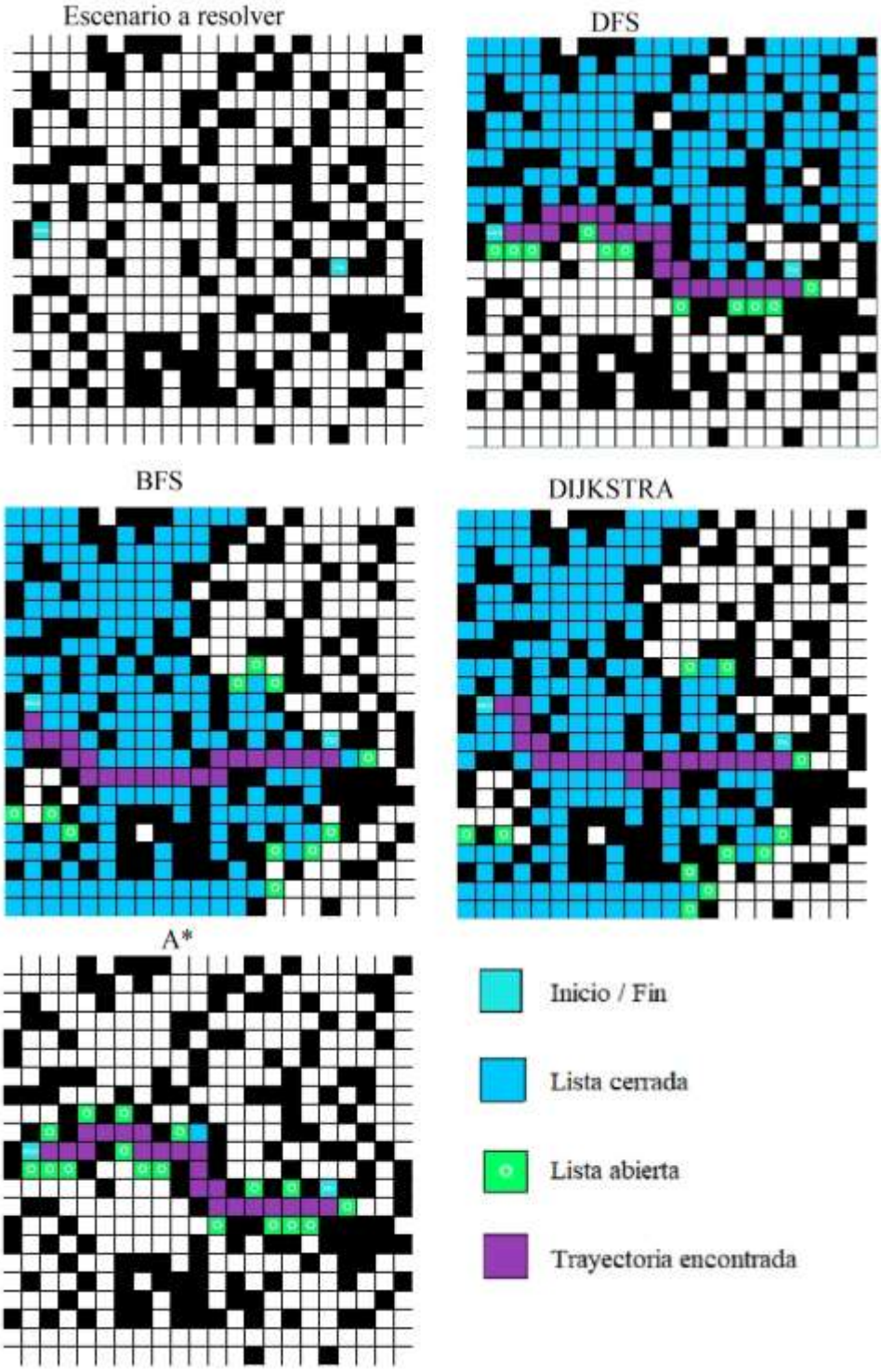

Figura 6.6 Bloques calculados y trayectorias encontradas para un escenario de 20x20 bloques con obstáculos.

6.2.3 Escenario 30x30

Continuando con el aumento del tamaño del escenario, así como los obstáculos presentes permite seguir con la evaluación del comportamiento y desempeño de los algoritmos. Al realizar la implementación y evaluación del escenario con obstáculos en un escenario de 30x30 bloques se obtienen los resultados que se presentan en la Tabla 12.

Tabla 6.7 Comparación del consumo de recursos computacionales de los algoritmos DFS, BFS, Dijkstra y A en un escenario 30x30 bloques con obstáculos.*

Algoritmo DFS					
Porcentaje de Obstáculos	Bloques Computados	Pasos realizados	Tiempo de ejecución (Segundos)	Uso de memoria (Megabits)	Porcentaje de CPU
10 %	909	219	0.1103	0.0542	1.8648 %
20 %	723	167	0.0831	0.0434	1.6873 %
30 %	648	96	0.0742	0.0362	2.9357 %
40 %	356	26	0.0409	0.0223	3.5902%

Algoritmo BFS					
Porcentaje de Obstáculos	Bloques Computados	Pasos realizados	Tiempo de ejecución (Segundos)	Uso de memoria (Megabits)	Porcentaje de CPU
10 %	934	23	0.1769	0.0547	1.9748 %
20 %	859	25	0.0147	0.0508	2.1554 %
30 %	675	25	0.1039	0.0404	3.1051 %
40 %	356	12	0.0516	0.0239	3.2412 %

Algoritmo Dijkstra					
Porcentaje de Obstáculos	Bloques Computados	Pasos realizados	Tiempo de ejecución (Segundos)	Uso de memoria (Megabits)	Porcentaje de CPU
10 %	916	23	0.1069	0.0605	2.2699 %
20 %	829	25	0.0964	0.0582	2.3117 %
30 %	693	26	0.0813	0.04739	2.7299 %
40 %	373	14	0.04465	0.0293	4.5504 %

Algoritmo A*					
Porcentaje de Obstáculos	Bloques Computados	Pasos realizados	Tiempo de ejecución (Segundos)	Uso de memoria (Megabits)	Porcentaje de CPU
10 %	188	23	0.0251	0.0159	4.0586 %
20 %	187	24	0.0249	0.0163	3.9000 %
30 %	279	26	0.0342	0.0200	4.0262 %
40 %	308	16	0.0380	0.0250	4.9018 %

En este tercer escenario, utilizando una malla de 20x20 con diferentes porcentajes de obstáculos, los algoritmos BFS, Dijkstra y A* vuelven a encontrar caminos similares hacia el punto final, aunque con variaciones en los tiempos de ejecución. El algoritmo A* continúa destacándose por ser el más eficiente en cuanto al tiempo necesario para encontrar la ruta óptima, seguido por el algoritmo DFS, luego Dijkstra y finalmente BFS. En términos de uso de recursos computacionales, se observa un incremento en el porcentaje de uso de CPU y memoria en comparación con los escenarios anteriores. Esto refleja cómo el aumento del tamaño de la cuadrícula y la introducción de obstáculos aleatorios incrementan la complejidad del problema, resultando en una mayor demanda de recursos computacionales por parte de los algoritmos.

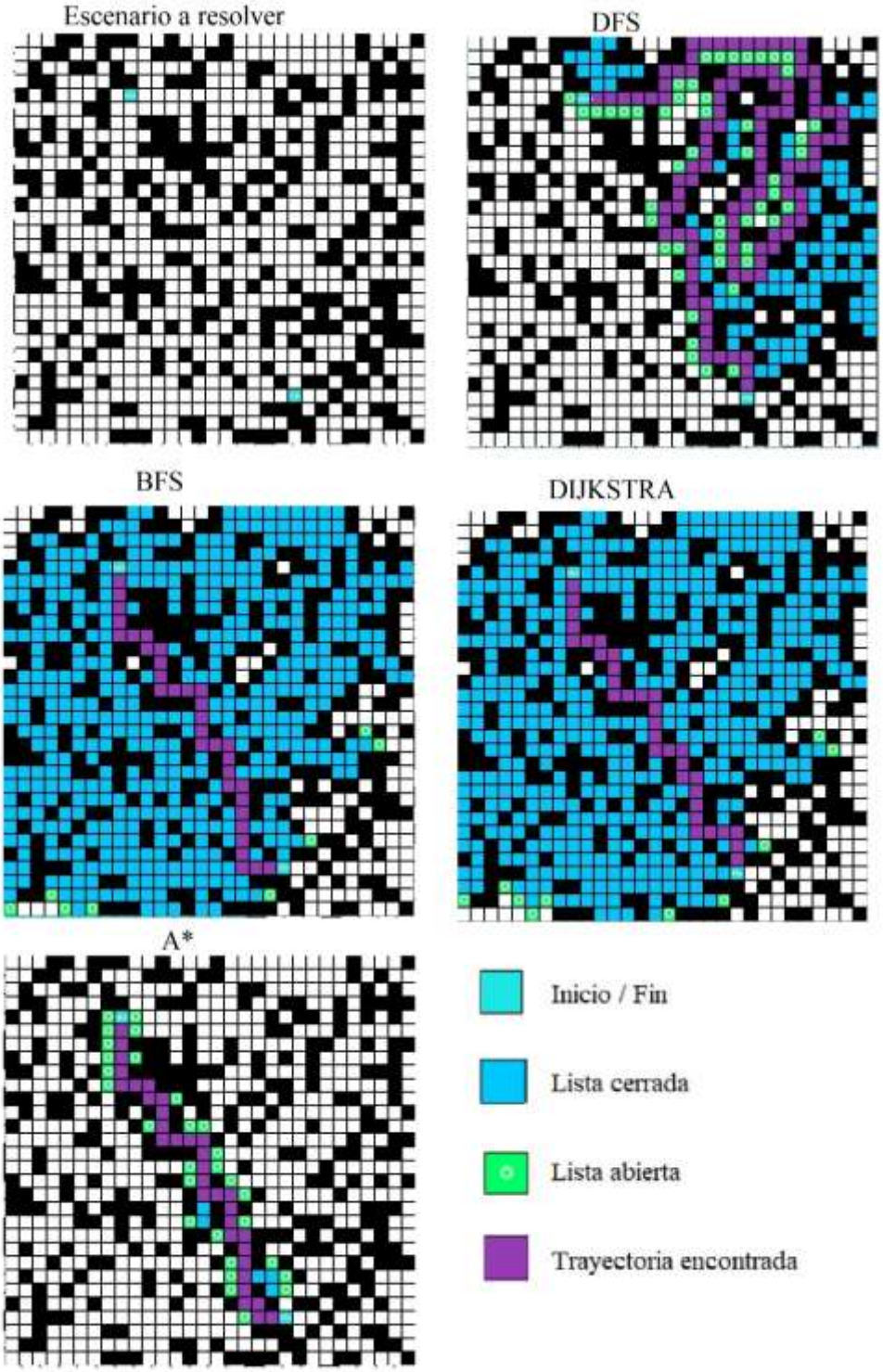

Figura 6.7 Bloques calculados y trayectorias encontradas para un escenario de 30x30 bloques con obstáculos.

En la figura 6.7, se muestra otro ejemplo de los diferentes caminos que siguen los algoritmos para llegar al punto de interés. En esta ocasión el algoritmo BFS y A* encontraron el mismo camino, el algoritmo DFS muestra que primero se realiza la búsqueda hacia la parte superior derecha y va realizando la búsqueda por los bloques que se encuentran libres para su paso. El algoritmo A* al tener una lista cerrada menor y encontrar el camino en un menor tiempo de ejecución se perfila para ser el algoritmo para implementar.

6.2.4 Escenario 50x50

En el escenario de 50x50, se realiza un análisis exhaustivo del rendimiento de los algoritmos de búsqueda de trayectorias en una malla significativamente más grande y compleja. Este escenario permite evaluar cómo los algoritmos manejan un entorno más extenso y detallado, donde el número de bloques computados y la longitud de los caminos pueden aumentar considerablemente.

Tabla 6.8 Comparación del consumo de recursos computacionales de los algoritmos DFS, BFS, Dijkstra y A en un escenario de 50x50 bloques con obstáculos.*

Algoritmo DFS					
Porcentaje de Obstáculos	Bloques Computados	Pasos realizados	Tiempo de ejecución (Segundos)	Uso de memoria (Megabits)	Porcentaje de CPU
10 %	1550	441	0.1802	0.1033	1.5950 %
20 %	1631	378	0.1926	0.1032	1.7394 %
30 %	1454	251	0.1653	0.0882	1.8438 %
40 %	715	34	0.0823	0.0350	7.1072 %

Algoritmo BFS					
Porcentaje de Obstáculos	Bloques Computados	Pasos realizados	Tiempo de ejecución (Segundos)	Uso de memoria (Megabits)	Porcentaje de CPU
10 %	2003	29	0.3884	0.1313	1.8273 %
20 %	1749	30	0.3039	0.1123	1.8479 %
30 %	1493	31	0.2332	0.0798	4.6284 %
40 %	704	13	0.1013	0.0347	8.3442 %

Algoritmo Dijkstra					
Porcentaje de Obstáculos	Bloques Computados	Pasos realizados	Tiempo de ejecución (Segundos)	Uso de memoria (Megabits)	Porcentaje de CPU
10 %	1995	29	0.2321	0.1477	1.9284 %
20 %	1744	30	0.2019	0.1282	1.9270 %
30 %	1486	32	0.1731	0.0943	3.1929 %
40 %	683	16	0.0794	0.04160	8.4298 %

Algoritmo A*					
Porcentaje de Obstáculos	Bloques Computados	Pasos realizados	Tiempo de ejecución (Segundos)	Uso de memoria (Megabits)	Porcentaje de CPU
10 %	423	29	0.0523	0.0311	3.2974 %
20 %	403	30	0.0492	0.0292	3.3913 %
30 %	342	31	0.0799	0.0252	4.7869 %
40 %	666	17	0.0102	0.0392	8.3711 %

En este escenario de 50x50, los algoritmos BFS y Dijkstra continúan mostrando un comportamiento similar en términos de bloques computados, reflejando su enfoque exhaustivo en la exploración del espacio de búsqueda. Sin embargo, el algoritmo A* sigue destacando por su eficiencia, utilizando significativamente menos tiempo de ejecución y ofreciendo un equilibrio óptimo entre todos los criterios evaluados. Por otro lado, el algoritmo BFS presentó el peor desempeño en cuanto a tiempo de ejecución.

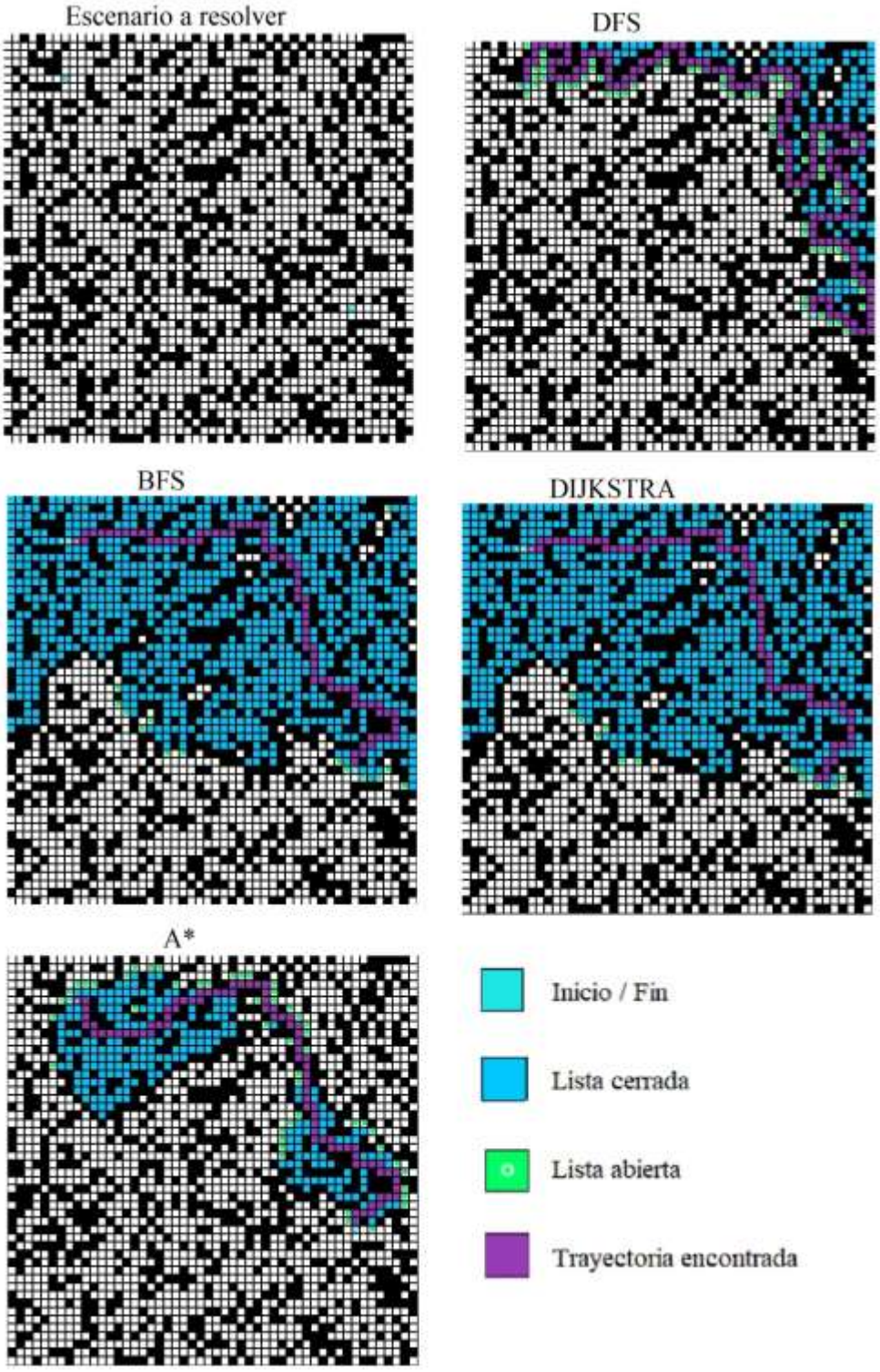

Figura 6.8 Bloques calculados y trayectorias encontradas para un escenario de 50x50 bloques con obstáculos.

6.3. Escenarios mayores a 50x50

A continuación, se presenta una tabla que resume los resultados obtenidos al ejecutar los algoritmos en mallas de mayor tamaño. Dado que en este punto resulta impráctico representar los resultados en una imagen, se optó por condensar la información en una única tabla por algoritmo.

Para evaluar el rendimiento y la escalabilidad de los algoritmos en escenarios más complejos, se llevaron a cabo pruebas en mallas de dimensiones 75x75, 100x100, 200x200, 300x300, 400x400 y 500x500. Estas pruebas permiten observar cómo el aumento en el tamaño de la malla afecta tanto el tiempo de ejecución como el uso de recursos computacionales, incluyendo la memoria y la CPU. El análisis de estos resultados es crucial para entender cómo cada algoritmo maneja la complejidad añadida por el incremento en el tamaño de la malla y la presencia de obstáculos, destacando las diferencias en eficiencia y comportamiento a medida que las mallas se vuelven más grandes y los escenarios más complejos.

Tabla 6.9 Tabla de resultados en escenarios mayores a 50x50 algoritmo DFS

Algoritmo DFS						
Tamaño de Escenario	Porcentaje de Obstáculos	Bloques Computados	Pasos realizados	Tiempo de ejecución (Segundos)	Uso de memoria (Megabits)	Porcentaje de CPU
75x75	0 %	4876	1562	0.6046	0.3767	1.5271 %
	10 %	4815	1471	0.5179	0.3647	1.4490 %
	20 %	4252	1100	0.4651	0.3026	1.5426 %
	30 %	3564	626	0.3933	0.2371	2.3189 %
	40 %	1764	84	0.1932	0.1046	5.4867 %
100x100	0 %	11035	3967	1.1391	1.0476	1.0121 %
	10 %	8047	2555	0.6908	0.7298	0.9937 %
	20 %	7624	1907	0.5904	0.6531	0.8608 %
	30 %	6235	1186	0.4187	0.4972	1.1684 %
	40 %	2887	90	0.2255	0.1744	5.0923 %
200x200	0 %	55426	16744	7.6652	5.4288	1.6334 %
	10 %	35897	10482	4.4220	3.6481	1.6543 %
	20 %	33352	7713	4.1375	3.2656	2.5125 %
	30 %	26109	4493	3.1496	2.3029	2.5670 %
	40 %	14471	380	1.7652	1.0178	5.3014 %
300x300	0 %	106096	32164	15.5476	9.8063	1.6997 %
	10 %	83516	27963	10.3545	8.1181	1.7263 %
	20 %	77566	20003	9.6896	7.0913	1.6833 %
	30 %	69839	11157	8.8824	6.2120	2.5770 %
	40 %	28312	644	4.1303	2.4057	5.4574 %
400x400	0 %	139816	43449	23.7512	14.1654	1.8454 %
	10 %	134541	38490	20.3220	13.6541	1.8855 %
	20 %	133443	29481	21.7371	13.1679	1.9608 %
	30 %	105808	15335	16.5105	10.0546	2.4574 %
	40 %	58609	1229	8.5105	5.0289	7.0186 %
500x500	0 %	257345	84042	32.5218	28.8230	1.4922 %
	10 %	196737	52928	24.1910	21.4757	1.3915 %
	20 %	179274	38818	19.6548	19.3781	1.1897 %
	30 %	164933	20968	16.2051	16.0592	2.0878 %
	40 %	65634	1367	6.8314	5.9952	6.1594 %

Tabla 6.10 Tabla de resultados en escenarios mayores a 50x50 algoritmo BFS

Algoritmo BFS						
Tamaño de Escenario	Porcentaje de Obstáculos	Bloques Computados	Pasos realizados	Tiempo de ejecución (Segundos)	Uso de memoria (Megabits)	Porcentaje de CPU
75x75	0 %	6233	54	1.3244	0.4766	1.6942 %
	10 %	5599	55	1.0148	0.3073	1.6407 %
	20 %	4928	56	0.8313	0.2680	1.7395 %
	30 %	4127	60	0.6257	0.2232	3.0895 %
	40 %	1835	30	0.2537	0.1076	6.7544 %
100x100	0 %	11527	74	1.9397	0.8502	1.1737 %
	10 %	10317	74	1.5518	0.7972	1.1358 %
	20 %	8737	73	0.9946	0.6859	1.2952 %
	30 %	7360	78	0.6557	0.5576	1.6986 %
	40 %	3208	42	0.2947	0.1723	5.0132 %
200x200	0 %	50224	150	11.8265	4.0801	1.8575 %
	10 %	45253	150	9.6465	3.8447	1.8771 %
	20 %	39931	152	7.5965	3.6207	2.7448 %
	30 %	34138	158	5.7718	3.1331	2.8210 %
	40 %	15234	92	2.3022	0.9661	5.1078 %
300x300	0 %	99361	228	24.3517	7.9134	1.9171 %
	10 %	89708	229	19.7656	6.6771	1.8976 %
	20 %	79235	231	15.4313	5.6785	2.7960 %
	30 %	70400	243	12.2501	5.3870	1.9049 %
	40 %	25995	138	4.8353	2.0657	5.0678 %
400x400	0 %	131476	217	36.2832	10.8891	2.0578 %
	10 %	118049	218	29.2841	10.0651	2.0897 %
	20 %	105025	219	24.6333	8.8990	2.1866 %
	30 %	91078	236	18.5620	7.5757	2.1533 %
	40 %	51565	232	8.9291	4.1269	7.0545 %
500x500	0 %	169583	248	39.5233	14.6963	1.6224 %
	10 %	152487	250	29.5035	12.5656	1.5353 %
	20 %	135857	256	19.4110	11.5399	2.2501 %
	30 %	117911	267	13.7025	9.9078	1.8220 %
	40 %	82677	248	9.7665	7.1475	4.5977 %

Tabla 6.11 Tabla de resultados en escenarios mayores a 50x50 algoritmo de Dijkstra

Algoritmo Dijkstra						
Tamaño de Escenario	Porcentaje de Obstáculos	Bloques Computados	Pasos realizados	Tiempo de ejecución (Segundos)	Uso de memoria (Megabits)	Porcentaje de CPU
75x75	0 %	6227	54	0.7079	0.5072	1.7615 %
	10 %	5573	54	0.5987	0.3384	1.6526 %
	20 %	4996	56	0.5572	0.3076	1.7832 %
	30 %	3871	53	0.4308	0.2480	4.9791 %
	40 %	1822	30	0.2041	0.1140	5.5127 %
100x100	0 %	11523	74	1.0277	0.9120	1.1967 %
	10 %	10316	74	0.9141	0.8759	1.1735 %
	20 %	9112	74	0.6827	0.7865	1.0111 %
	30 %	7247	78	0.4750	0.5983	1.1728 %
	40 %	3038	42	0.2560	0.1957	5.1529 %
200x200	0 %	50297	150	6.3133	4.3585	1.9396 %
	10 %	45303	150	5.6734	4.1930	1.9442 %
	20 %	40131	153	5.0093	4.0482	1.9434 %
	30 %	33717	156	4.1843	3.4464	3.7174 %
	40 %	13063	109	1.6023	1.0027	4.7276 %
300x300	0 %	99328	228	13.0275	8.6652	2.0113 %
	10 %	89640	229	11.6649	7.3374	1.9884 %
	20 %	80107	230	10.3233	6.4260	1.9656 %
	30 %	68503	245	8.8075	5.9653	2.8603 %
	40 %	28599	159	4.1287	2.6741	6.6661 %
400x400	0 %	131467	217	19.5452	11.7623	2.1971 %
	10 %	119626	215	17.4603	11.1366	2.1990 %
	20 %	103231	220	15.9545	9.7033	3.1987 %
	30 %	90136	234	13.6776	8.4324	3.1316 %
	40 %	52528	175	7.3861	5.1129	4.5041 %
500x500	0 %	169829	248	21.4348	16.1626	1.7265 %
	10 %	152726	250	17.6309	14.0032	1.6179 %
	20 %	135783	256	12.9450	12.9530	1.3918 %
	30 %	127767	274	11.4973	12.2138	1.3754 %
	40 %	75610	220	7.2115	7.5432	2.5836 %

*Tabla 6.12 Tabla de resultados en escenarios mayores a 50x50 algoritmo A**

Algoritmo A*						
Tamaño de Escenario	Porcentaje de Obstáculos	Bloques Computados	Pasos realizados	Tiempo de ejecución (Segundos)	Uso de memoria (Megabits)	Porcentaje de CPU
75x75	0 %	954	54	0.1014	0.0816	2.3828 %
	10 %	828	55	0.0822	0.0555	2.1527 %
	20 %	675	56	0.0790	0.0531	2.4411 %
	30 %	1027	58	0.1222	0.0660	5.2978 %
	40 %	1354	37	0.1524	0.0885	5.6233 %
100x100	0 %	1527	74	0.1242	0.1157	1.8636 %
	10 %	1307	74	0.1232	0.0976	1.8349 %
	20 %	1311	74	0.1035	0.0916	1.6894 %
	30 %	1284	80	0.0849	0.0859	1.3290 %
	40 %	2703	44	0.2200	0.1768	5.1036 %
200x200	0 %	4936	150	0.6278	0.4846	2.2666 %
	10 %	4202	150	0.5379	0.3260	2.3127 %
	20 %	3767	150	0.4854	0.3206	3.0804 %
	30 %	5399	160	0.7065	0.4057	3.0947 %
	40 %	10166	117	1.2909	0.7659	8.5516 %
300x300	0 %	13944	228	1.8664	1.3080	2.4014 %
	10 %	11600	229	1.5640	1.0782	2.1610 %
	20 %	9575	234	1.3175	0.8740	2.0585 %
	30 %	10327	243	1.4215	0.8818	2.1375 %
	40 %	27178	163	3.9902	2.5531	6.0160 %
400x400	0 %	7570	217	1.0621	0.7769	2.5567 %
	10 %	6598	219	1.0606	0.7978	2.4164 %
	20 %	7188	223	1.1808	0.7110	2.4043 %
	30 %	13313	224	2.1956	1.2736	4.0425 %
	40 %	39687	227	5.8843	3.8037	3.9046 %
500x500	0 %	16459	248	2.1596	1.6906	2.0291 %
	10 %	13487	250	1.5187	1.5802	1.8244 %
	20 %	12899	256	1.2860	1.2617	1.5366 %
	30 %	21663	280	2.1812	2.2048	1.4391 %
	40 %	65566	237	6.5788	6.8291	2.2115 %

6.4 Comparación de resultados

En esta sección, se presentan una serie de imágenes, que ilustran los resultados obtenidos al aplicar los algoritmos de búsqueda en mallas de diferentes tamaños y con diversos porcentajes de obstáculos. Estas imágenes permiten visualizar de manera clara y comparativa el rendimiento de cada algoritmo en términos de bloques computados, pasos realizados, tiempo de ejecución, uso de memoria y porcentaje de CPU.

6.4.1 Uso de CPU por algoritmo

En las siguientes figuras se muestra una comparación del consumo de porcentaje de uso de CPU de los algoritmos. Las figuras se muestran divididas por porcentaje de obstáculos.

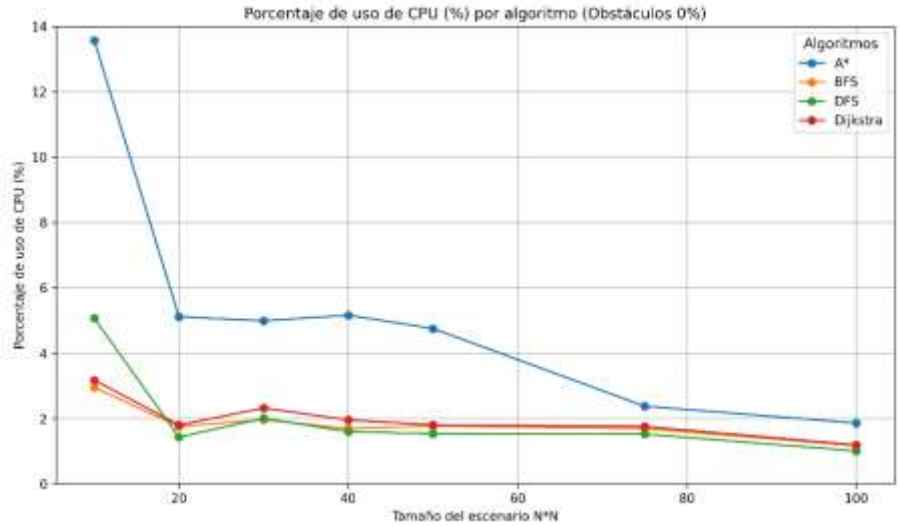

Figura 6.9 Uso de CPU por algoritmo - Obstáculos 0%

Al analizar las pruebas sin obstáculos, el algoritmo que más porcentaje de CPU utiliza es el algoritmo A*. En escenarios pequeños, la cantidad de nodos es reducida, pero el costo relativo de esas operaciones adicionales es mayor en comparación con BFS/DFS/Dijkstra.

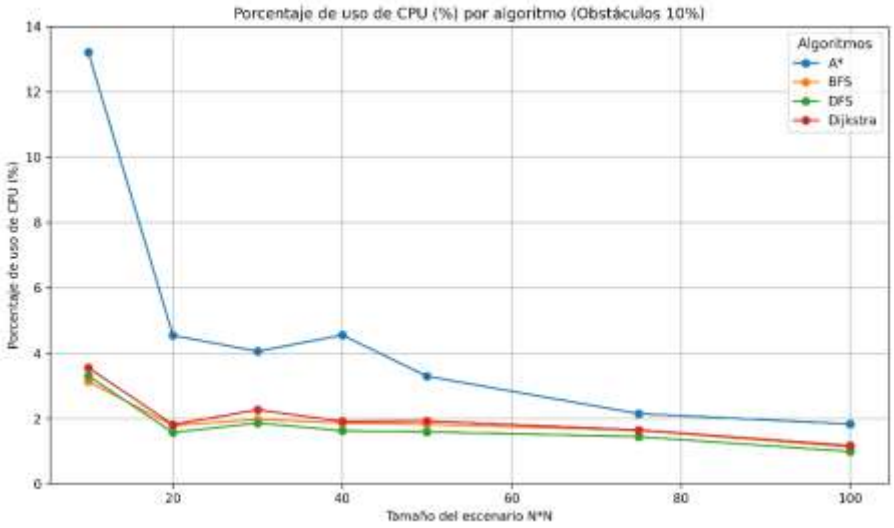

Figura 6.10 Uso de CPU por algoritmo - Obstáculos 10%

Lo que sucede en las figuras 6.10 y 6.11 se puede observar una reducción del uso del CPU, esto se debe que al haber menos nodos por visitar hace que el cálculo de la trayectoria requiera menos uso del CPU.

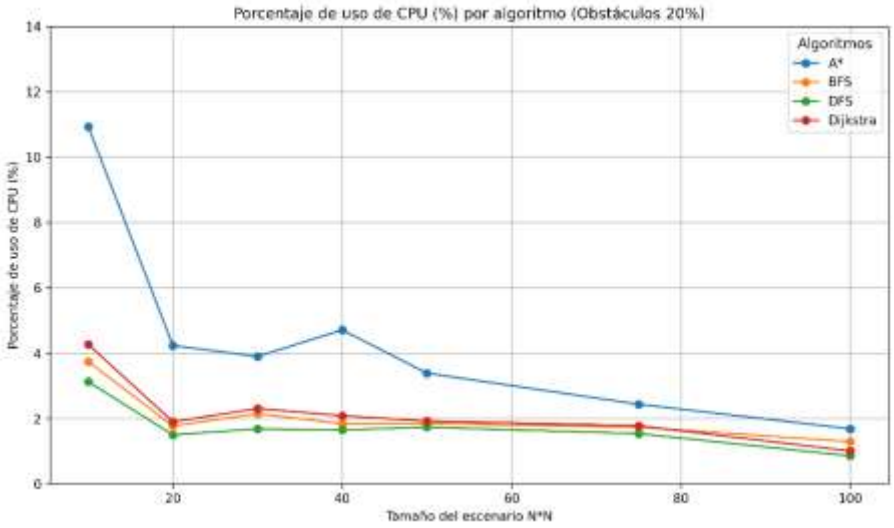

Figura 6.11 Uso de CPU por algoritmo - Obstáculos 20%

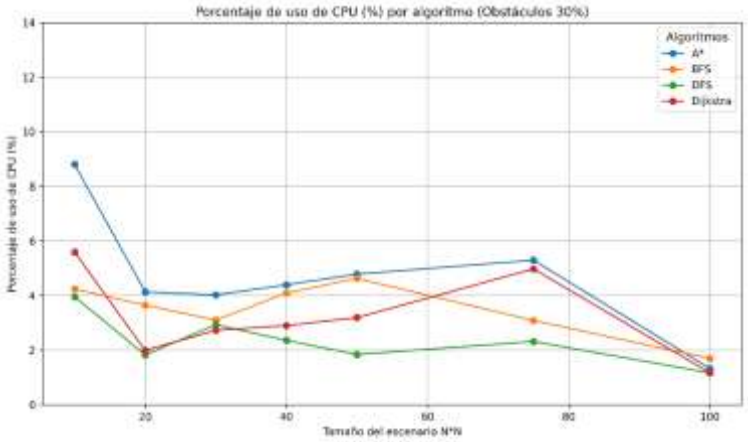

Figura 6.12 Uso de CPU por algoritmo - Obstáculos 30%

En las figuras 6.12 y 6.13, correspondientes a escenarios con un 30% y 40% de obstáculos respectivamente, se observa un comportamiento más irregular en el uso de CPU entre los diferentes algoritmos en comparación con el caso de menor densidad de obstáculos. A medida que aumenta la cantidad de obstáculos, el consumo de CPU tiende a variar debido a que los algoritmos deben realizar más evaluaciones de nodos y recalcular rutas alternativas, incrementando la carga computacional de manera no uniforme.

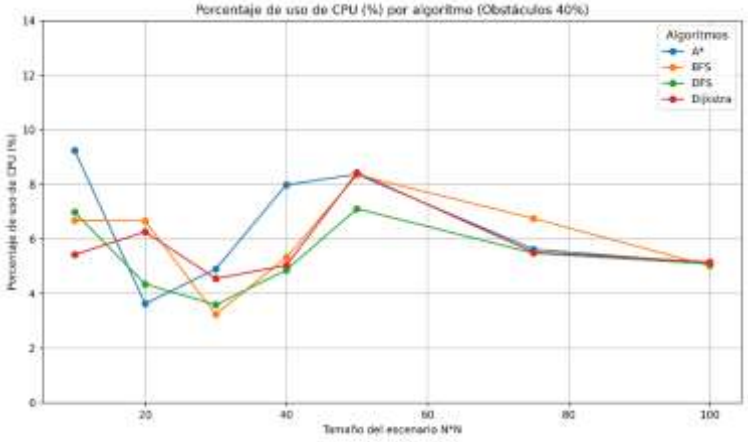

Figura 6.13 Uso de CPU por algoritmo - Obstáculos 40%

6.4.2 Tiempo por algoritmo

En las siguientes figuras se muestra una comparación de la duración del tiempo de ejecución de los algoritmos. Las figuras se muestran divididas por porcentaje de obstáculos.

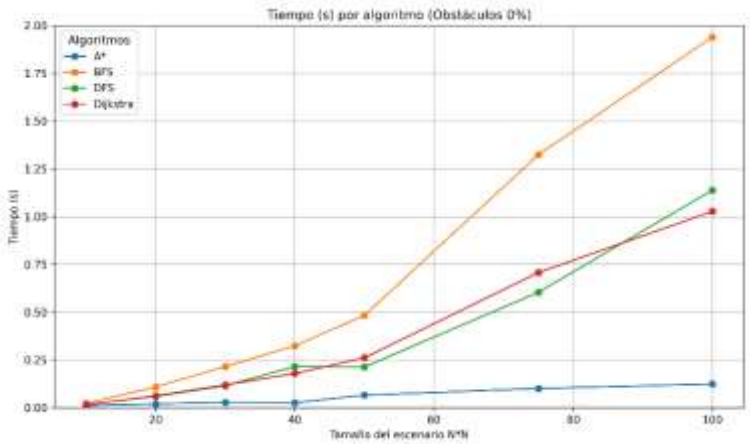

Figura 6.14 Tiempo por Algoritmo - Obstáculos 0%

En un entorno sin obstáculos o con 10% de obstáculos, se puede apreciar que el algoritmo A* es el que menos tiempo requiere para encontrar la solución, siendo el algoritmo BFS el que peor desempeño demuestra en las diferentes pruebas realizadas.

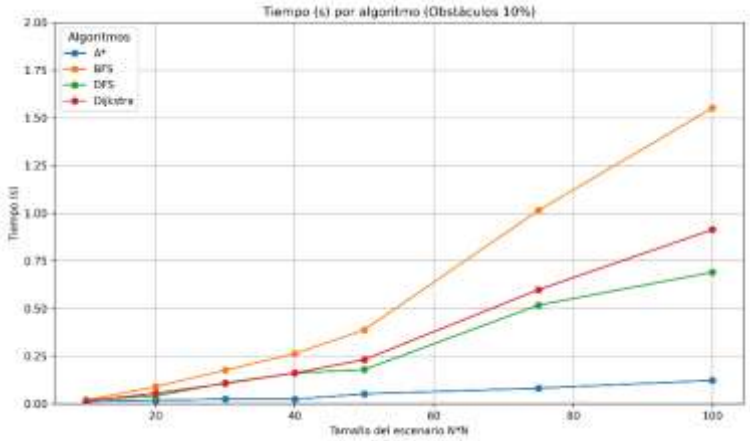

Figura 6.15 Tiempo por Algoritmo - Obstáculos 10%

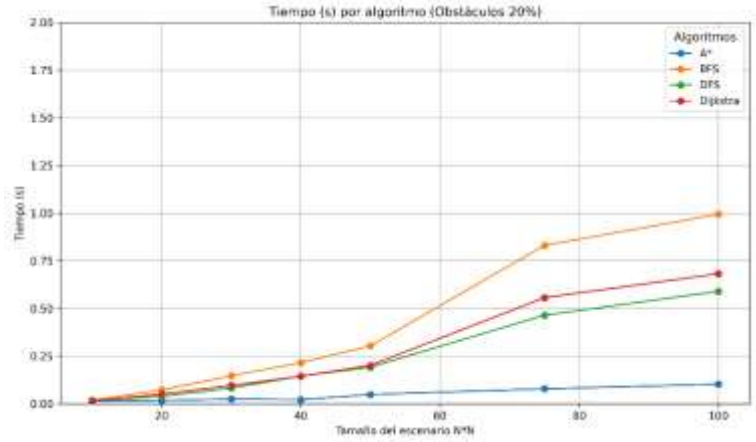

Figura 6.16 Tiempo por Algoritmo - Obstáculos 20%

Al disminuir el área de búsqueda se reduce el tiempo que requieren los algoritmos para realizar la búsqueda de la solución, esto se debe a que al existir menos nodos libres para visitar se realizan menos iteraciones, lo cual conlleva a una reducción de tiempo como se muestra en las figuras.

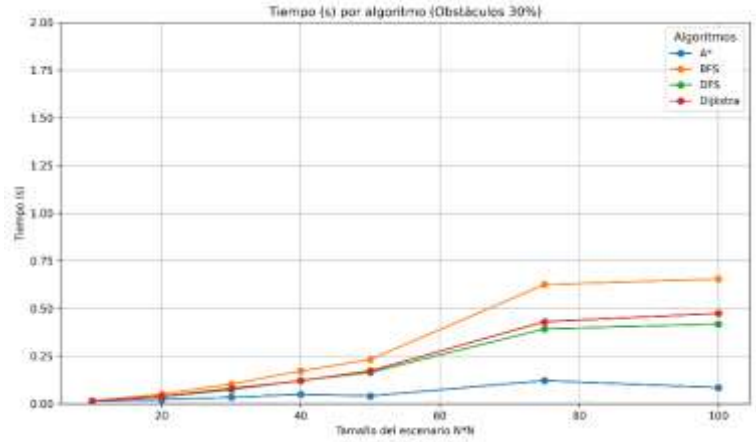

Figura 6.17 Tiempo por Algoritmo - Obstáculos 30%

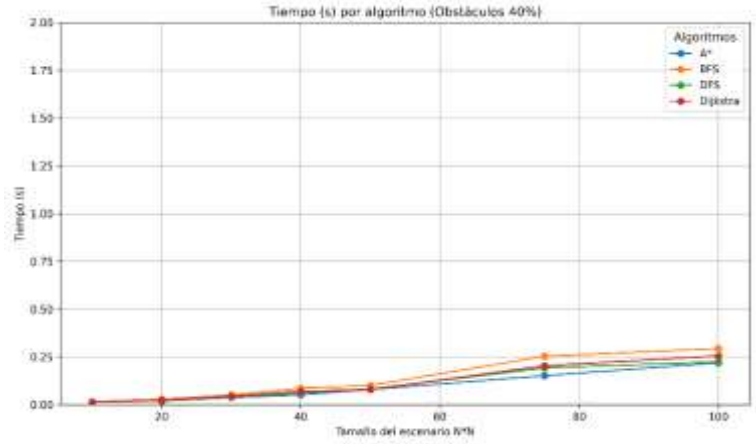

Figura 6.18 Tiempo por Algoritmo - Obstáculos 40%

En la siguiente figura se muestra la distribución del tiempo de ejecución de los algoritmos, sin importar su porcentaje de obstáculos. El algoritmo A* muestra el mejor desempeño y el algoritmo BFS muestra el peor desempeño.

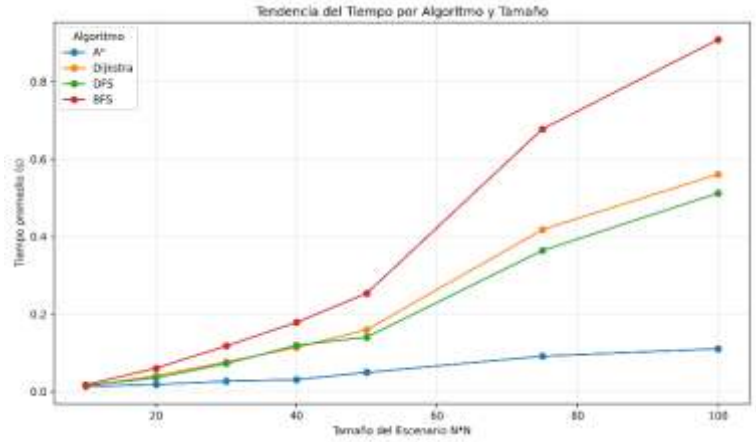

Figura 6.19 Tendencia del tiempo por algoritmo

6.4.3 Uso de memoria por algoritmo

En las siguientes figuras se muestra una comparación del consumo de los algoritmos. Las figuras se muestran divididas por porcentaje de obstáculos.

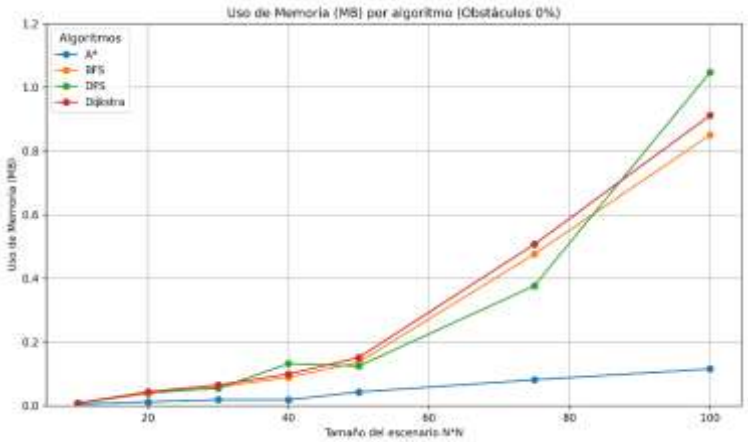

Figura 6.20 Uso de memoria por algoritmo - Obstáculos 0%

En un entorno sin obstáculos, los algoritmos al inicio de las pruebas presentan un comportamiento similar, siendo hasta las pruebas de 50*50 donde se comienza a visualizar un mayor uso de memoria en los diferentes algoritmos, siendo el algoritmo A* el que menos varianza tiene dentro de las pruebas, y siendo en este caso el algoritmo DFS el que demuestra el pero desempeño al final de las pruebas.

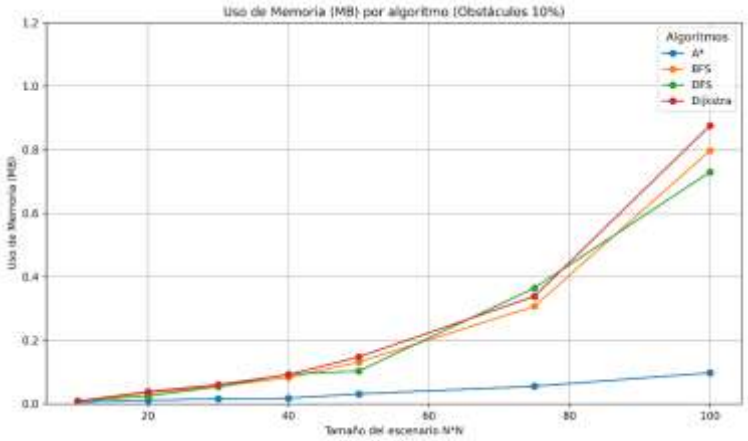

Figura 6.21 Uso de memoria por algoritmo - Obstáculos 10%

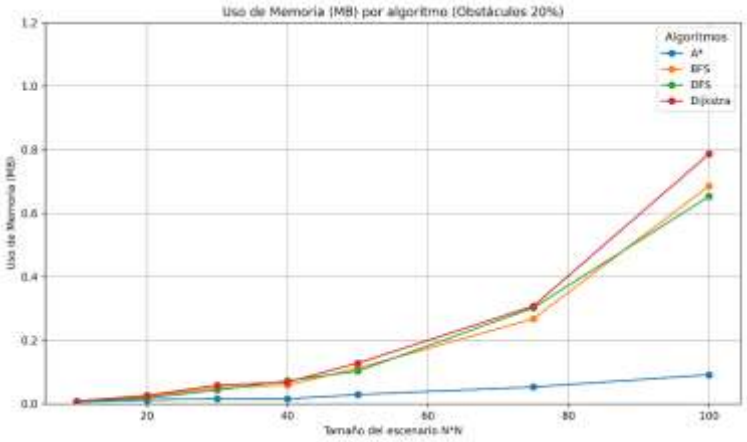

Figura 6.22 Uso de memoria por algoritmo - Obstáculos 20%

Tal como sucede en el tiempo que requiere cada algoritmo para encontrar la solución, en el uso de memoria ocurre algo similar, al existir menos nodos para realizar la búsqueda cuando se va aumentando el número de obstáculos disminuye también el uso de memoria en cada algoritmo.

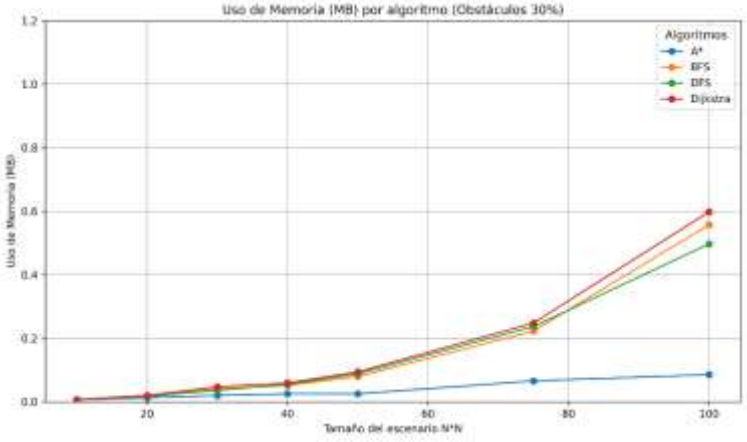

Figura 6.23 Uso de memoria por algoritmo - Obstáculos 30%

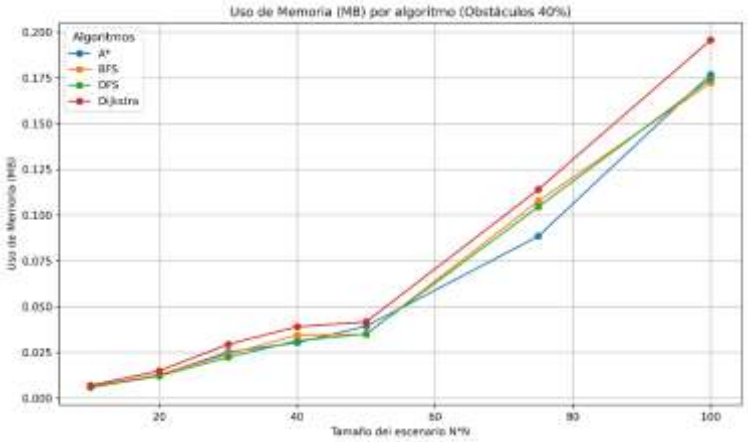

Figura 6.24 Uso de memoria por algoritmo - Obstáculos 40%

En las pruebas con 40% de obstáculos, se presenta muy poca diferencia en el uso de memoria entre los diferentes algoritmos evaluados, en esta ocasión siendo el algoritmo BFS el que muestra el mejor desempeño y el algoritmo Dijkstra el que peor desempeño demuestra.

6.4.4 Pasos en ruta encontrada

En las siguientes figuras se muestra una comparación de los pasos requeridos durante la trayectoria.

Figura 6.25 Pasos en ruta encontrada por tamaño y algoritmo

Para una mejor visualización de los resultados de los algoritmos A*, BFS y Dijkstra se elimina de la gráfica el algoritmo DFS.

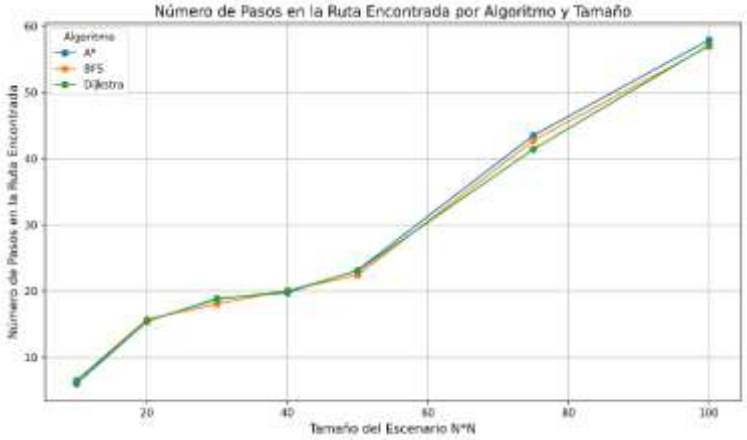

Figura 6.26 Pasos en ruta encontrada por tamaño y algoritmo sin DFS

6.4.5 Número de bloques computados por algoritmo y tamaño

En las siguientes figuras se muestra una comparación de los bloques de los algoritmos. La muestra de los resultados se divide en dos, en la primera figura se muestran los escenarios de 10x10 bloques hasta 100x100 bloques.

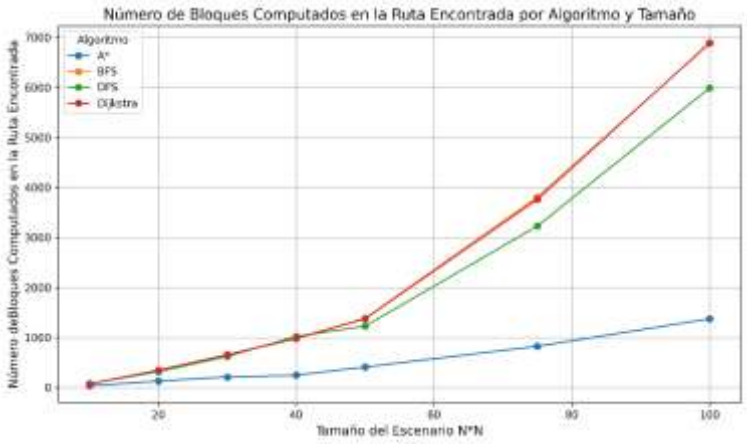

Figura 6.27 Número de bloques computados por algoritmo y tamaño de 10 a 100

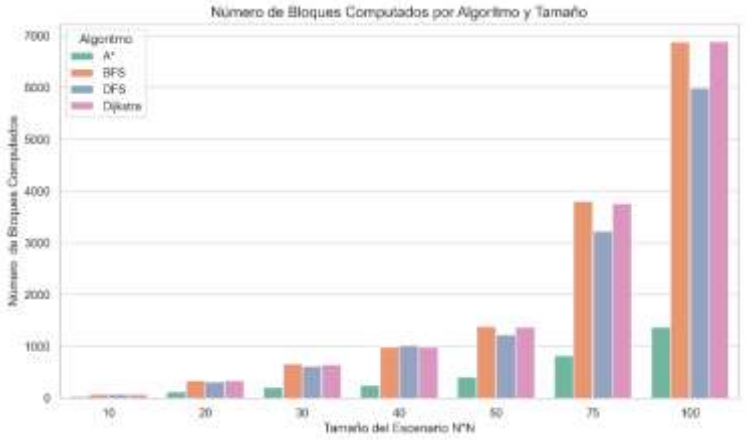

Figura 6.28 Número de bloques computados por algoritmo y tamaño de 10 a 100

6.4.6 Comparación de los algoritmos en memoria, tiempo y uso de CPU

La siguiente figura permite evaluar el equilibrio entre las diferentes métricas. Si todas las dimensiones de un algoritmo están cercanas al centro, podría indicar un rendimiento más equilibrado en términos de CPU, memoria y tiempo. Por otro lado, si un algoritmo se extiende en una dirección específica, muestra un punto débil en esa métrica.

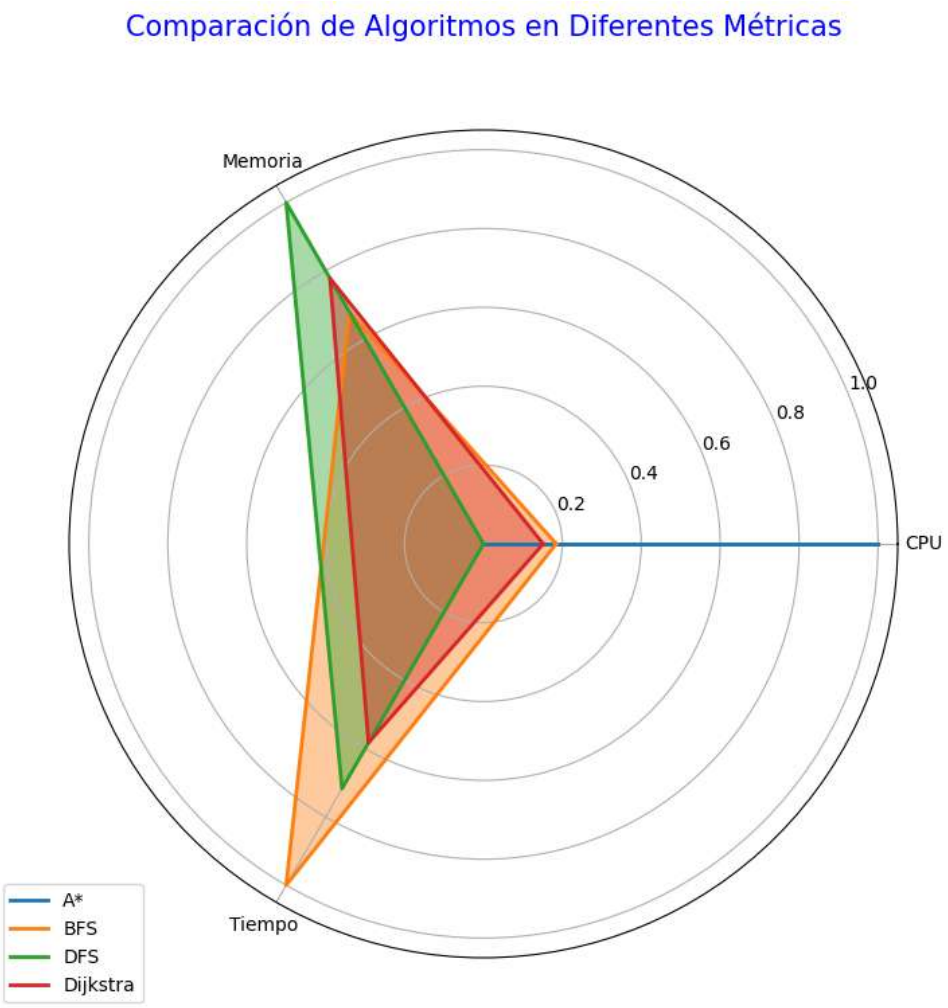

Figura 6.29 Comparación de los algoritmos en memoria, tiempo y uso de CPU

6.4.7 Puntuación ponderada

El análisis de la puntuación ponderada se realizó para proporcionar una evaluación más equilibrada y representativa de los algoritmos bajo estudio. Dado que los algoritmos pueden mostrar un rendimiento variable en diferentes métricas (como el uso de CPU, memoria, tiempo de ejecución, y número de pasos realizados), una evaluación basada en una sola métrica podría llevar a conclusiones sesgadas.

La puntuación ponderada permite integrar múltiples aspectos del rendimiento en una única métrica, asignando un peso a cada criterio según su importancia relativa en el contexto del problema. Esto facilita una comparación más justa y completa entre los algoritmos, destacando aquellos que logran un buen equilibrio entre eficiencia, uso de recursos, y capacidad de encontrar soluciones óptimas. De esta manera, el análisis ayuda a identificar el algoritmo que ofrece el mejor rendimiento general, considerando todas las dimensiones relevantes del problema.

Lectura y Preparación de Datos:

- Se lee un archivo CSV que contiene los resultados de las ejecuciones de los algoritmos. Posteriormente, se filtran ciertos datos basados en las columnas Porcentaje, y Tamaño para centrarse en los casos de interés.

Agrupación de Datos:

- Los datos se agrupan por los campos Algoritmo y Tamaño, calculando las medias de las métricas CPU y Memoria. Esto permite obtener una visión general del rendimiento promedio de cada algoritmo en relación con el tamaño de la entrada.

Análisis de la Métrica:

- Para cada algoritmo, se analiza cada métrica de rendimiento (CPU, Memoria) ajustando un modelo polinómico de grado 3 a los datos.
- El grado del polinomio más significativo se utiliza para inferir la complejidad computacional del algoritmo en relación con el tamaño de la entrada (O(1), O(n), O(n^2), O(n^3), etc.).
- Finalmente, se visualizan los resultados mediante gráficos que muestran tanto los datos reales como el ajuste del modelo polinómico, lo que permite observar visualmente cómo la métrica se comporta a medida que aumenta el tamaño de la entrada.

Resultados y Conclusiones:

- Se presentan los gráficos junto con la estimación de la complejidad computacional para cada combinación de algoritmo y métrica. Esto proporciona una visión clara de cómo se comporta cada algoritmo en términos de recursos computacionales y cómo este comportamiento escala con el tamaño de la entrada.

Figura 6.30 Resultados evaluación según puntuación ponderada

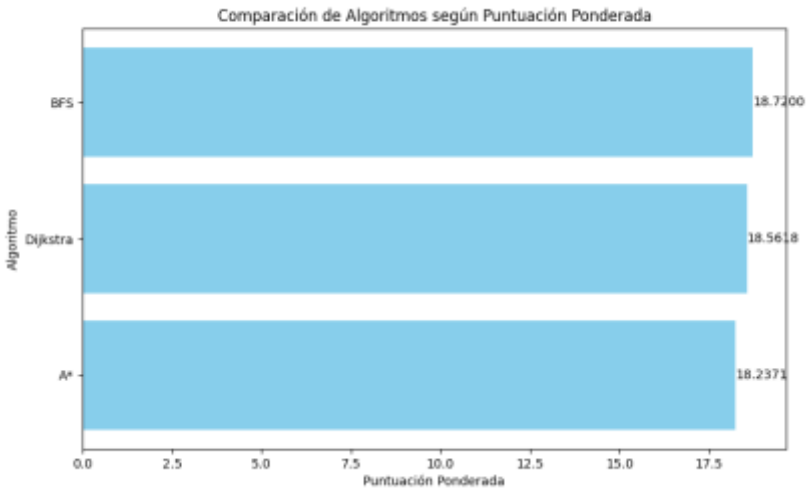

Figura 6.31 Resultados evaluación según puntuación ponderada sin DFS

El análisis detallado de los algoritmos de búsqueda revela que el algoritmo A* sobresale como el mejor en términos generales. A lo largo de los escenarios evaluados, A* mostró el menor tiempo de ejecución y un uso eficiente de los recursos computacionales, manteniendo un equilibrio óptimo entre la rapidez para encontrar soluciones y la eficiencia en el uso de CPU y memoria. Su capacidad para integrar de manera efectiva la heurística con el costo acumulado lo posiciona como la opción más robusta y confiable para la resolución de problemas de búsqueda en mallas complejas, superando a los otros algoritmos en prácticamente todas las métricas clave evaluadas.

6.4.8 Notación Big O

Las siguientes figuras, muestran un gráfico que evalúa el uso de memoria de los algoritmos en función del tamaño del escenario. El eje X representa el "Tamaño" del problema y el eje Y representa la "Memoria" utilizada.

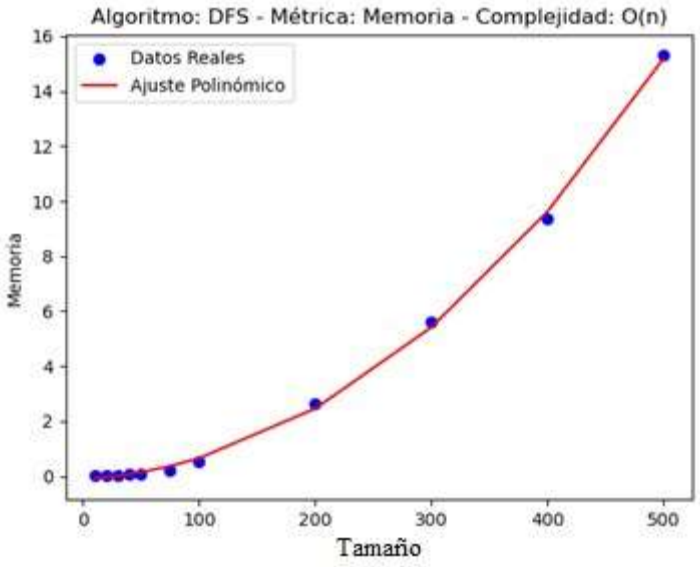

Figura 6.32 Comportamiento de algoritmo DFS

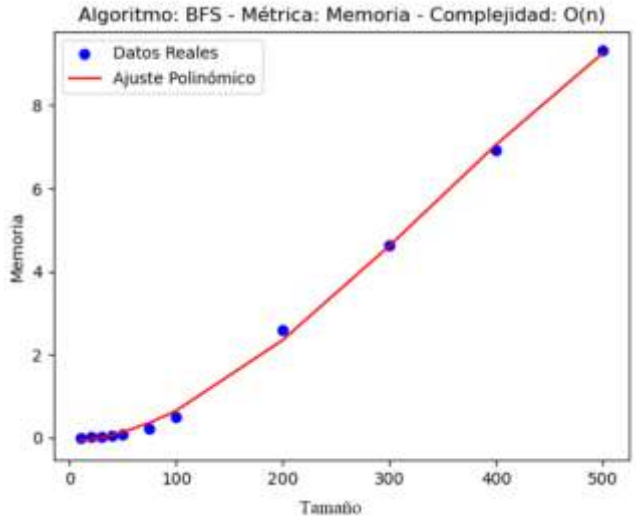

Figura 6.33 Comportamiento de algoritmo BFS

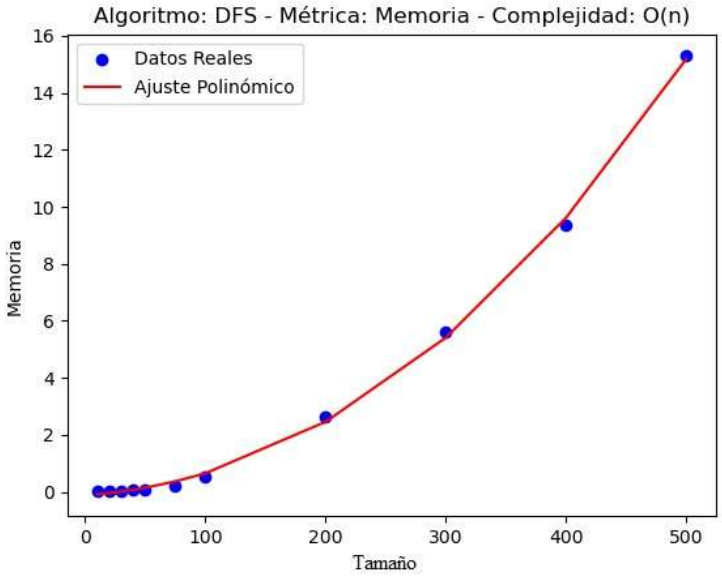

Figura 6.34 Comportamiento de algoritmo Dijkstra

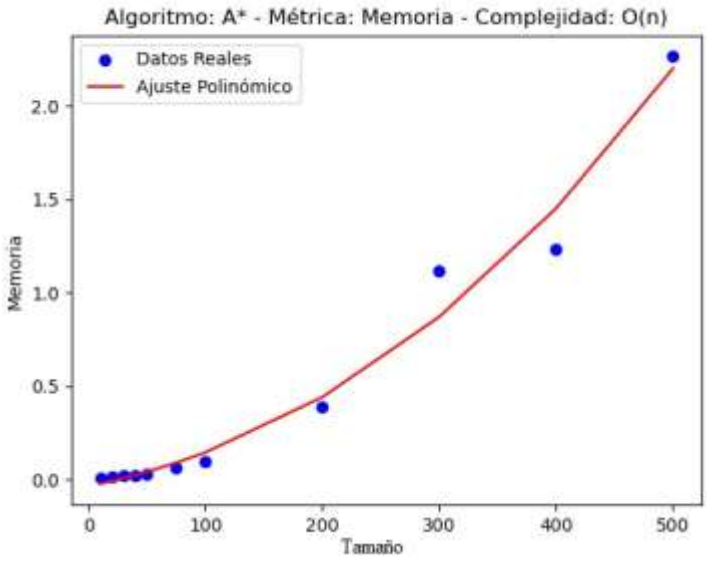

*Figura 6.35 Comportamiento de algoritmo A**

Las cuatro imágenes, muestran la relación entre el tamaño de la malla y el tiempo de ejecución para los algoritmos BFS, A*, Dijkstra y DFS. Todas presentan un comportamiento que sigue una tendencia de *O(n)*, lo que significa que el tiempo de ejecución de cada algoritmo crece de manera lineal conforme aumenta el tamaño de la malla.

6.4.9 Explicación del Comportamiento de *O(n):*

1. BFS (Breadth-First Search):
    - El tiempo de ejecución aumenta casi linealmente con el tamaño de la malla, lo cual es típico para un algoritmo con complejidad *O(n)*. Esto refleja que el número de operaciones necesarias crece proporcionalmente al tamaño de la malla, dado que BFS explora todos los nodos posibles en una malla de este tipo.
2. A*:
    - A* también muestra un crecimiento lineal en el tiempo de ejecución con respecto al tamaño de la malla. Este comportamiento se debe a que A* busca eficientemente el camino más corto, pero su rendimiento sigue estando ligado al tamaño total de la malla, lo que provoca un crecimiento lineal en situaciones donde la heurística es razonablemente eficaz.
3. Dijkstra:
    - Similar a A*, el algoritmo de Dijkstra presenta un aumento lineal en tiempo de ejecución. Dado que Dijkstra evalúa todas las posibles rutas antes de encontrar la óptima, el tiempo de ejecución crece proporcionalmente al número de nodos, lo que explica el comportamiento *O(n)*.
4. DFS (Depth-First Search):
    - El DFS también sigue una tendencia lineal, ya que recorre cada posible camino en profundidad antes de retroceder. Esto implica que, a medida que el tamaño de la malla crece, el número de nodos explorados por DFS se incrementa de manera lineal, llevando a un comportamiento *O(n)*.

Los gráficos confirman que los algoritmos tienen un comportamiento lineal respecto al tamaño de la malla, con una complejidad *O(n)*. Esto sugiere que, aunque la eficiencia de cada algoritmo puede variar según el contexto, todos experimentan un crecimiento similar en tiempo de ejecución cuando el tamaño de la malla aumenta.

6.4.10 Comparación con otros autores

En esta sección se presenta una comparación entre los resultados obtenidos en esta investigación y los resultados reportados en el artículo "Comparative Analysis of Pathfinding Algorithms A*, Dijkstra, and BFS on Maze Runner Game"[22]. La comparación se enfoca en el desempeño de los algoritmos en una malla de 25x25, considerando tres escenarios diferentes.

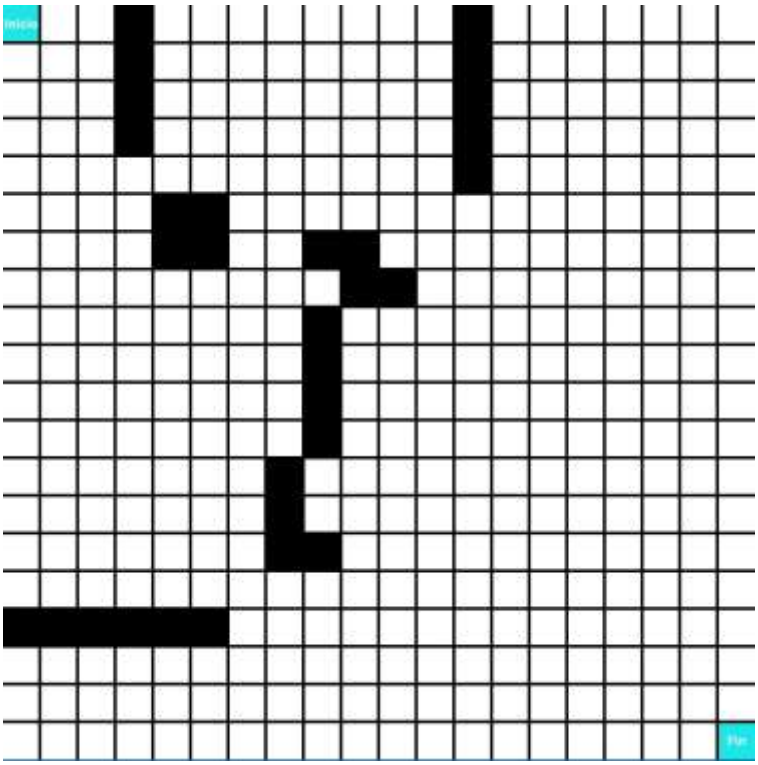

Figura 6.36 Primer escenario a comparar

Tabla 6.13 Comparación de resultados en primer escenario

	Autor			Esta investigación		
Algoritmo	A*	Dijkstra	BFS	A*	Dijkstra	BFS
Pasos	38	38	38	38	38	38
Tiempo [ms]	0.35	0.3	0.8	0.3	0.33	0.35
Bloques computados	323	738	738	103	735	735

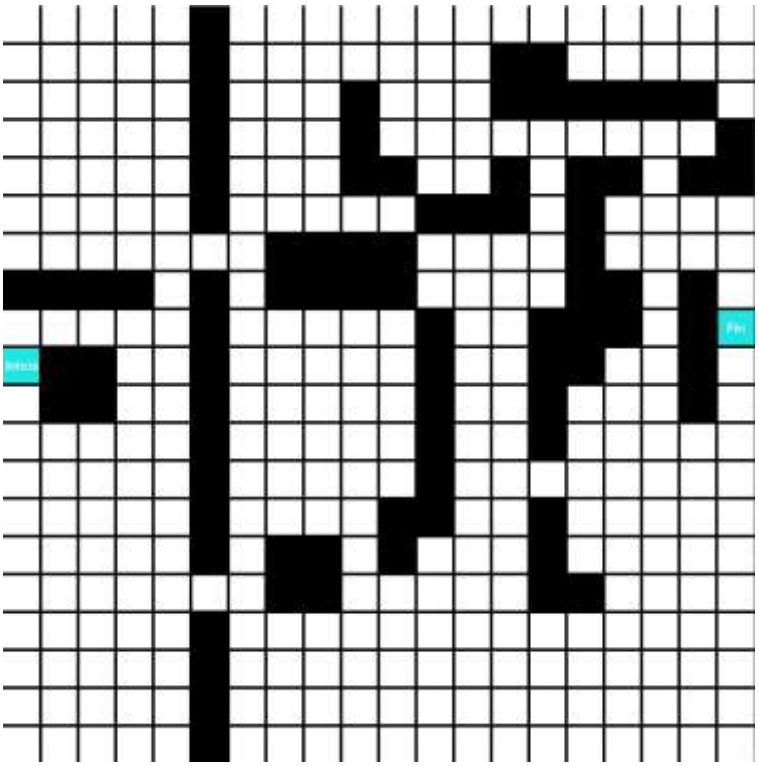

Figura 6.37 Segundo escenario a comparar

Tabla 6.14 Comparación de resultados en segundo escenario

	Autor			Esta investigación		
Algoritmo	A*	Dijkstra	BFS	A*	Dijkstra	BFS
Pasos	34	34	34	34	34	34
Tiempo [ms]	0.4	0.6	0.8	0.3279	0.33	0.33
Bloques computados	415	618	618	201	629	629

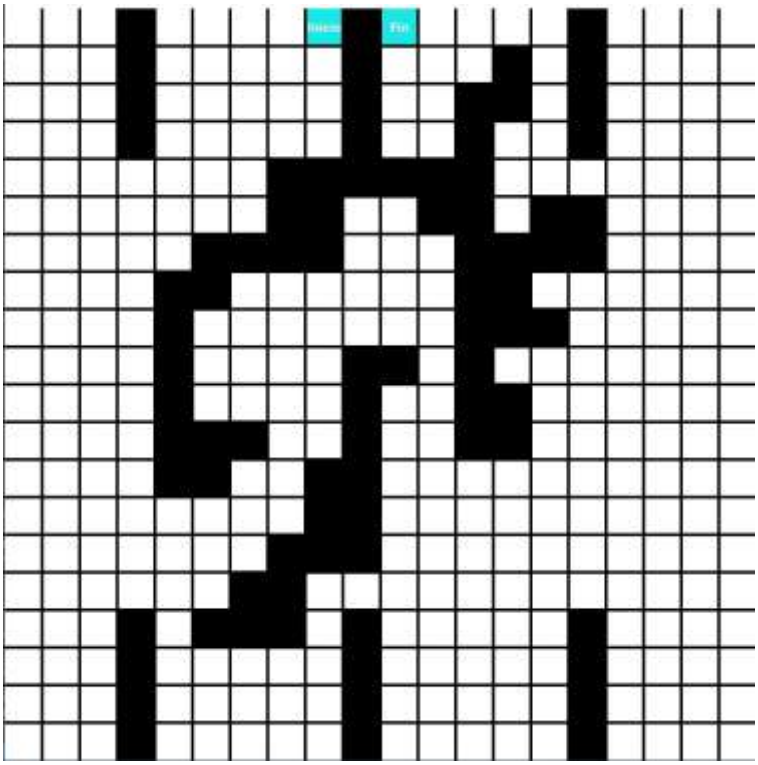

Figura 6.38 Tercer escenario a comparar

Tabla 6.15 Comparación de resultados en tercer escenario

	Autor			Esta investigación		
Algoritmo	A*	Dijkstra	BFS	A*	Dijkstra	BFS
Pasos	58	58	58	58	58	58
Tiempo [ms]	1.1	0.9	1	0.32	0.39	0.38
Bloques computados	504	624	624	389	621	621

Continuando con la comparación de resultados con otros autores a continuación se presenta la siguiente comparativa con el articulo “Comparing Path-Finding Algorithms” [23], las pruebas implican 250 mapas aleatorios en un escenario de 25 x 25.

Tabla 6.16 Comparación de resultados con el articulo “Comparing Path-Finding Algorithms”

	Autor				Esta investigación			
Algoritmo	A*	Dijkstra	BFS	DFS	A*	Dijkstra	BFS	DFS
Pasos	14.08	15.38	14.08	257.32	13.45	13.57	13.44	86.51
Tiempo [ms]	1.3199	7.2498	8.5872	7.5487	1.981	4.441	6.555	4.157

Como se mencionó al inicio de esta investigación, la mayoría de los estudios previos se han centrado principalmente en evaluar el desempeño de los algoritmos de búsqueda en términos de pasos, tiempo de ejecución y número de bloques computados. Sin embargo, la innovación de este estudio radica en la incorporación de nuevas métricas, como el uso de CPU y memoria RAM, para proporcionar una evaluación más completa del rendimiento de los algoritmos en diferentes escenarios. Esta ampliación del enfoque tradicional permite una comprensión más profunda de los recursos computacionales requeridos y cómo estos afectan la eficiencia general de los algoritmos.

6.5 Creación del entorno utilizando Ursina

La incorporación de imágenes para simular texturas es una práctica habitual en la programación de entornos tridimensionales. Estas imágenes se integran en el motor gráfico de Ursina. Es importante mencionar que el uso de texturas es una herramienta fundamental en la creación de ambientes virtuales, ya que permite darle una apariencia más realista a los objetos y elementos presentes en el escenario.

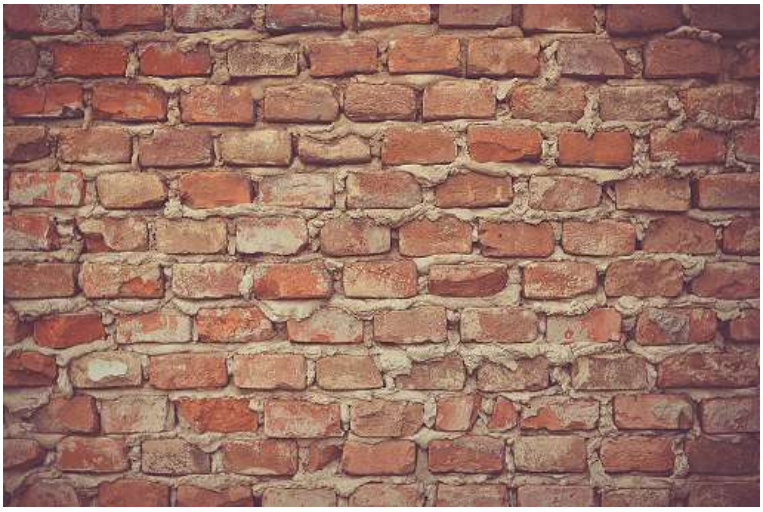

Figura 6.39 Imagen de textura "Ladrillo"

En este caso, se utiliza una textura de ladrillo para los bloques que representan obstáculos. Cada cara del cubo generado se carga con esta textura, lo que implica un uso de memoria y CPU adicional durante la ejecución del programa. Por lo tanto, es importante que los algoritmos de búsqueda de trayectorias sean eficientes para minimizar la carga computacional y lograr una experiencia fluida.

Figura 6.40 Imagen de textura "Césped"

En el proceso de creación de los entornos, la imagen de la figura 6.40 es utilizada para simular el césped en el piso del escenario. Esta textura puede ser observada en toda el área del escenario donde se lleva a cabo la búsqueda de trayectorias.

Figura 6.41 Imagen de textura "Cielo"

La textura del cielo es un elemento importante en la creación de ambientes tridimensionales, ya que ayuda a crear una sensación de espacio y profundidad. En este caso, la imagen de la figura 6.41 es utilizada para simular un cielo despejado, la cual se carga constantemente en el fondo del escenario. Al ser un elemento que está presente durante toda la ejecución del

programa, es necesario que su carga sea lo más eficiente posible para no afectar el rendimiento de la búsqueda de trayectorias.

Figura 6.42 Entidad "Cubo"

La figura 6.42 muestra la primera estructura que se utiliza para construir el escenario y se utiliza para realizar pruebas de funcionamiento. Esta estructura funciona como piso en el escenario y se puede observar el código utilizado en el anexo 1. Es importante mencionar que, aunque esta estructura no se utiliza en la versión final del escenario, es fundamental en la etapa de pruebas y en la optimización del algoritmo de búsqueda de trayectorias.

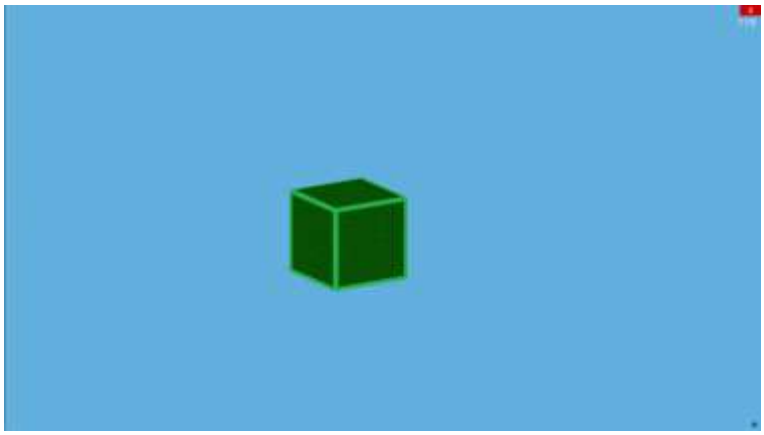

Figura 6.43 Entidad cubo con textura "Césped"

Posteriormente cómo se puede apreciar en la figura 6.44; el objetivo de utilizar la imagen de la figura 6.39 en el cubo de la figura 6.43 es para añadir una textura de césped y crear un ambiente más amigable en el escenario. Esto permite la creación de un entorno más grande con obstáculos, lo que a su vez posibilita realizar pruebas de búsqueda de trayectorias en un escenario más complejo. El código utilizado para cargar la textura de césped se puede encontrar en el anexo 2.

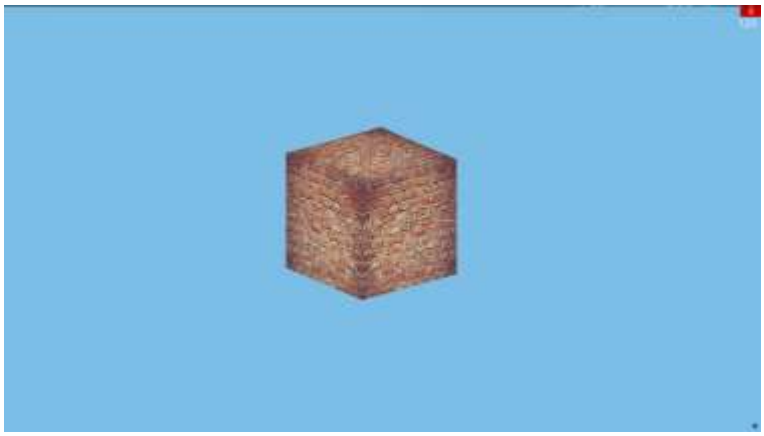

Figura 6.44 Entidad cubo con textura "ladrillo"

Para los obstáculos se utiliza otra textura, esta sirve para diferenciar los obstáculos y el escenario en el cual se puede realizar el recorrido.

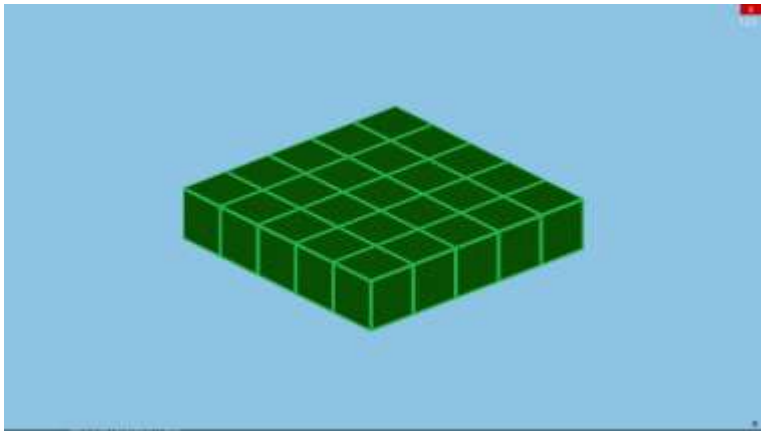

Figura 6.45 Escenario con formas cubicas.

En la figura 6.45 se presenta la adición de texturas de césped y ladrillo a los cubos que conforman el entorno. Además, se incorporan obstáculos para aumentar la complejidad del escenario y poner a prueba los algoritmos de búsqueda de trayectorias. Los obstáculos se generan colocando cubos en posiciones aleatorias, lo que hace necesario un proceso de detección de colisiones para evitar que los objetos se superpongan.

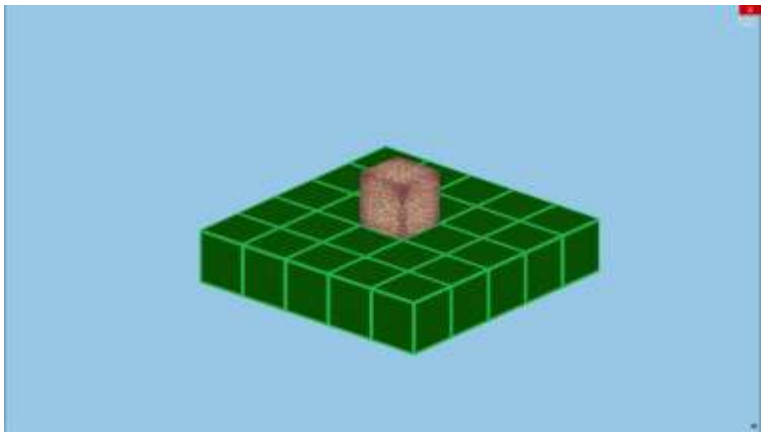

Figura 6.46 Obstáculo en escenario.

Una vez creado el escenario con entidades cúbicas, se procede a añadir obstáculos al mismo. Estos obstáculos se distinguen del resto del entorno gracias a la textura de ladrillo que se les aplica. De esta manera, se logra diferenciar las áreas donde se puede transitar del resto del escenario donde no es posible avanzar. Esta distinción resulta fundamental para la correcta ejecución del programa, ya que permite que el algoritmo de búsqueda de trayectorias pueda operar de forma eficiente y efectiva.

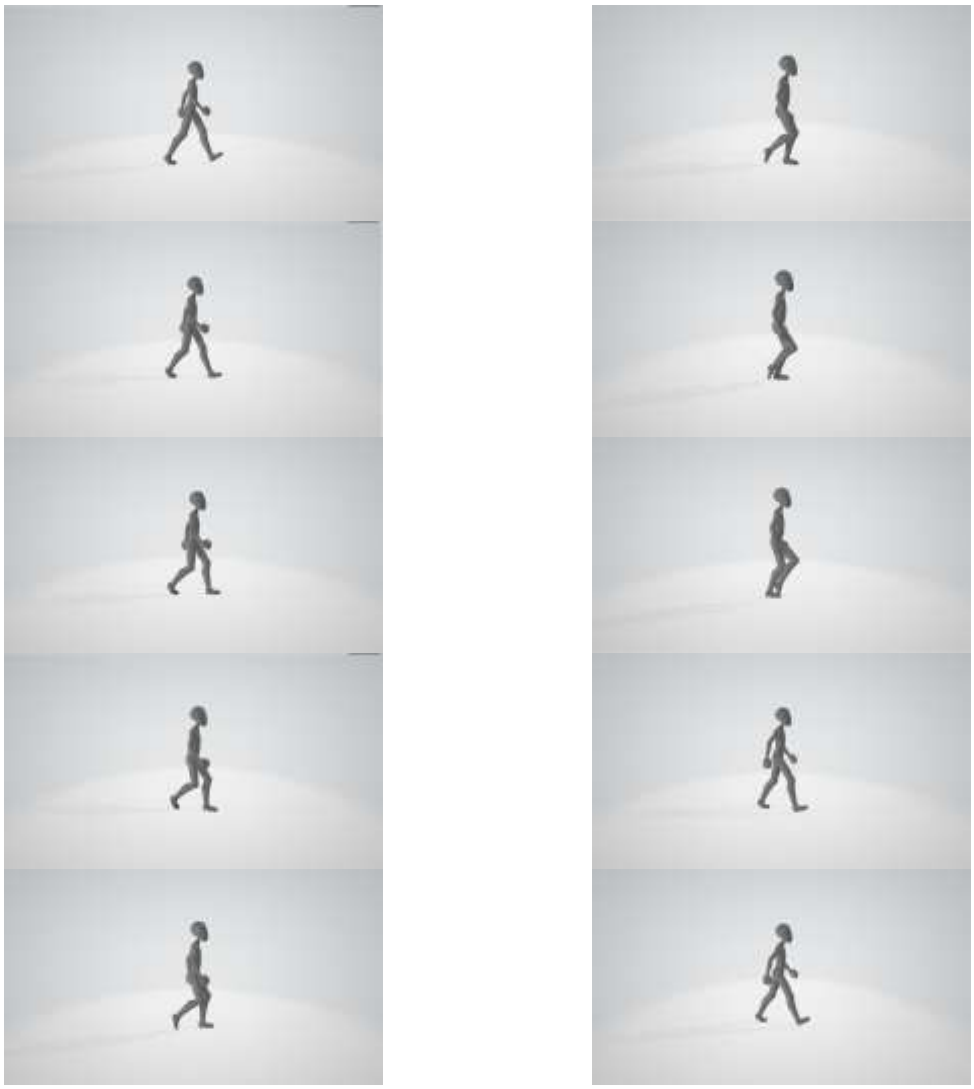

Figura 6.47 Imágenes para dar una sensación de movimiento al NPC

La anterior figura presenta ejemplos de imágenes empleadas para proporcionar una perspectiva de movimiento al personaje no jugador. Estas imágenes resultan fundamentales para generar una ilusión de movimiento en el personaje, logrando así una apariencia fluida.

Después de elegir el algoritmo con el mejor rendimiento computacional, se crea el escenario bidimensional con elementos gráficos tridimensionales utilizando la librería Ursina en el lenguaje de programación Python. La primera opción elegida para la creación del escenario es mediante la implementación de un escenario con figuras cúbicas que sirven como suelo para el personaje jugador.

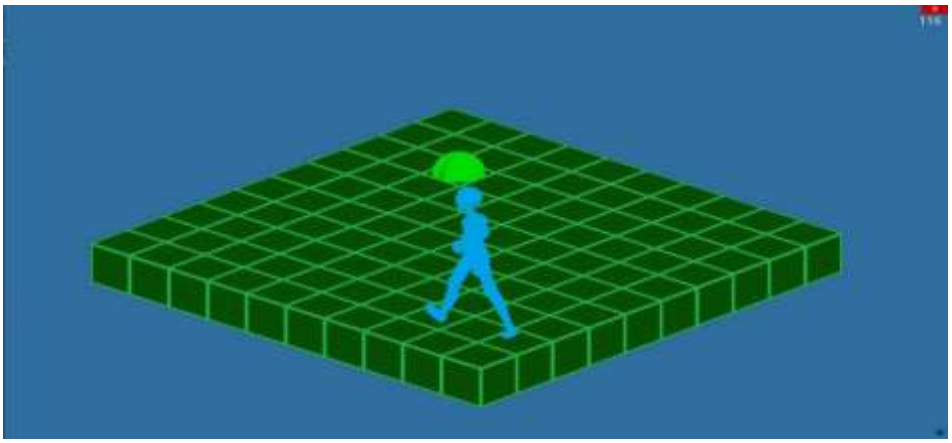

Figura 6.48Escenario de 10x10 bloques, piso con cubos

El uso de entidades cúbicas sin obstáculos es una opción para las primeras pruebas de funcionamiento del programa. En este tipo de escenarios se puede verificar el correcto desempeño del algoritmo y la capacidad de movimiento del personaje no jugador en un ambiente simple y controlado. Sin embargo, posteriormente se incluyen obstáculos para simular un entorno desafiante para el personaje jugador.

Si se requiere ver la implementación de un cubo en el lenguaje Python con la librería Ursina consultar Anexo 1.

6.5.1 Escenario de 10x10 sin obstáculos

La inclusión de una tabla simulando el escenario a construir se realiza con el propósito de proporcionar una representación visual y organizada de los componentes y características clave del entorno. Esta tabla servirá como una herramienta gráfica que facilitará la planificación, el diseño y la comprensión de los elementos estructurales del escenario, como las figuras cúbicas, el terreno, y cualquier otro detalle relevante.

Tabla 6.17 Escenario 10x10

3	3	3	3	3	3	3	3	3	3
3	0	0	0	0	0	0	0	0	3
3	0	0	0	0	2	0	0	0	3
3	0	0	0	0	0	0	0	0	3
3	0	0	0	0	0	0	0	0	3
3	0	0	0	0	0	0	0	0	3
3	0	0	1	0	0	0	0	0	3
3	0	0	0	0	0	0	0	0	3
3	0	0	0	0	0	0	0	0	3
3	3	3	3	3	3	3	3	3	3

En la tabla 14 se muestra la definición con diferentes números para los objetos que se posicionaron en el escenario, siendo de la siguiente manera el significado para cada número en el escenario:

a. Número 1 – Personaje no jugador
b. Número 2 – Objetivo
c. Numero 3 – obstáculos en el escenario
d. Numero 0 – Bloques por los cuales se puede realizar un recorrido.

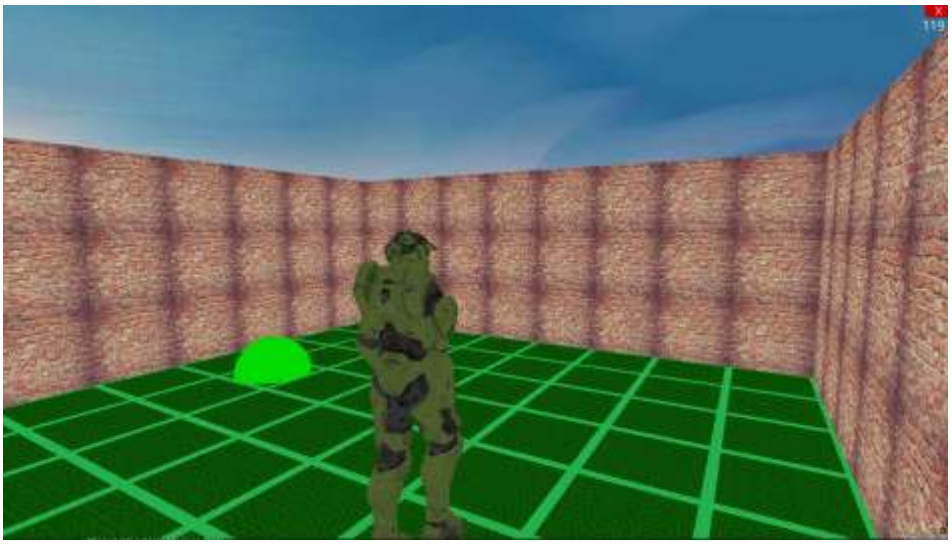

Figura 6.49 Vista desde el suelo de un escenario de 10x10 sin obstáculos

En la figura anterior se presenta una vista desde el suelo, en el cual se aprecia el punto de interés, el personaje no jugador, y los obstáculos, al ser un escenario pequeño las imágenes por segundo se encuentran en 119, esto significa que el escenario se mueve con fluidez.

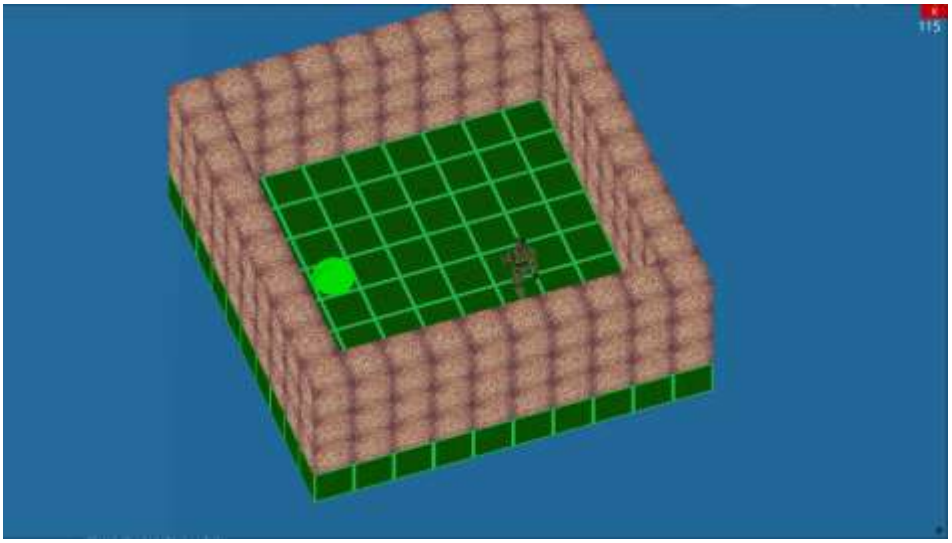

Figura 6.50 Vista aérea de los primeros escenarios de 10x10

La figura 6.50 muestra una vista detallada del escenario desde otro ángulo, en la cual se pueden apreciar el personaje no jugador, el objetivo al cual debe llegar, los bloques obstáculos y el piso del escenario.

6.5.2 Escenario de 10x10 con obstáculos

Ahora, se procede con la creación de una tabla, Tabla 15, que simule el escenario, incorporando la presencia de obstáculos. Esta tabla proporcionará una representación detallada y estructurada de cómo se distribuirán y posicionarán los obstáculos en el entorno. Al documentar estos obstáculos en la tabla, se proporcionará una guía para la implementación de las características específicas del entorno

Tabla 6.18 Escenario 10x10 con obstáculos.

3	0	3	3	3	3	0	3	3	3
3	0	0	0	0	0	3	0	3	0
0	3	1	3	0	0	0	3	0	0
3	0	0	0	0	3	0	0	0	0
0	3	3	0	0	3	3	3	0	0
3	0	0	0	0	3	0	0	0	0
3	0	3	0	3	0	0	0	0	0
0	3	3	0	3	3	0	0	0	2
0	3	0	0	0	0	0	3	3	3
3	3	3	0	0	3	0	3	3	0

En la tabla presentada, siendo un escenario de 10x10 bloques, la disposición de los objetos y facilita la identificación clara de posibles soluciones.

Figura 6.51 Vista desde el suelo de un escenario con obstáculos.

Por otro lado, en la figura 6.51 se puede observar el escenario con obstáculos desde una perspectiva más a nivel del suelo, en este momento no es posible identificar el punto de interés, lo cual dificulta la visualización del recorrido del personaje no jugador por el escenario.

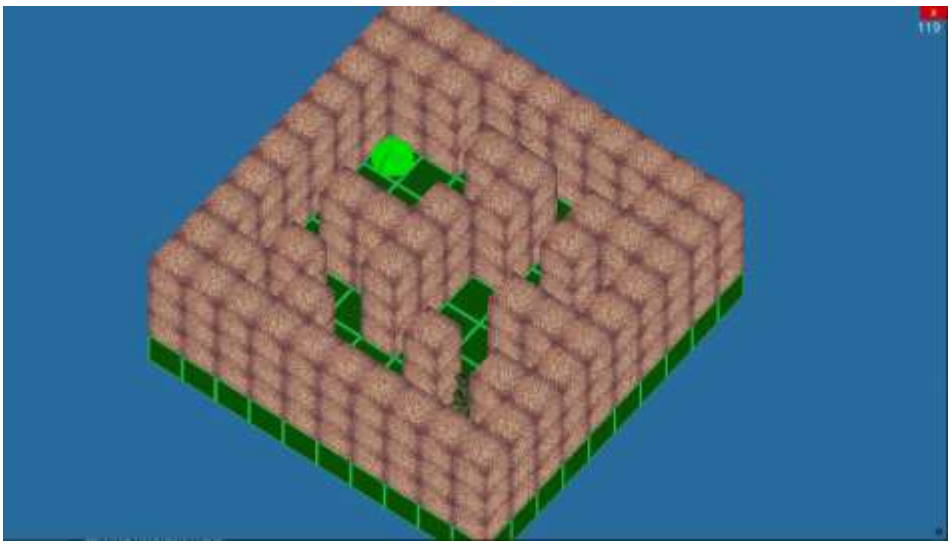

Figura 6.52 Vista aérea de los primeros escenarios de 10x10 con obstáculos.

Además, en la figura 6.52 se muestra una vista aérea del escenario, donde se aprecia claramente el punto meta y el personaje no jugador. Cabe destacar que, aunque este escenario es relativamente pequeño, los FPS se mantuvo en un nivel alto, alcanzando los 119.

6.5.3 Escenario de 20x20 sin obstáculos

Continuando con la muestra de los escenarios, en esta etapa se muestra el escenario de 20x20 bloques sin obstáculos.

Tabla 6.19 Escenario de 20x20 bloques sin obstáculos

3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3
3	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	3
3	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	3
3	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	3
3	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	3
3	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	3
3	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	3
3	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	3
3	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	2	3
3	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	3
3	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	3
3	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	3
3	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	3
3	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	3
3	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	3
3	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	3
3	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	3
3	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	3
3	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	3
3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3

En la tabla 16 se puede observar el escenario de 20x20, y en la figura 6.53 se puede apreciar un ambiente abierto complejo sin la presencia de obstáculos.

En este escenario se buscó aumentar el consumo de memoria y uso de CPU y se implementaron diferentes texturas para los obstáculos con el objetivo de dar una apariencia

más realista al escenario. Se realizaron diversas pruebas en este escenario y se comprueba que el programa tenía un buen desempeño con una tasa de FPS de 111 en promedio.

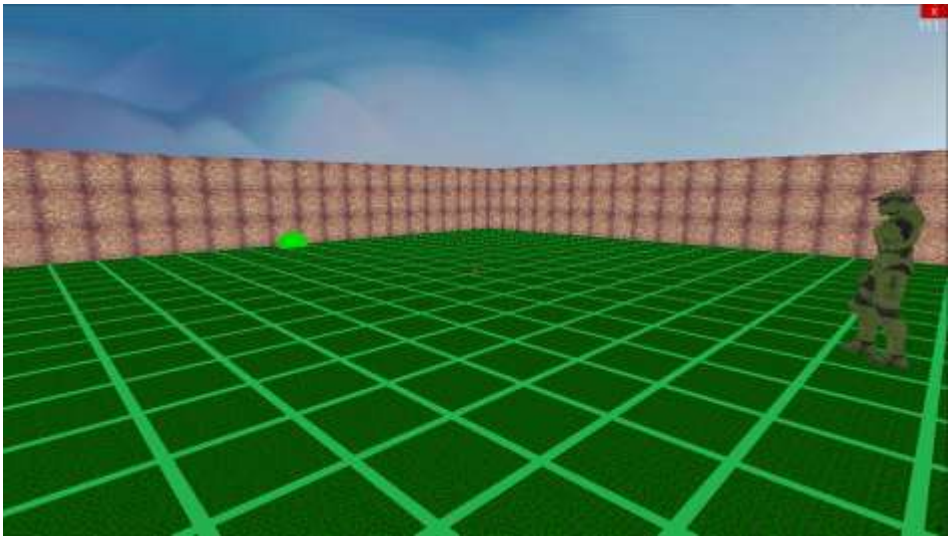

Figura 6.53 Vista desde el suelo de un escenario de 20x20

En la figura 6.54 se puede apreciar la perspectiva desde el lado izquierdo del NPC, teniendo el objetivo meta en una línea recta, aunque puede parecer que es una solución sencilla, el cálculo de la trayectoria tiene la misma complejidad que si estuviera en el fondo del escenario.

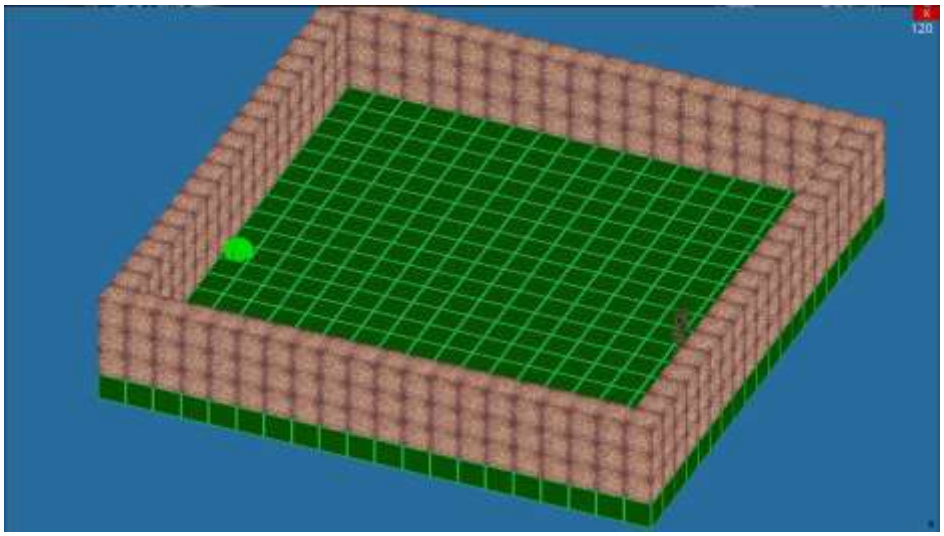

Figura 6.54 Vista aérea de un escenario de 20x20

6.5.4 Escenario de 20x20 con obstáculos

A continuación, tenemos la respectiva tabla del mismo tamaño de escenario, pero esta vez con obstáculos.

Tabla 6.20 Escenario de 20x20 bloques con obstáculos

0	3	0	0	0	0	3	3	3	0	0	0	0	0	0	0	0	3	3	0
0	0	3	0	3	0	0	3	0	3	0	0	3	0	0	3	0	0	0	0
0	3	0	0	0	3	3	3	0	3	0	3	3	3	0	3	0	0	3	0
0	0	0	3	0	0	0	3	0	0	0	0	0	0	0	0	3	0	0	3
0	0	0	0	0		0	3	3	3	0	0	3	0	0	0	0	0	0	0
3	0	3	0	0	0	0	0	0	0	0	0	0	0	0	3	0	3	0	0
3	0	0	3	0	0	0	0	0	0	0	0	3	0	0	0	0	0	0	0
0	0	0	0	3	0	3	3	0	0	0	3	0	0	3	3	0	0	0	0
0	0	1	3	0	3	0	0	0	0	0	3	3	0	3	3	0	0	2	3
0	3	0	3	3	0	3	0	0	0	0	0	0	0	3	0	0	0	0	3
3	3	0	0	0	0	0	3	0	0	0	3	0	0	0	0		0	0	0
0	3	3	0	0	0	0	0	0	0	0	0	3	0	0	3	0	3	0	0
0	0	3	0	0	0	0	0	0	3	0	0	3	3	0	3	3	3	0	3
0	0	0	0	0	3	0	0	0	3	0	3	0	3	0	0	0	3	0	3
0	0	0	0	3	3	0	0	0	0	0	3	3	0	0	0	0	0	0	0
0	0	3	0	0	0	0	0	3	3	3	3	0	3	0	3	0	0	0	3
3	0	0	3	3	3	0	0	0	3	3	0	0	3	0	3	0	3	3	0
3	3	0	3	3	3	3	3	3	0	3	3	0	0	0	3	3	3	0	0
0	0	0	0	0	3	0	3	0	3	3	0	0	3	0	3	0	0	3	3
3	0	0	3	0	3	0	0	3	3	0	0	3	0	3	0	0	0	3	3

El escenario de 20x20 cómo se puede apreciar en las figuras 6.55 y 6.56 es una versión avanzada del escenario anterior, donde se han agregado obstáculos para aumentar el nivel de dificultad. A pesar de los obstáculos, los FPS ha mejorado significativamente y ahora se encuentra en 98, lo que indica una mejor optimización del programa. Este escenario es ideal para pruebas más avanzadas en un entorno más complejo.

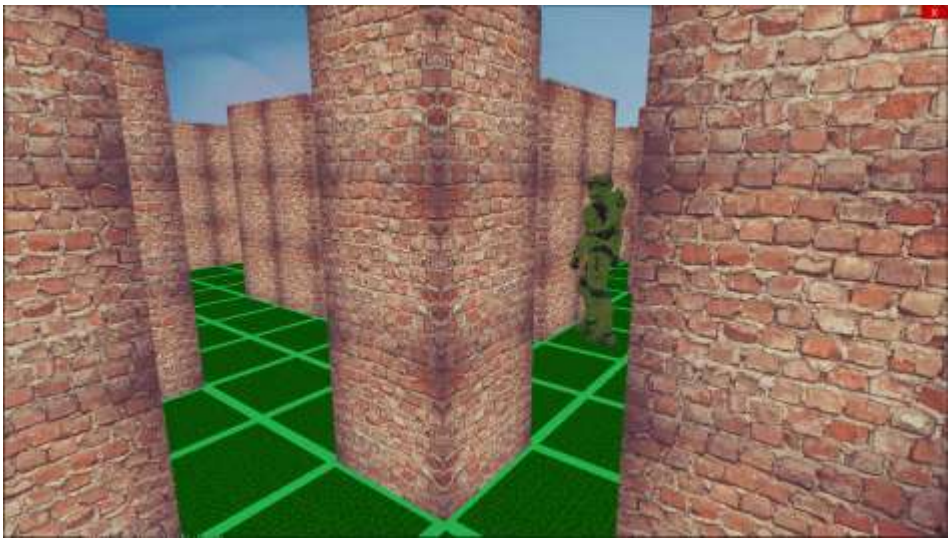

Figura 6.55 Vista desde el suelo de un escenario de 20x20 con obstáculos

En la figura 6.56 se puede apreciar claramente la presencia de bloques de obstáculos que dificultan el recorrido del personaje no jugador.

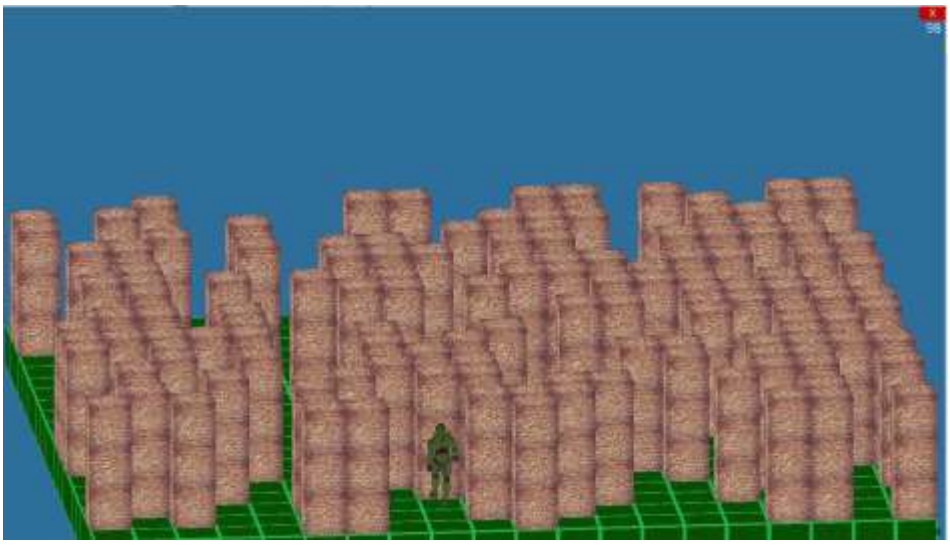

Figura 6.56 Vista aérea de un escenario de 20x20 con obstáculos

Desde esta perspectiva no es posible visualizar el objetivo que se trata de hallar en la búsqueda de la trayectoria.

6.5.5 Escenario de 30x30 sin obstáculos

Continuando con la muestra de los diferentes escenarios ahora se muestra el siguiente, el cual está compuesto por una malla de 30x30 bloques, para las primeras pruebas en este tamaño se optó por realizar la prueba sin obstáculos para tener el parámetro de cuanto consumo de memoria y cuantas imágenes por segundo se podría llegar a mostrar.

Tabla 6.21 Escenario de 30x30 bloques sin obstáculos

3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3
3	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	3
3	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	3
3	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	3
3	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	3
3	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	3
3	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	3
3	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	3
3	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	3
3	0	0	0	0	0	0	2	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	3
3	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	3
3	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	3
3	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	3
3	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	3
3	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	3
3	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	3
3	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	3
3	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	3
3	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	3
3	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	3
3	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	3
3	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	3
3	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	3
3	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	3
3	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	3
3	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	3
3	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	3
3	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	3
3	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	3
3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3

En el siguiente escenario se puede observar una textura de césped en el suelo y una perspectiva desde el suelo para visualizar la distribución de las entidades cúbicas que conforman el escenario.

Durante las pruebas realizadas en este escenario, se logró una tasa de cuadros por segundo (FPS) de 78, lo que indica un desempeño adecuado del programa en la generación y visualización del escenario. Cabe destacar que este escenario es utilizado durante las primeras pruebas para la validación del correcto funcionamiento del programa.

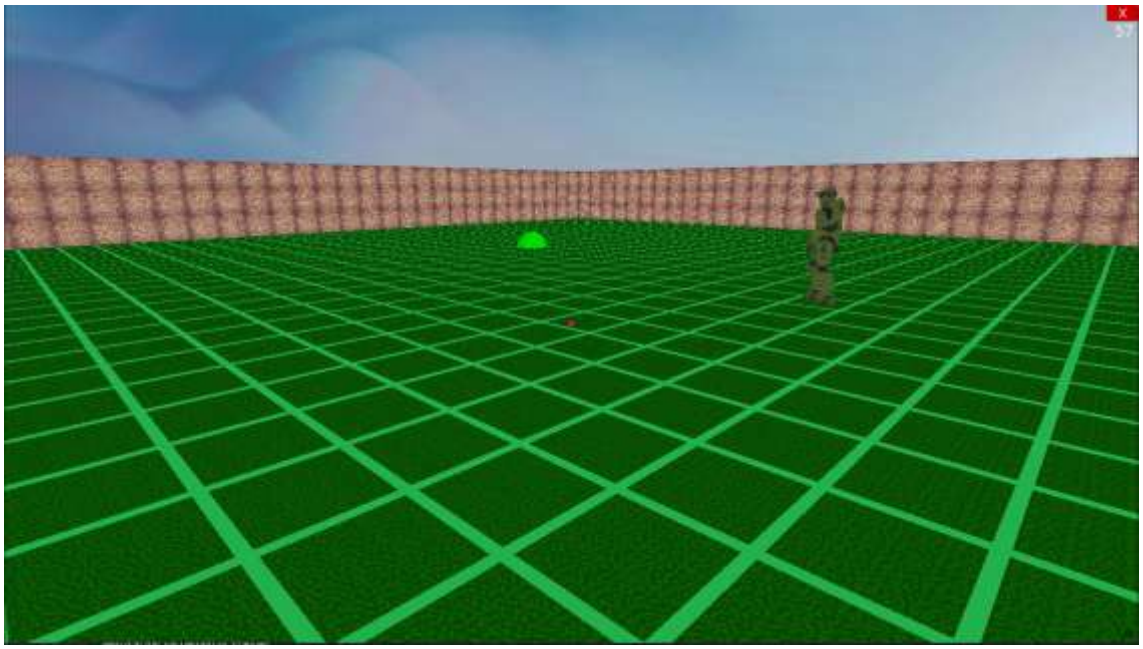

Figura 6.57 Vista desde el suelo de un escenario de 30x30 con obstáculos

A medida que se avanza en el experimento se identifica que los FPS disminuyen cuando se aumenta el tamaño del escenario, para este experimento no influye la disminución de FPS ya que solamente se está cargando el escenario y algunos obstáculos.

6.5.6 Fotogramas por segundo en Ursina

Durante la ejecución de los escenarios es importante tener en cuenta los fotogramas por segundo (FPS), que se refieren a la cantidad de imágenes por segundo que muestra el escenario. Durante las pruebas realizadas, se obtiene un máximo de 116 FPS en los escenarios de 10x10 y 20x20 bloques. Es importante destacar que mientras mayor sea la cantidad de FPS obtenidos, mayor será la fluidez con la que el NPC se moverá hacia el objetivo. Por lo tanto, se realiza un esfuerzo en la optimización del código y la utilización de librerías para la generación del escenario y lograr la mayor cantidad de FPS posibles durante la ejecución del programa.

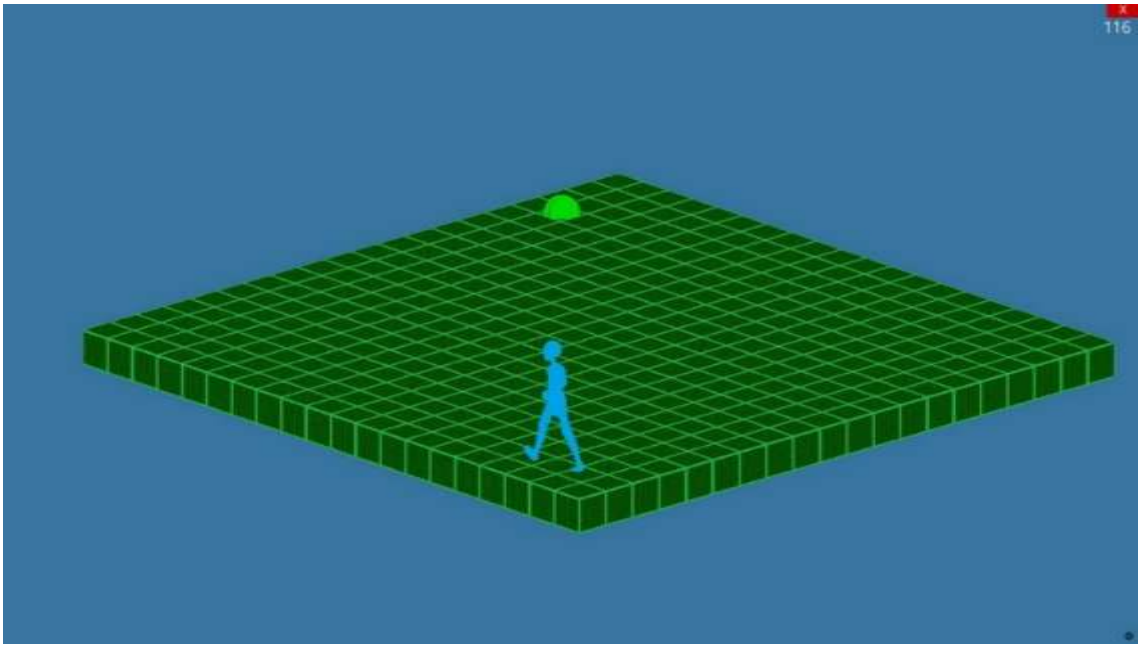

Figura 6.58 Escenario de 20x20 bloques, piso con cubos y 116 FPS

Durante la experimentación, se ha observado que a medida que se aumenta el tamaño del escenario, los FPS disminuyen. En la figura presentada se muestra un escenario de 30x30 bloques en el que se obtuvo una tasa de 96 FPS. Aunque aún se mantiene una tasa de FPS razonable, la disminución es notable en comparación con los escenarios de menor tamaño.

Esta disminución puede tener un impacto en la fluidez de los movimientos del NPC, lo que puede afectar el rendimiento de la búsqueda de trayectorias.

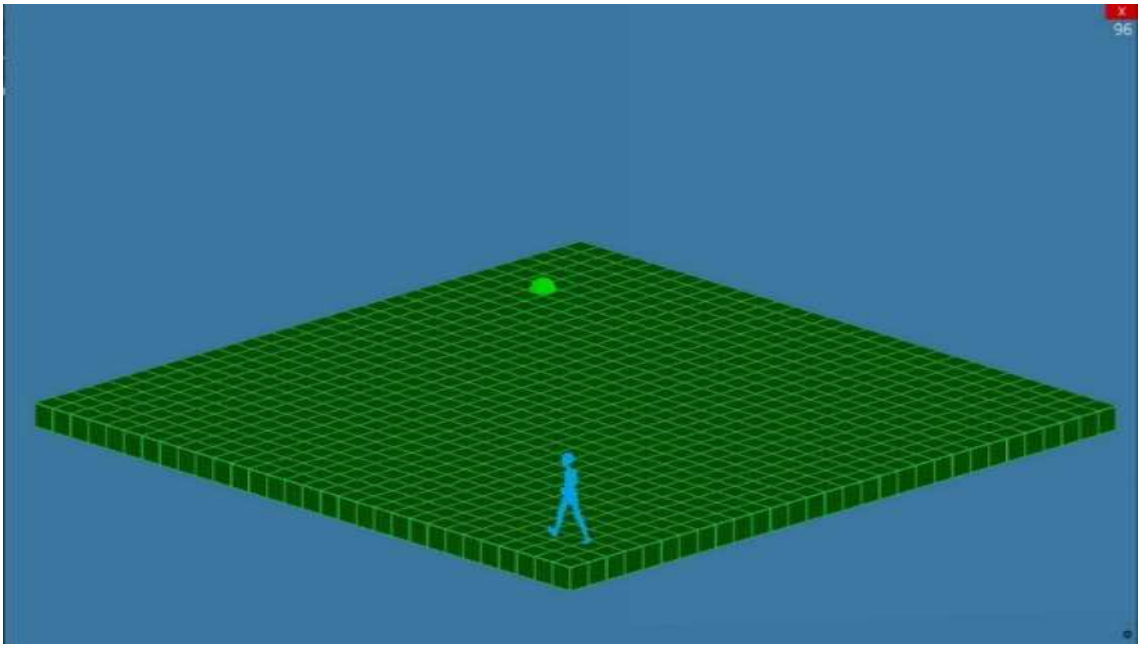

Figura 6.59 Escenario de 30x30 bloques, piso con cubos y 96 FPS

Al continuar aumentando el tamaño del escenario, como se puede observar en la implementación de 50x50 bloques, los FPS disminuyeron aún más drásticamente a 33 FPS. Esto provoca que la visualización de la búsqueda de la trayectoria se volviera difícil de seguir, ya que el NPC en su movimiento a veces no mostraba completamente el recorrido al punto meta.

Es importante considerar que la reducción en los FPS también puede impactar en la capacidad de procesamiento de la computadora y la eficiencia del algoritmo de búsqueda utilizado.

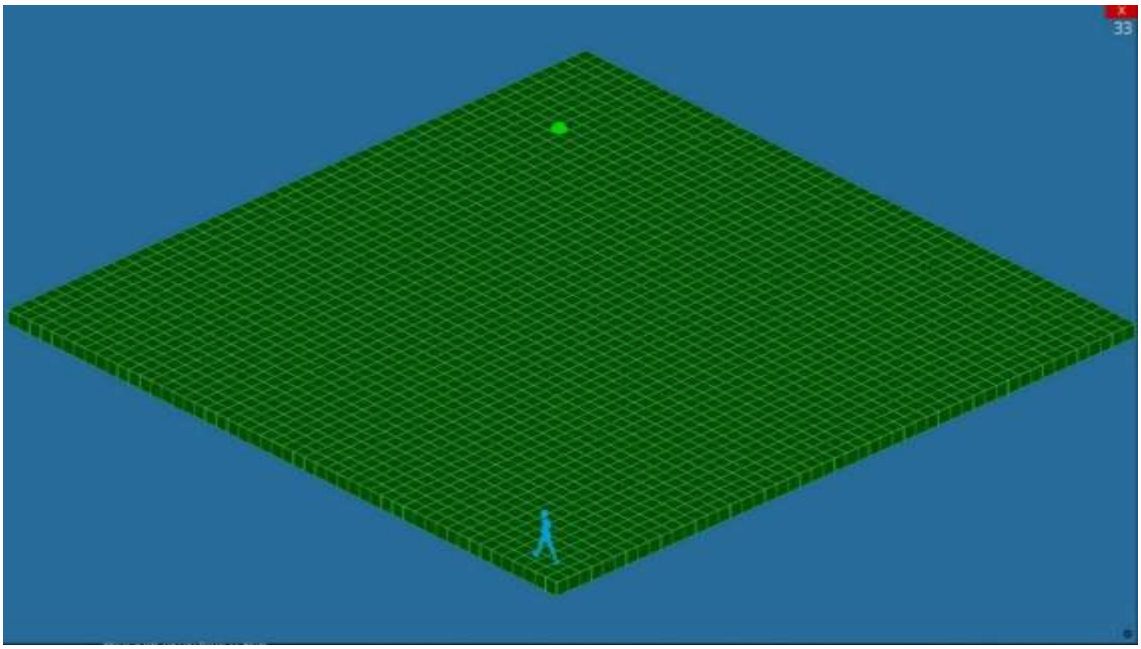

Figura 6.60 Escenario de 50x50 bloques, piso con cubos

El problema con la disminución de FPS no solo afecta al escenario de 50x50 bloques, sino también a los escenarios posteriores. La visualización del recorrido del NPC es cada vez más difícil, llegando al punto en el que es casi nula. Incluso al intentar cambiar el ángulo de la cámara para tener una mejor vista del escenario, en el programa es difícil de controlar debido a que el movimiento de la cámara no corresponde con el deseado en el escenario. Estas dificultades impiden la implementación de otros algoritmos como la búsqueda con múltiples objetivos o la búsqueda de trayectorias en tiempo real en escenarios de un tamaño mayor a 30 bloques.

En el experimento, se observa que la fluidez del movimiento del NPC es nuevamente afectada negativamente en escenarios de mayor tamaño. En la figura 6.60, se puede apreciar cómo la colocación del punto objetivo en el extremo del escenario de 100x100 bloques afecta su fluidez. Es importante considerar la relación entre el tamaño del escenario y los FPS para lograr una implementación eficiente y óptima del movimiento del NPC.

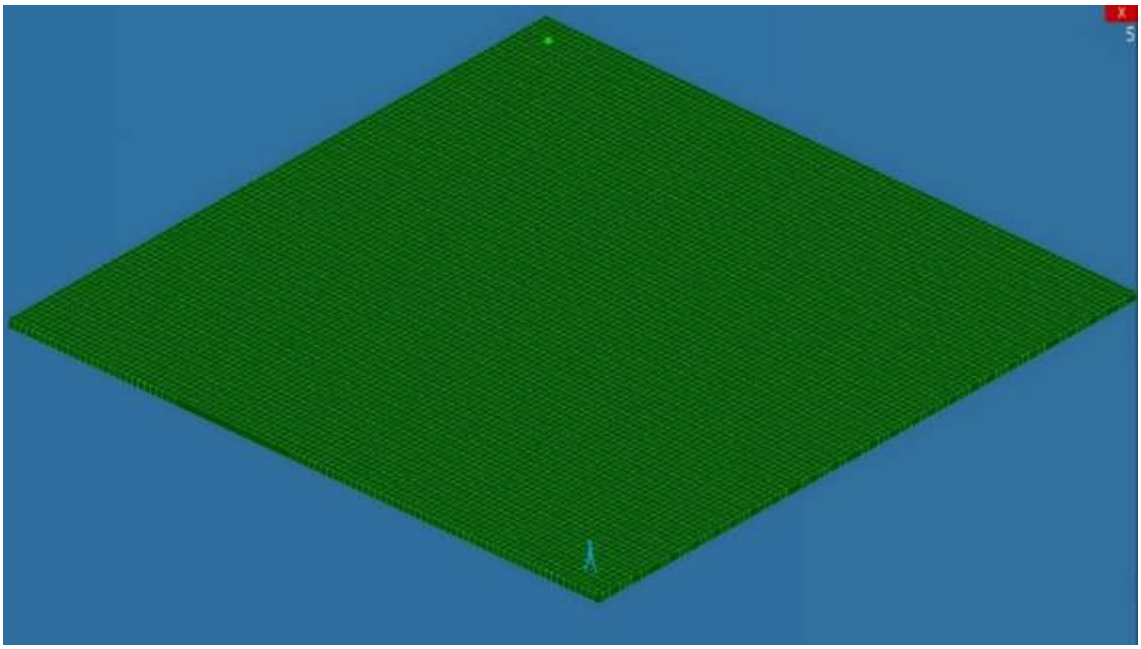

Figura 6.61 Escenario de 100x100 bloques, piso con cubos

En vista de que se presenta una baja de FPS en el escenario de 100x100 bloques y en el de 150x150 bloques, es necesario buscar una solución que permita mejorar el desempeño del motor gráfico. Esta situación genera la necesidad de realizar una investigación exhaustiva para determinar cuál era el problema y cómo se podía solucionar.

Después de varios análisis, se encuentra que la baja de FPS se debe a la carga de texturas en tiempo real, lo cual consume una gran cantidad de recursos del sistema. Para resolver este problema se implementa una técnica de precarga de texturas, así la carga se hace de manera anticipada y no en tiempo real, lo cual mejora significativamente el desempeño del motor gráfico.

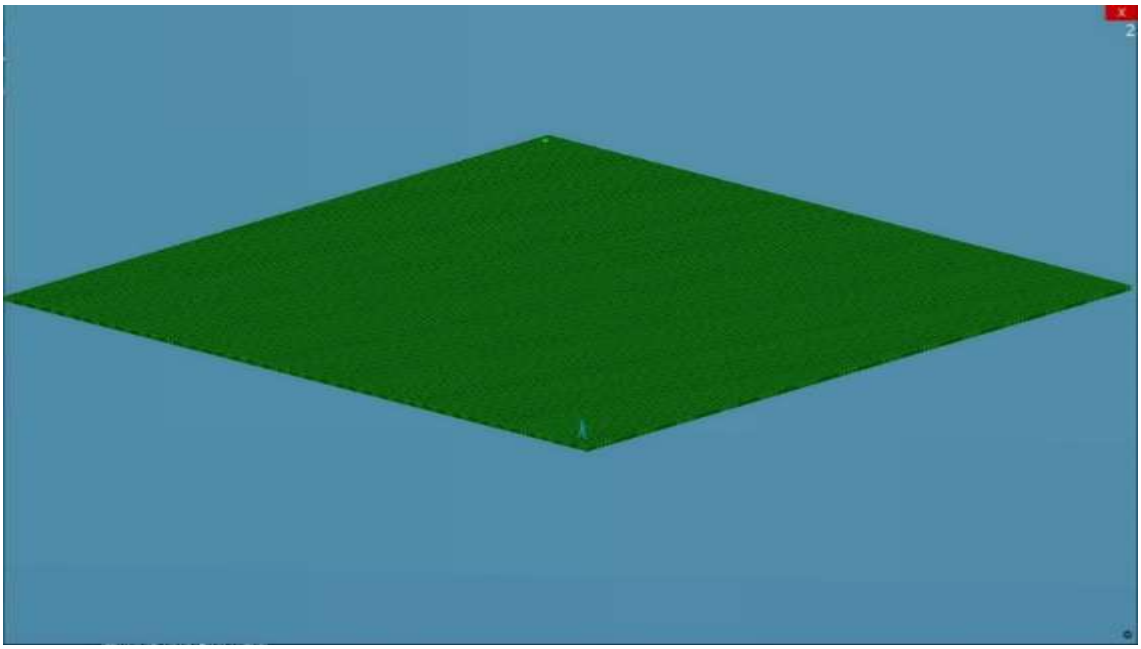

Figura 6.62 Escenario de 150 x 150 bloques, piso con cubos

Gracias a esta solución, se logra que los escenarios de 100x100 bloques y de 150x150 bloques tengan un mejor rendimiento, lo que a su vez proporciona una mejor visualización del recorrido del NPC en la búsqueda de trayectorias.

6.5.7 Carga de imagen como escenario

La implementación de la creación del escenario mediante la importación de una imagen existente permite obtener la misma cantidad de FPS en la generación del escenario de 10x10 bloques y el escenario de 100x100 bloques, 118 FPS en cada escenario. Además, el uso de imágenes reemplazando la generación de cubos como piso, permite generar escenarios más complejos y detallados que no podrían haber sido generados utilizando bloques individuales, si se generan los escenarios con la carga de cubos implica un consumo excesivo de los recursos computacionales disponibles. Esto da lugar a una mejora significativa en la calidad visual del escenario y la experiencia de usuario al utilizar el programa.

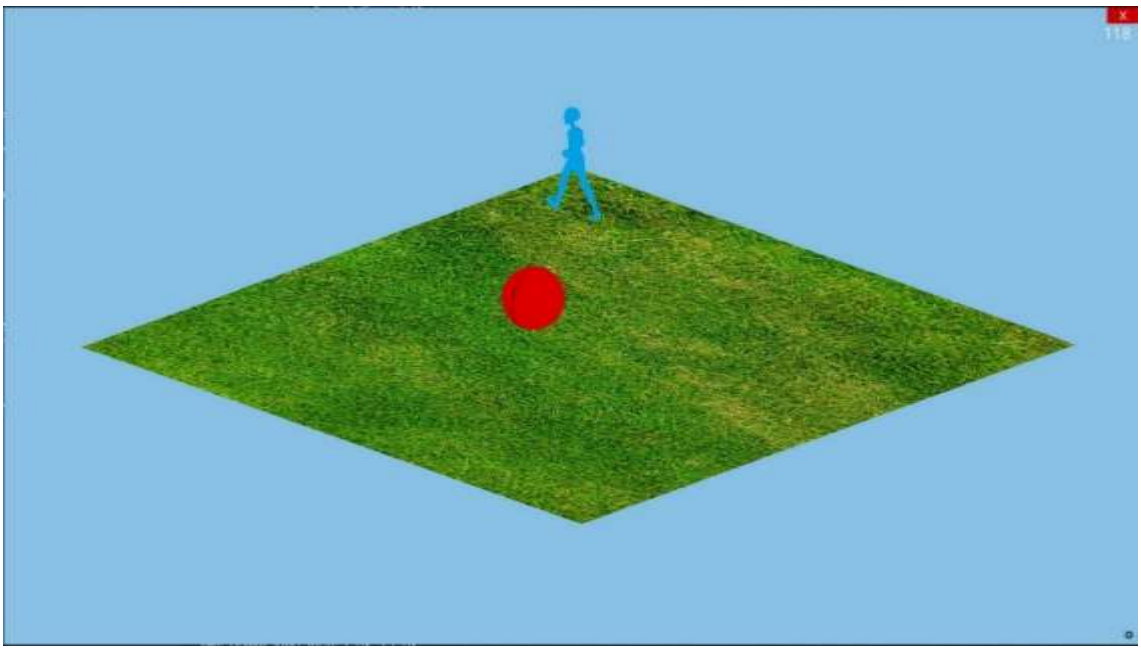

Figura 6.63 Escenario de 10x10 bloques, piso importando imagen

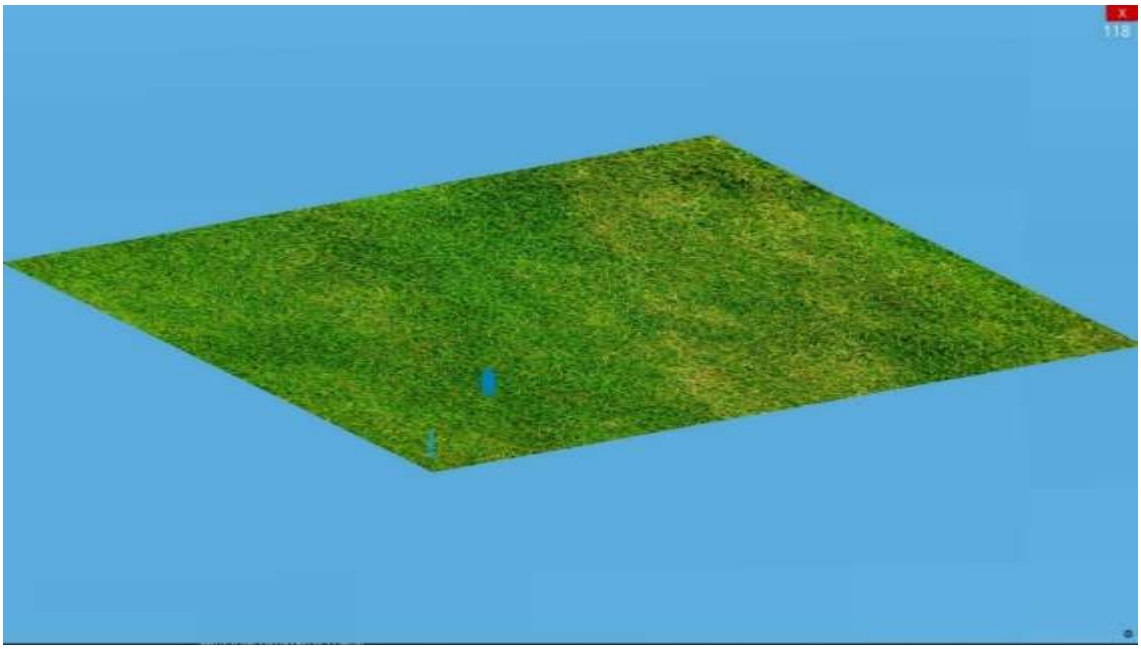

Figura 6.64 Escenario de 100 x 100 bloques, piso con textura de césped

6.6 Implementación de escenario con obstáculos

Una vez definidos los elementos necesarios para la construcción del escenario, como son el tamaño del escenario, obstáculos, punto de interés y posición de NPC, se genera un escenario y se ejecuta la búsqueda de la trayectoria utilizando el algoritmo A*.

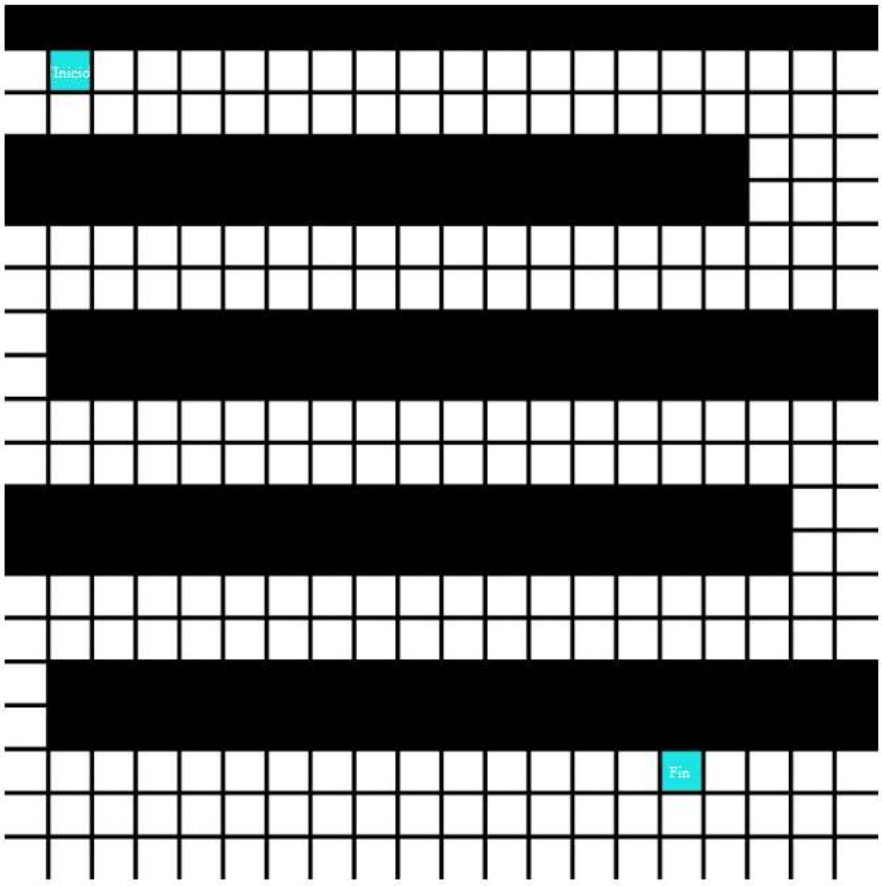

Figura 6.65 Escenario de 20x20 bloques con obstáculos

Primeramente, se muestra la trayectoria encontrada por el algoritmo A* y se muestra la solución en 2D del recorrido que realiza el NPC durante la ejecución del programa. Durante la búsqueda de trayectoria, se restringe el paso del NPC por estos obstáculos y se realiza la búsqueda en los bloques que se encuentran libres para el paso del NPC.

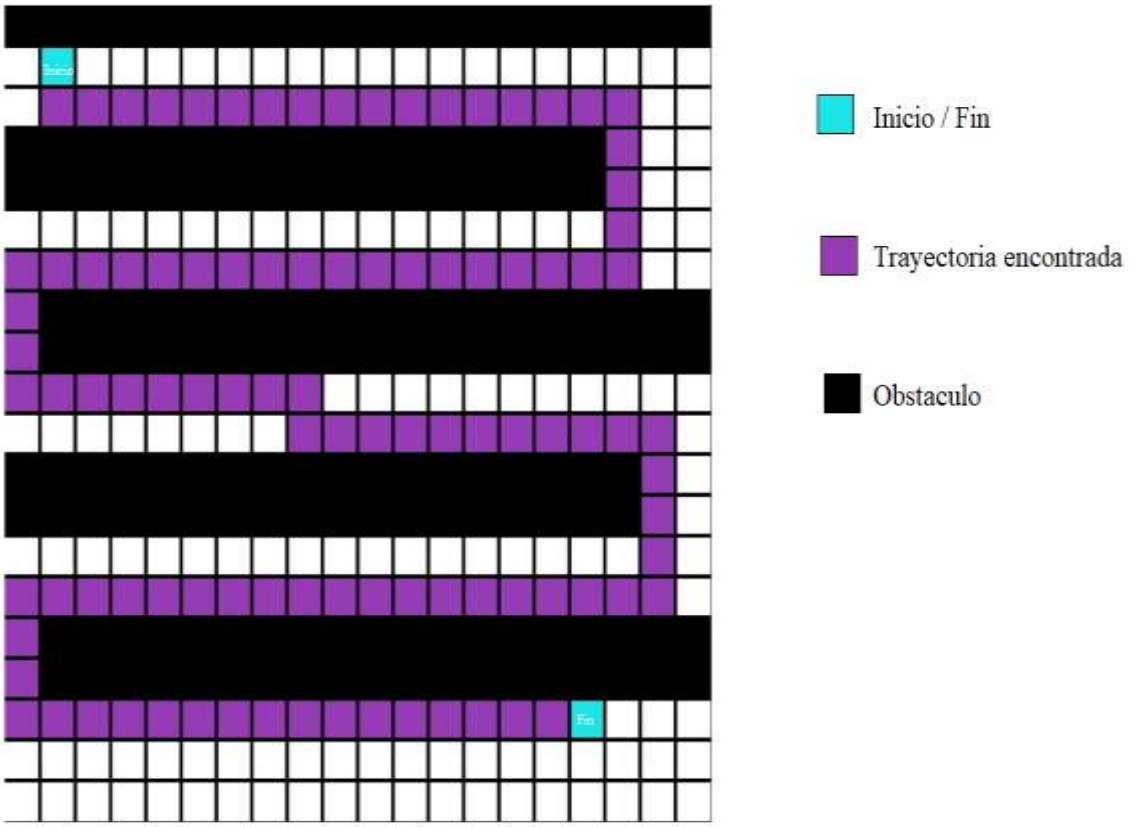

Figura 6.66 Solución de escenario de 20x20 bloques con obstáculos

La siguiente figura muestra el escenario con obstáculos, en el que se logró una tasa de 121 FPS.

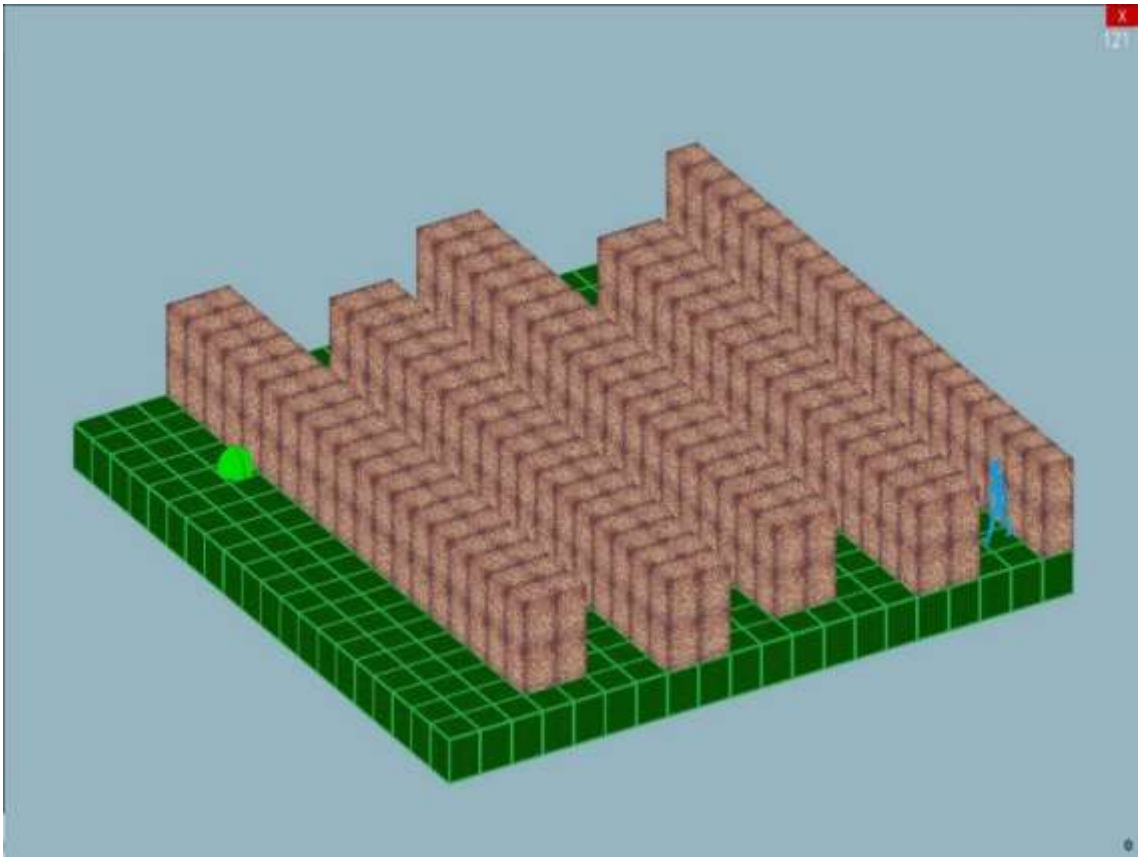

Figura 6.67 Escenario con obstáculos implementado en Ursina

Para darle una perspectiva visual diferente al escenario, se realiza la prueba de la posibilidad de mover la cámara dentro del entorno, lo que permite ver el escenario desde diferentes ángulos. Se incluye una perspectiva visual desde el campo de visión de un NPC.

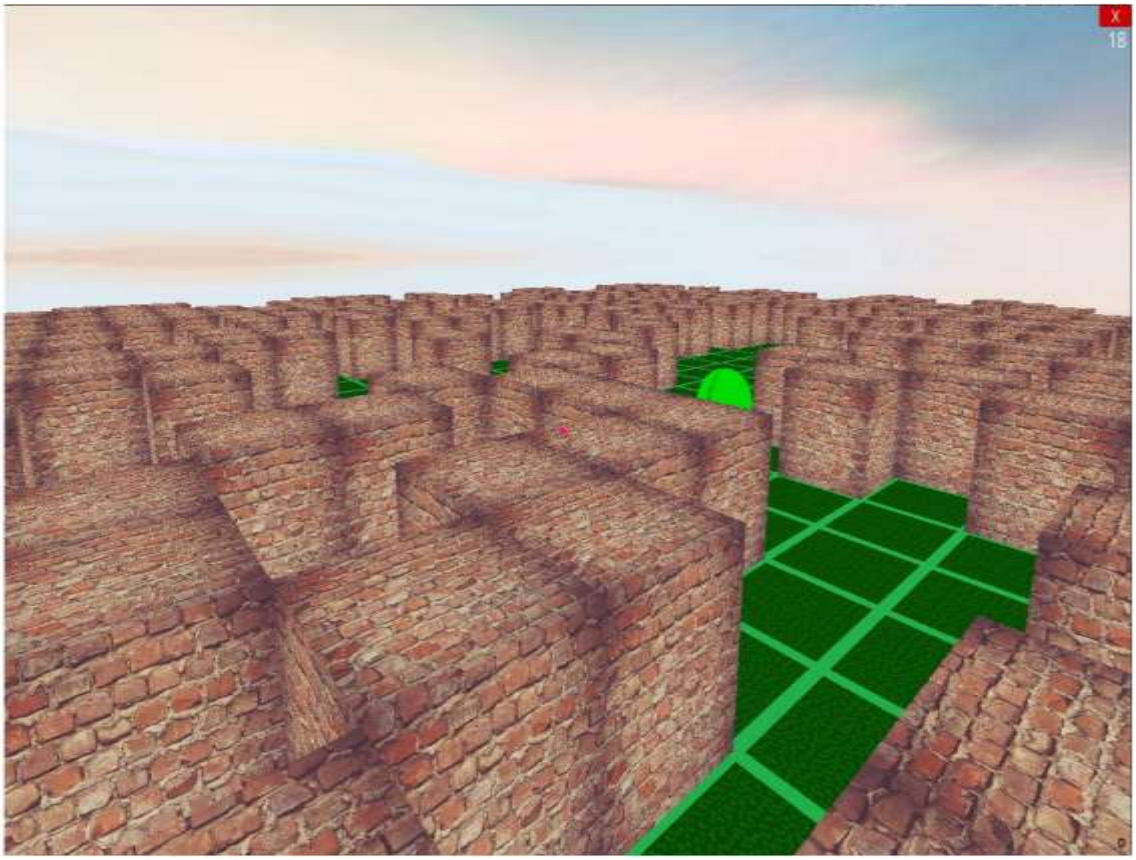

Figura 6.68 Vista de un escenario con obstáculos desde la perspectiva del NPC.

6.7 Implementación de búsqueda de trayectorias con múltiples objetivos

Una vez obtenido un resultado favorable en la generación del escenario con obstáculos, se procede a ejecutar el escenario con múltiples objetivos. En este tipo de escenario, se realiza la búsqueda de trayectoria desde la posición inicial del NPC hasta el primer objetivo definido. Una vez alcanzado el primer objetivo, se realiza la búsqueda de trayectoria hacia el segundo objetivo y así sucesivamente, hasta obtener todas las trayectorias posibles. Este tipo de escenario permite una mayor complejidad en la planificación de rutas y es útil en situaciones en las que el NPC necesita realizar varias tareas.

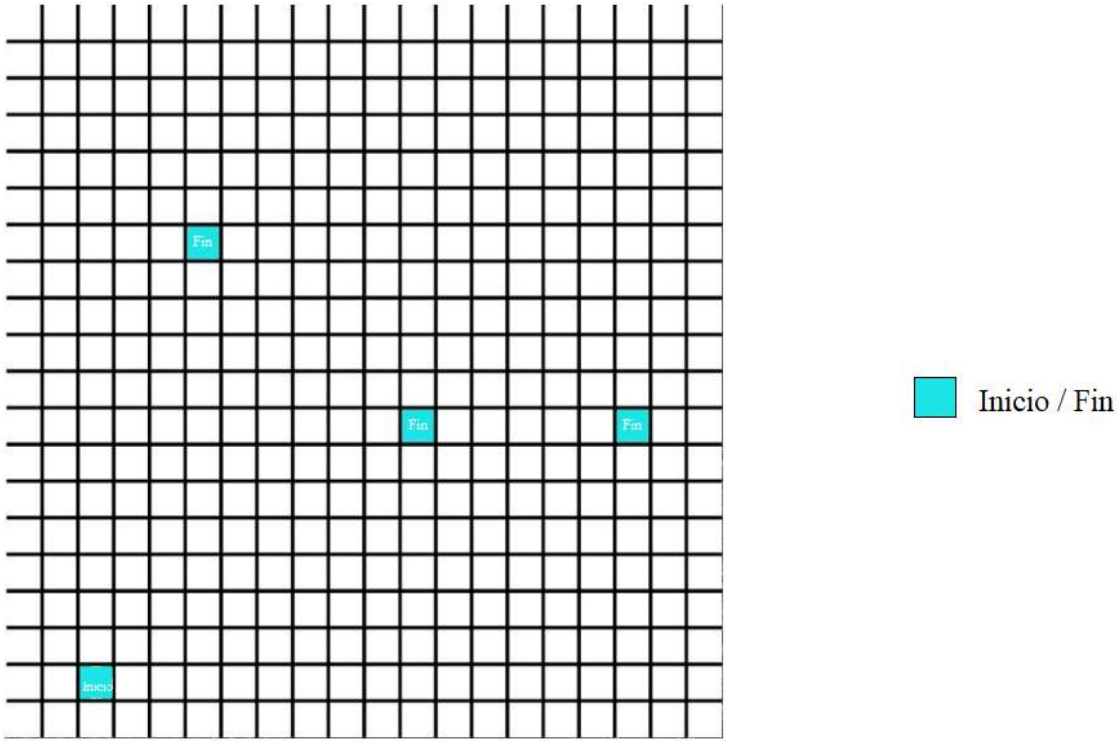

Figura 6.69 Escenario de 20x20 bloques con múltiples objetivos

Para implementar la búsqueda de trayectoria hacia múltiples objetivos, se utiliza un algoritmo similar al utilizado para la búsqueda de trayectoria hacia un solo objetivo. La diferencia radica en que, una vez encontrada la trayectoria hacia el primer objetivo, esta se almacena en un arreglo y la posición inicial de la nueva búsqueda es la posición final de la trayectoria

anterior. Este proceso se repite hasta que se hayan encontrado las trayectorias hacia todos los objetivos definidos en el escenario.

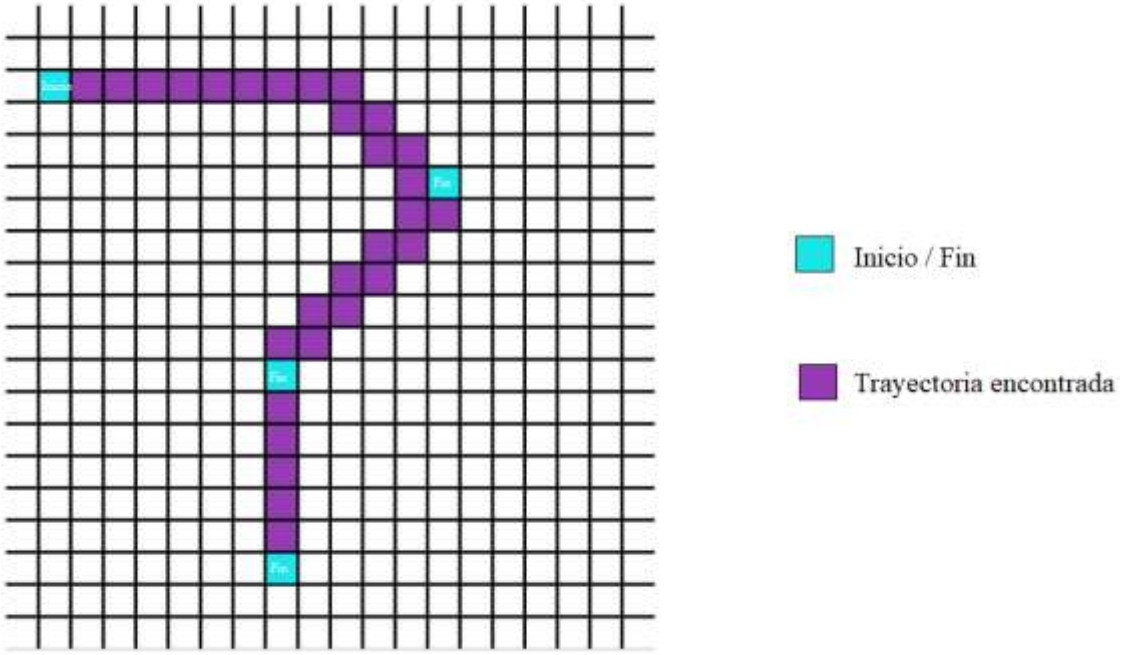

Figura 6.70 Trayectoria a realizar con múltiples objetivos

Cuando se obtiene una ejecución del escenario con múltiples objetivos exitosa, logrando mantener la fluidez visual en el mismo nivel que la implementación de un solo objetivo. En la figura 6.70 se puede observar el escenario generado a partir de la figura 6.69, que consta de veinte bloques, y se logra una visualización suave con un promedio de 114 FPS.

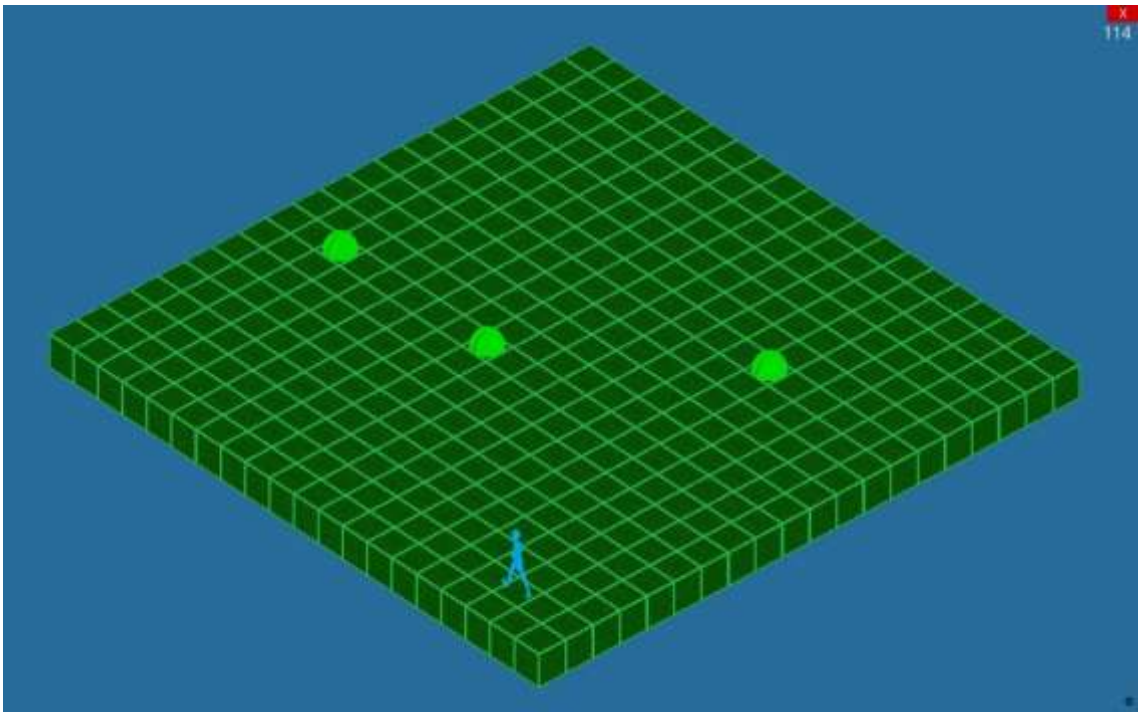

Figura 6.71 Implementación del escenario con objetivos múltiples en Ursina

6.8 Carga de Objeto como obstáculos

Continuando con la optimización de recursos en el escenario con búsqueda de trayectoria en tiempo real se detecta que sucede lo mismo que en un principio cuando se generaba cubo por cubo para el suelo. Esto nuevamente lleva a una pérdida de FPS lo cual afecta el rendimiento del escenario y el recorrido del movimiento del NPC hacia el objetivo de interés.

Por lo cual se busca la manera de crear un solo objeto que agrupara todos los obstáculos a partir del escenario que se define al principio del programa.

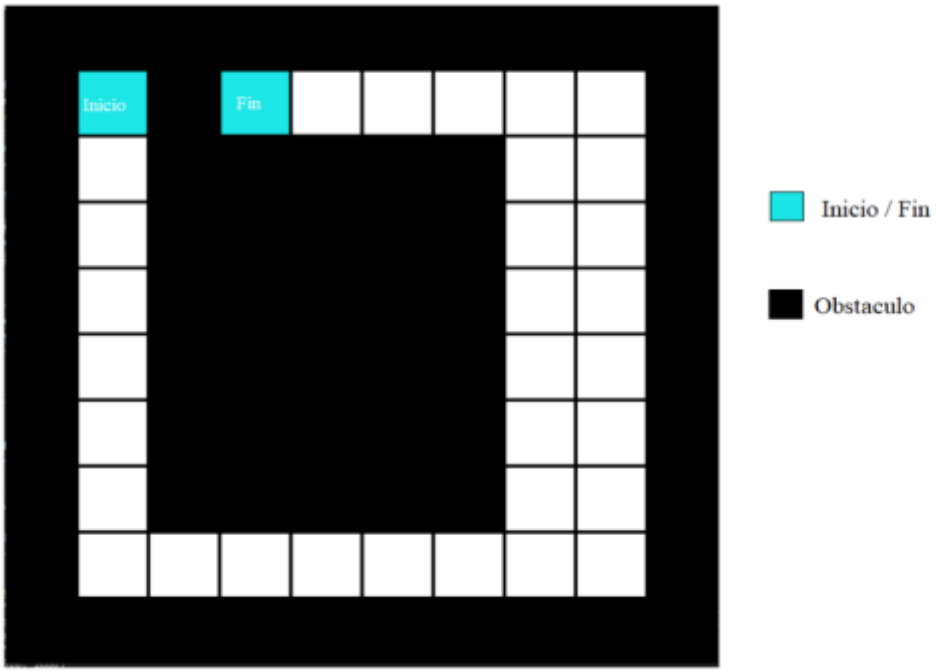

Figura 6.72 Escenario muestra

La solución es crear un solo objeto "obstáculos" que contiene todos los obstáculos del escenario, lo que permite generar los obstáculos de manera más eficiente. De esta forma, se elimina la necesidad de crear y destruir múltiples objetos en tiempo real y se mejora significativamente el rendimiento del programa. Además, esto permite una mejor organización y mantenimiento del código, ya que se reduce el número de elementos en la escena y facilita la implementación de mejoras en el futuro.

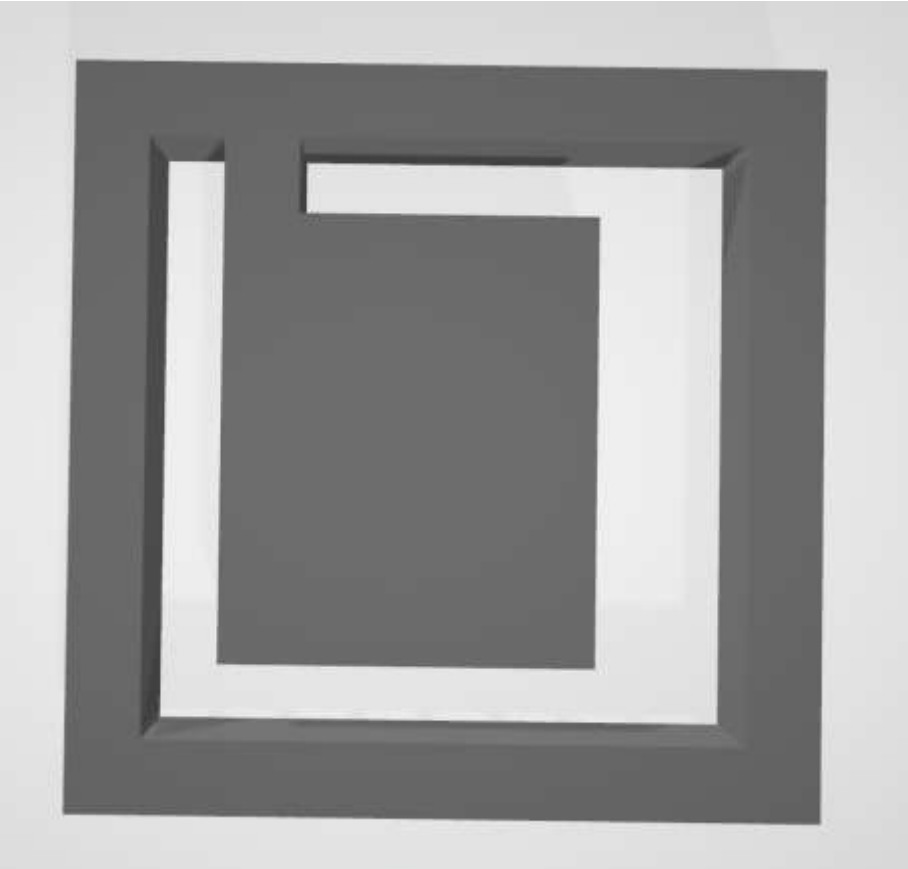

Figura 6.73 Objeto creado a partir del escenario muestra

En este caso, se generan cubos a partir de una matriz definida al inicio del programa y se unen en un solo objeto. Luego, se guarda este objeto en un archivo con extensión OBJ para que pueda ser cargado en el escenario en otro momento.

Figura 6.74 Escenario con cubos

Durante la optimización de recursos en el escenario de búsqueda de trayectoria en tiempo real, se realizó una comparación de los FPS entre dos enfoques. El primer enfoque consiste en crear el escenario cubo por cubo, lo cual genera una pérdida de FPS a medida que el NPC se movía, llegando a solo 8 FPS en escenarios complejos. En contraste, al cargar un solo objeto que agrupe todos los obstáculos definidos al inicio del programa, se logra obtener 157 FPS. Este resultado evidencia que la creación de un objeto único para los obstáculos del escenario es una solución a la reducción de FPS.

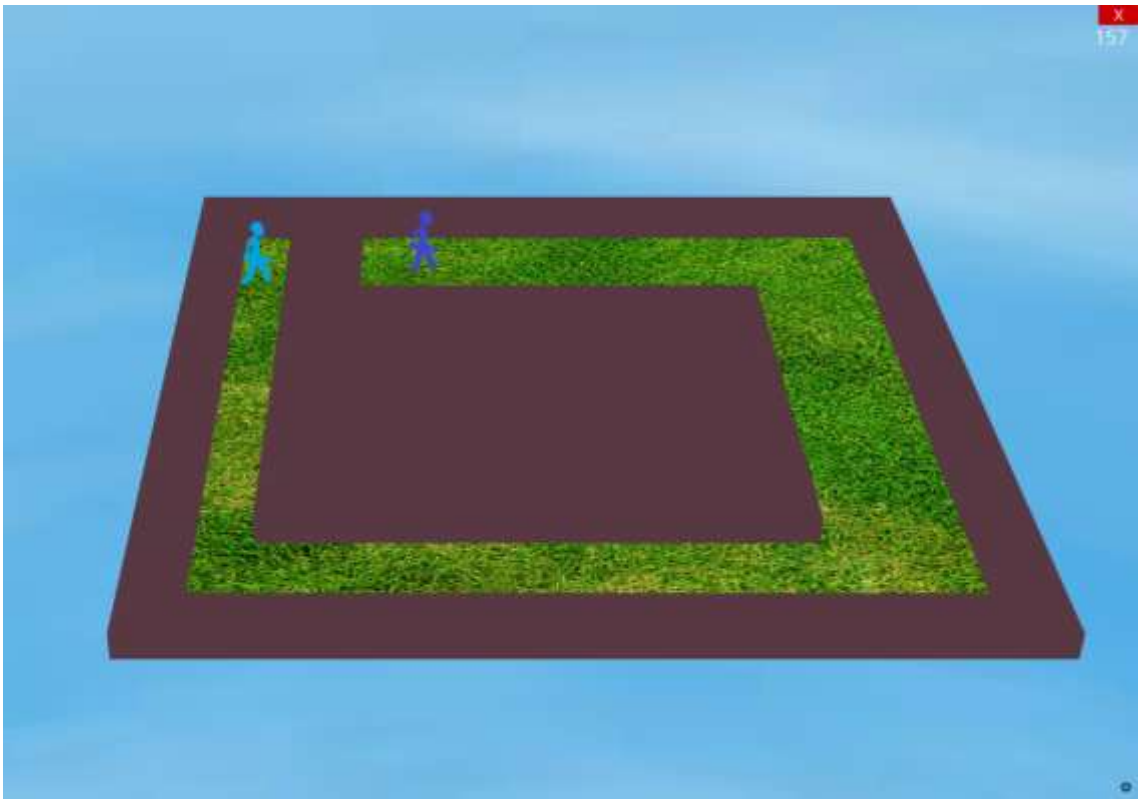

Figura 6.75 Escenario muestra cargado con un solo Objeto

Esto demuestra la eficacia de la optimización realizada en la generación y carga de los obstáculos del escenario en tiempo real. La carga de un solo objeto con todos los obstáculos definidos desde el inicio del programa evita la generación de cubos individuales y la consecuente pérdida de recursos computacionales.

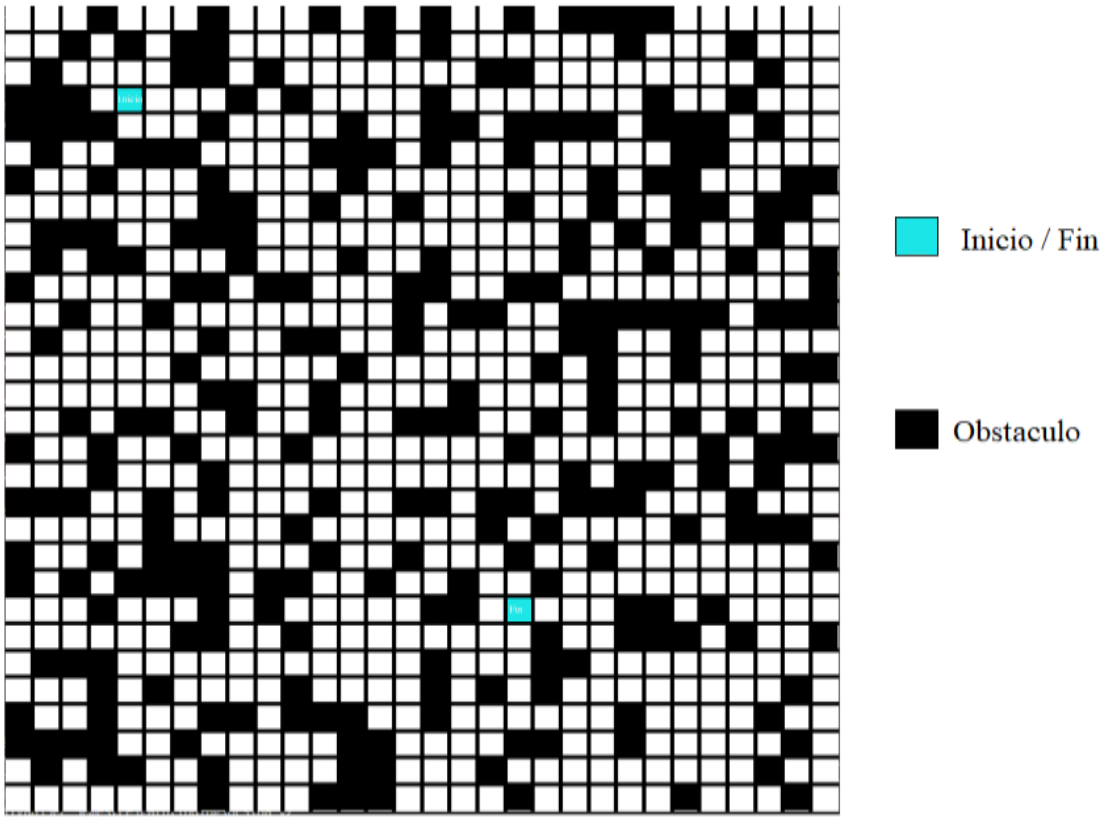

Figura 6.76 Escenario con obstáculos generados aleatoriamente

Además, se puede observar que la creación de obstáculos de forma aleatoria no afecta el rendimiento del programa, ya que la carga del objeto no depende de la complejidad o cantidad de obstáculos, sino de la estructura del objeto mismo.

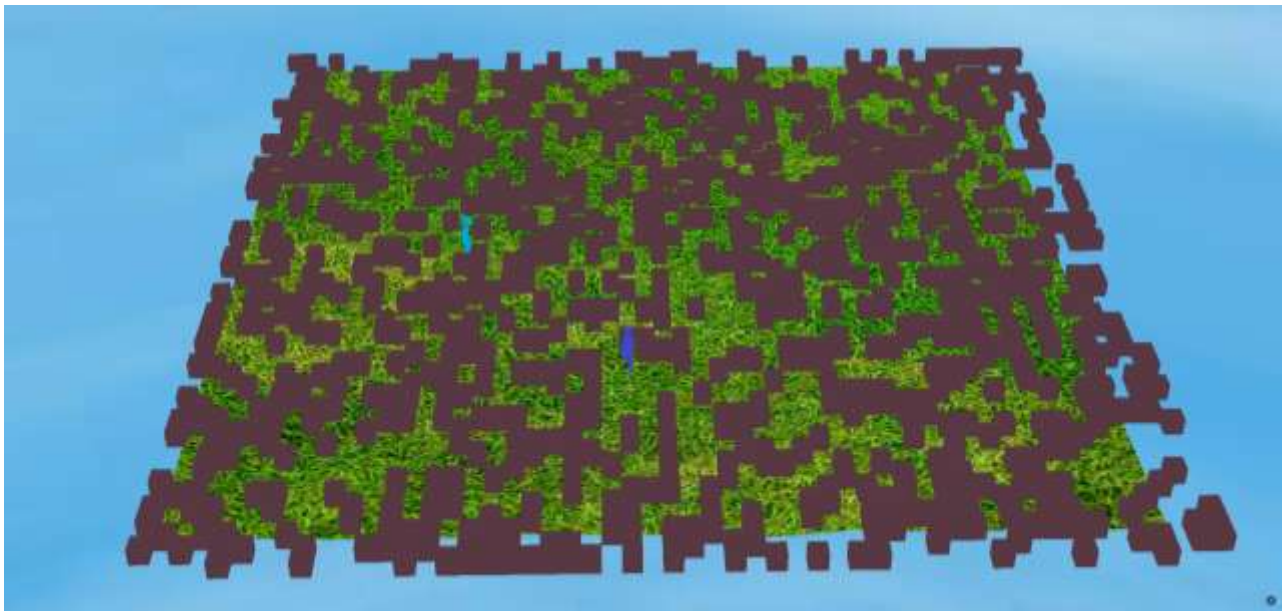

Figura 6.77 Escenario con obstáculos con un solo objeto

6.9 Implementación de escenario con búsqueda de trayectoria en tiempo real

El algoritmo A* en tiempo real requiere que la búsqueda de trayectorias se realice de manera constante y en tiempo real, lo cual se logra mediante la carga de una imagen que sirve como suelo y la colocación manual del NPC y el objetivo en diferentes posiciones para demostrar la búsqueda de trayectorias en tiempo real.

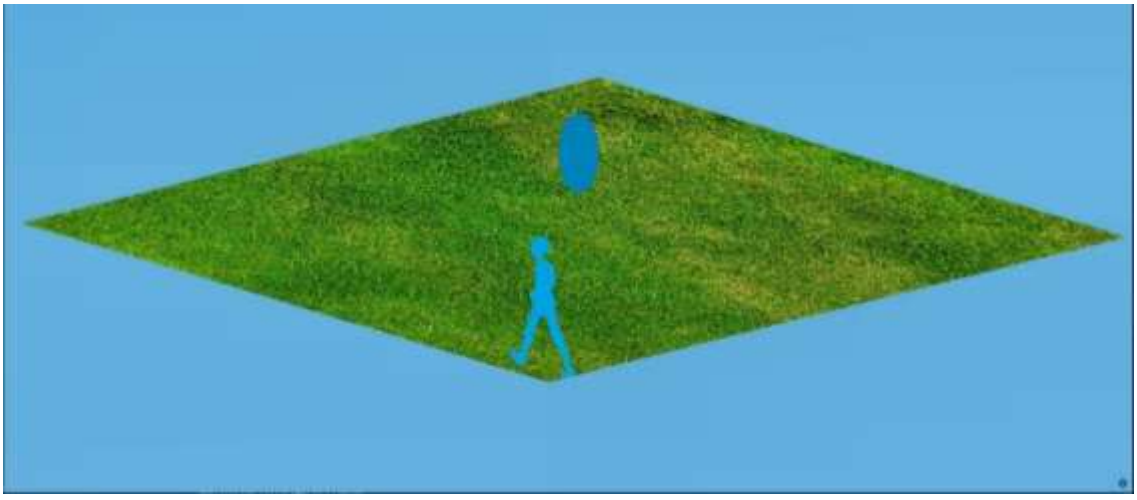

Figura 6.78 Implementación de escenario de búsqueda de trayectoria en tiempo real

En la implementación del algoritmo A* en tiempo real, se comienza con un escenario de malla de 10x10, como en los escenarios anteriores. En este caso, se utilizó una imagen precargada como suelo en la que se ubicaron el NPC y el objetivo. La figura 6.79 muestra cómo se ve el escenario en la pantalla.

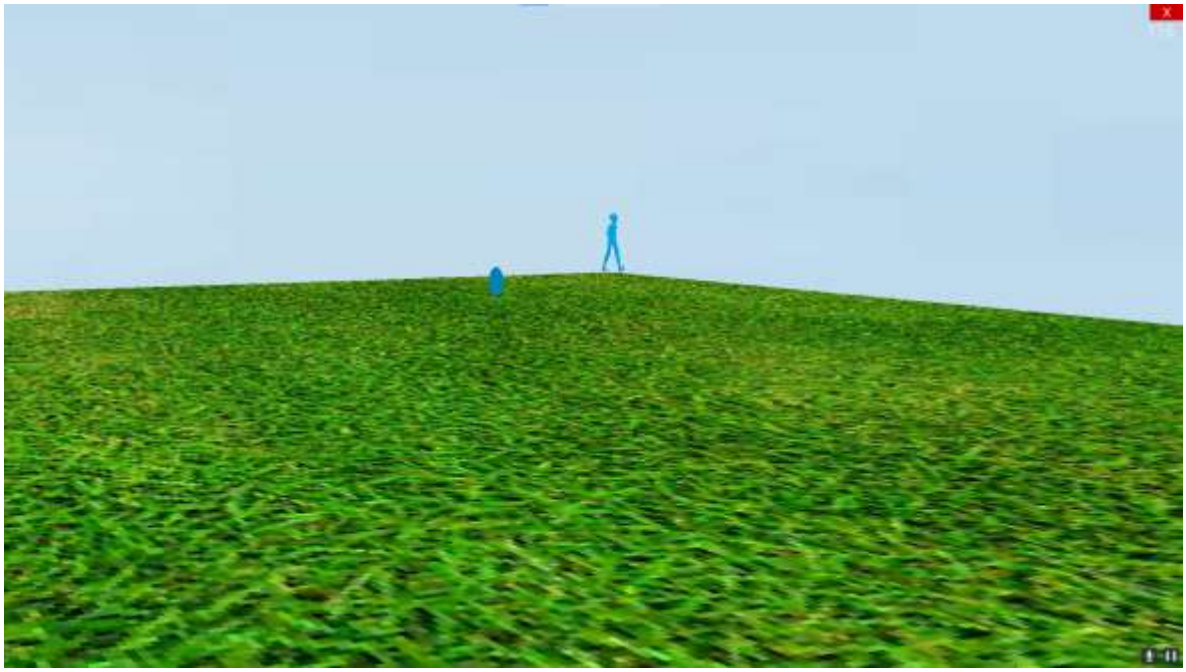

Figura 6.79 Vista de un escenario de 30x30 con imagen de suelo precargada

El algoritmo D* es una variante de A* diseñado específicamente para la búsqueda en tiempo real. Se emplea para buscar trayectorias en tiempo real y opera de manera similar a A*, con la distinción de que, mientras el objetivo permanezca dentro de un rango predefinido, no se llevará a cabo una nueva búsqueda.

Esto se ilustra en la figura 6.80, donde se presenta la ruta hacia el objetivo y el alcance de la búsqueda cuando el objetivo no ha experimentado cambios de posición.

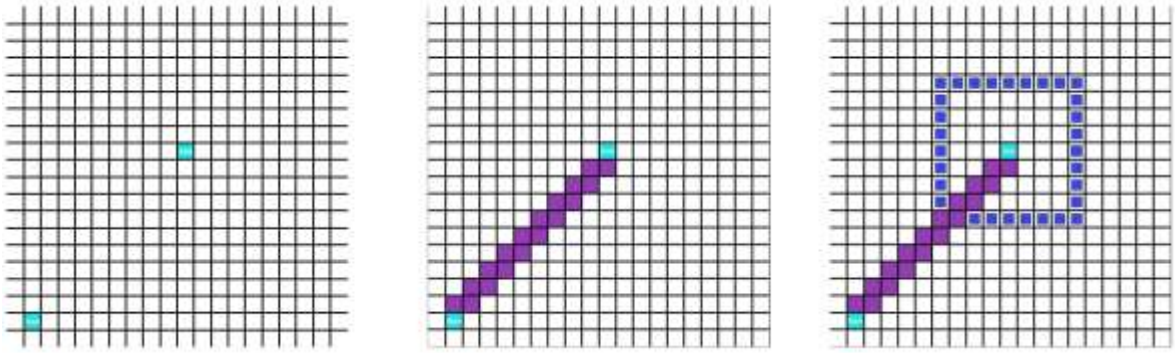

Figura 6.80 Algoritmo A búsqueda sin cambios de trayectoria*

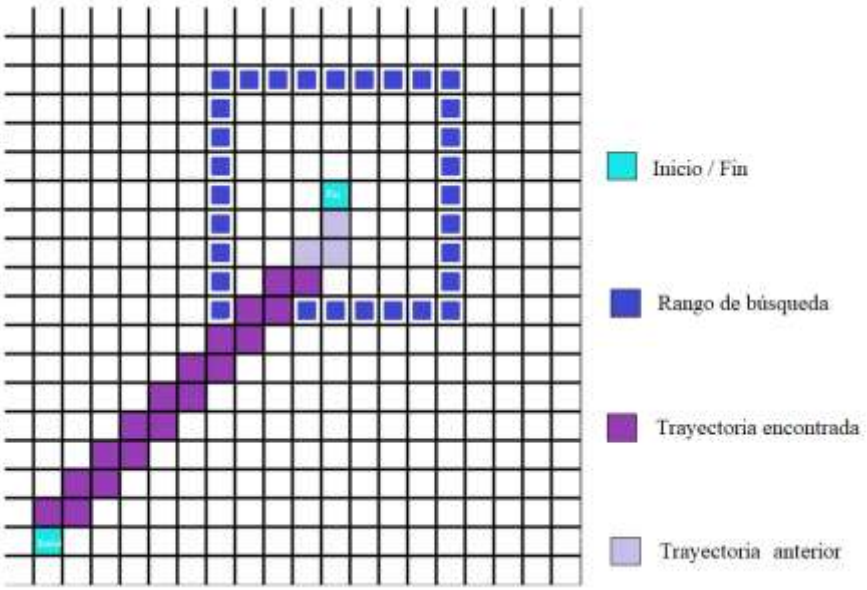

Figura 6.81 Nueva búsqueda de trayectoria mientras el punto objetivo aún se encuentre en rango.

El NPC se moverá hacia la nueva posición del punto objetivo, teniendo en cuenta el rango de bloques definido anteriormente. Si el punto objetivo se desplaza fuera de este rango, se iniciará nuevamente una búsqueda de trayectoria para adaptarse al nuevo entorno. Esta estrategia se muestra en la figura 6.81.

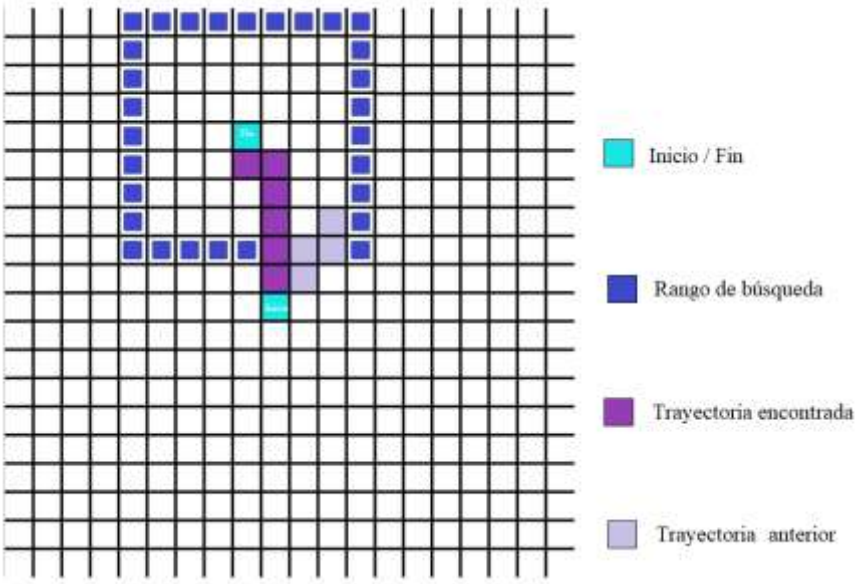

Figura 6.82 Nueva trayectoria cuando el punto objetivo ha salido del rango limite.

6.10 Velocidad del NPC

Durante la implementación del algoritmo de búsqueda en tiempo real, se observa que el desplazamiento del NPC en el escenario se realiza en un cuadro por movimiento, lo que ocasiona que el NPC se desplace a una velocidad relativamente alta. Para solucionar este problema, se decide disminuir el desplazamiento del NPC a 0.25 cuadros por movimiento, por lo que se observa un movimiento más suave del NPC. Posteriormente, con la implementación del escenario que permite la carga de una imagen como suelo, se llevan a cabo diferentes experimentos para modificar el tamaño del NPC y del punto de interés, ajustando así la velocidad del NPC y logrando un movimiento más rápido o lento según las necesidades de las pruebas.

7 Conclusiones

En la primera etapa de esta investigación, se implementan diferentes algoritmos, DFS, BFS, Dijkstra y A*, en escenarios bidimensionales. De los resultados obtenidos, se concluye que el algoritmo A* es el más eficiente en términos de tiempo de ejecución para llegar a un punto de interés, independientemente del tamaño del escenario. Y se elige implementar el algoritmo A* para buscar trayectorias en un entorno bidimensional con elementos gráficos tridimensionales con la librería Ursina, utilizando diferentes escenarios y puntos de interés para medir el desempeño de los algoritmos de búsqueda.

Los algoritmos BFS y Dijkstra tienen un consumo de recursos similar, pero el algoritmo de Dijkstra consume más CPU que el algoritmo BFS.

Durante la implementación del escenario con la librería Ursina, se detecta que el consumo de recursos no solo depende del algoritmo de búsqueda, sino también de los objetos que se cargan en el escenario. Se buscan soluciones alternativas para ahorrar recursos, como la generación de escenarios a partir de una imagen precargada que sirva como suelo.

Los objetivos específicos de la investigación se cumplen satisfactoriamente, ya que se identifica e implementa un algoritmo que optimiza el consumo de recursos en la búsqueda de trayectorias en los escenarios propuestos. Además, se evalúa el algoritmo en un entorno bidimensional con elementos gráficos tridimensionales y se verifica la escalabilidad de los escenarios considerando factores como el tamaño del entorno, la presencia de obstáculos y la interacción con personajes autónomos.

7.1 Trabajo futuro

Como trabajo futuro, se pretende llevar a cabo en el escenario la carga de imágenes con diferentes texturas que actúen como obstáculos. Para lograr esto, será necesario aplicar reglas que emulen la física del escenario, por ejemplo, una regla que simule la gravedad. Esto permitirá crear escenarios más realistas y desafiantes para el NPC en su búsqueda de trayectorias en tiempo real. Además, se espera explorar la posibilidad de incorporar otros algoritmos de búsqueda, para comparar su desempeño con el algoritmo D* utilizado actualmente. También se buscará la implementación de una interfaz gráfica de usuario que permita al usuario seleccionar los parámetros del escenario y los algoritmos de búsqueda a utilizar. Todo esto con el objetivo de seguir mejorando y ampliando las capacidades del sistema de búsqueda de trayectorias en tiempo real.

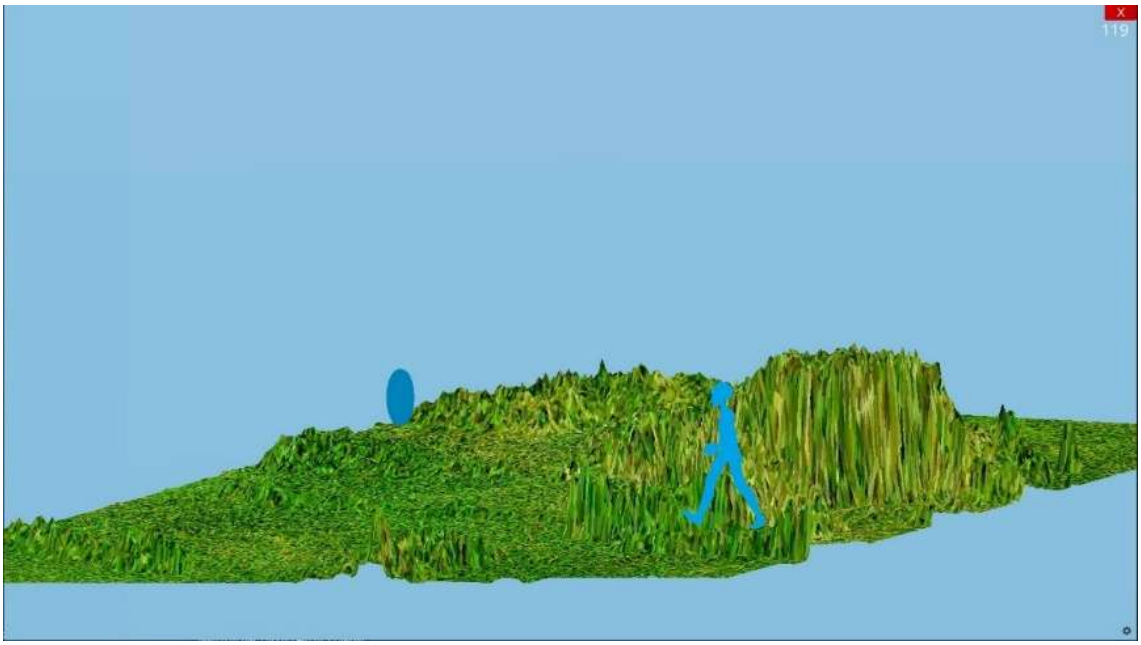

Figura 7.1 Escenario con obstáculos con imagen precargada

Además de la importación de objetos en formato OBJ, se tiene planeado incluir otros NPC para la creación de un entorno más dinámica. Esto se puede apreciar en la figura 7.1, donde se muestra un ejemplo de una imagen precargada con obstáculos ya incorporados.

Figura 7.2 Diferentes tipos de objetos que se pueden importar

Se está trabajando en la creación de diferentes tipos de escenarios o terrenos que tengan un efecto en el desplazamiento del NPC durante la búsqueda de la trayectoria o su movimiento hacia el punto objetivo. Estos escenarios podrán incluir terrenos irregulares, resbaladizos, con obstáculos naturales y otros factores que puedan influir en el movimiento del NPC.
Esto aumenta las opciones de investigación y permite el desarrollo de aplicaciones prácticas en diversos campos.

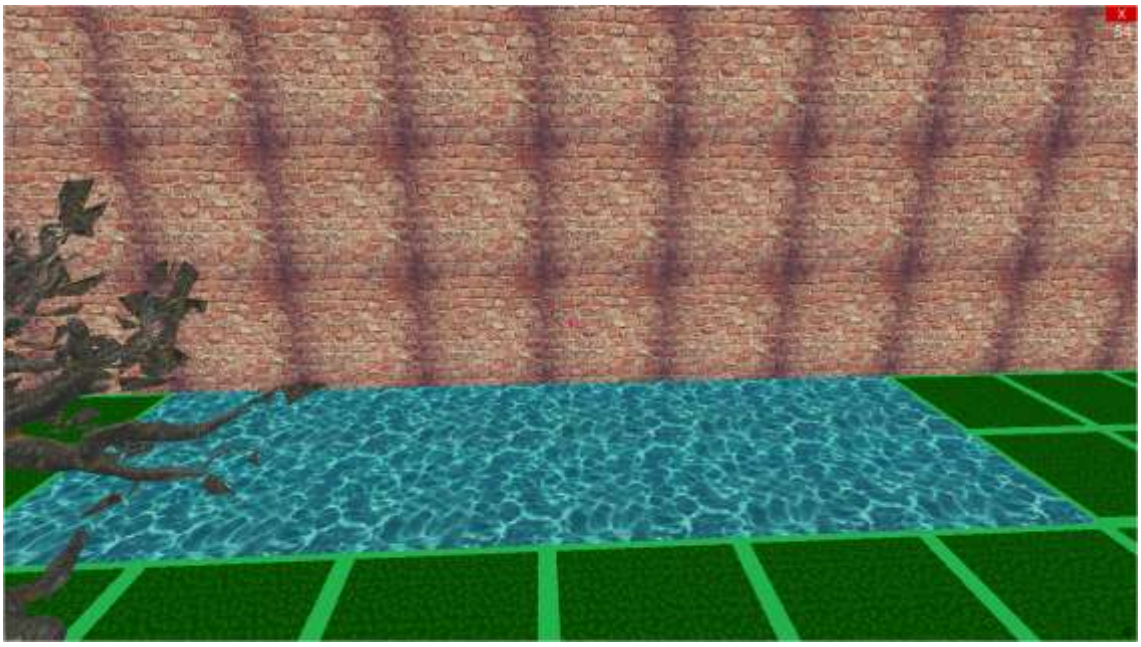

Figura 7.3 Diferentes terrenos en el escenario.

Se proyecta extender el desarrollo a un entorno tridimensional completo, integrando el eje Z, lo cual permitirá modelar escenarios más avanzados y detallados. Actualmente, la investigación se realiza en un ambiente bidimensional con elementos gráficos tridimensionales, lo cual limita a la manipulación de dos dimensiones, utilizando únicamente los ejes X e Y. Al incorporar el eje Z, se habilitará la posibilidad de explorar el rendimiento de los algoritmos en situaciones que involucren la altura o profundidad, como en aplicaciones de navegación en espacios 3D, robótica, o simulaciones físicas avanzadas. Este paso permitirá abordar desafíos que no pueden ser representados en un plano bidimensional, ofreciendo una visión más integral y precisa del comportamiento de los algoritmos en ambientes tridimensionales.

Referencias Bibliográficas

[1] X. Cui & H. Shi. "Direction oriented pathfinding in video games", International Journal of Artificial Intelligence & Applications (IJAIA), Vol.2, No.4, October 2011.

[2] A. Rafiq, T. A. Abdul Kadir, & S. Normaziah Ihsan (2020). "Pathfinding Algorithms in Game Development". IOP Conference Series: Materials Science and Engineering. https://doi.org/10.1088/1757-899X/769/1/01202.

[3] R. Graham, H. McCabe & S. Sheridan (2003). "Pathfinding in Computer Games". The ITB Journal. Volume:4. Issue: 2 Article: 6. https://doi.org/10.21427/D7ZQ9J.

[4] Azyan Yusra Kapi, Mohd Shahrizal Sunar & Muhamad Najib Zamri (2020). "A Review on Informed Search Algorithms for Video Games Pathfinding". International Journal of Advanced Trends in Computer Science and Engineering. Disponible: https://doi.org/10.30534/ijatcse/2020/42932020.

[5] R. Leigh, S. J. Louis & C. Miles (2007). "Using a Genetic Algorithm to Explore A*-like Pathfinding Algorithms". Proceedings of the 2007 IEEE Symposium on Computational Intelligence and Games (CIG 2007). https://doi.org/10.1109/CIG.2007.368081.

[6] A. Fernandes da V. Machado et al (2011). "Real Time Pathfinding with Genetic Algorithm". 2011 Brazilian Symposium on Games and Digital Entertainment Year: 2011, Volume: 1, Pages: 215-221. https://doi.ieeecomputersociety.org/10.1109/SBGAMES.2011.23

[7] U. O. Santos, A. F. V. Machado & E. W. G. Clua (2012). "Pathfinding Based on Pattern Detection Using Genetic Algorithms". SBC - Proceedings of SBGames 2012 https://sbgames.org/sbgames2012/proceedings/papers/computacao/comp-full_09.pdf.

[8] G. Recio, E. Martin, C. Estebanez, and Y. Saez, "AntBot: Ant colonies for video games" IEEE Trans. Comput. Intell. AI Games, vol. 4, no. 4, pp. 295–308, 2012. https://doi.org/10.1109/TCIAIG.2012.2212194.

[9] M. Zikky (2016). "Review of A* (A Star) Navigation Mesh Pathfinding as the Alternative of Artificial Intelligent for Ghosts Agent on the Pacman Game". EMITTER International Journal of Engineering Technology Vol. 4, No. 1, June 2016. ISSN: 2443-1168. https://doi.org/10.24003/emitter.v4i1.117.

[10] A. Primanita, R. Effendi & W. Hidayat (2017)."Comparison of A* and Iterative Deepening A* Algorithms for Non-Player Character in Role Playing Game". International Conference on Electrical Engineering and Computer Science (ICECOS) 2017. https://doi.org/10.1109/ICECOS.2017.8167134.

[11] A. N. Sabri, N. H. Radzi & A. Samah (2018). "A study on Bee Algorithm and A* Algorithm for Pathfinding in Games". ISCAIE 2018 - 2018 IEEE Symp. Comput. Appl. Ind. Electron., pp. 224–229, 2018. https://doi.org/10.1109/ISCAIE.2018.8405474

[12] Y. Sazaki & A. Primanita (2017). "Pathfinding car racing game using dynamic pathfinding algorithm and algorithm A*". 2017 3rd International Conference

on Wireless and Telematics (ICWT). https://doi.org/10.1109/ICWT.2017.8284160

[13] C. Esteves. (2015, Sep 19). Geometría Computacional [Online] Disponible: https://www.cimat.mx/~cesteves/cursos/geometria/pdf/02_ConvexHull2D.pdf

[14] M. de Berg et al. "Line Segment Intersection" in Computational Geometry Algorithms and Applications. Springer-Verlag: Berlin Heidelberg, 2008, pp 19- 43.

[15] J. Erickson (2019). Basic Graph Algorithms. 978-1-792-64483-2

[16] H. Aly-Abbass et al. (2002). Data Mining A Heuristic Approach. Idea Group Publishing. ISBN 1-930708-25-4

[17] G. Laakmann-McDowell (2016). Cracking the coding interview, sixth edition. 978-0-9847828-5-7 (ISBN 13)

[18] Intelligent Systems Group. Faculty of Informatics. University of the Basque Country (2004). "ALGORITMOS GENETICOS" [Online]. Disponible: http://www.sc.ehu.es/ccwbayes/docencia/mmcc/docs/temageneticos.pdf.

[19] Conogasi et al. (2018). Algoritmos genéticos [Online]. Disponible: http://conogasi.org/articulos/algoritmos-geneticos/.

[20] Parpinelli, Rafael & Lopes, Heitor & Freitas, Alex. (2001). "An Ant Colony Algorithm for Classification Rule Discovery". 6. 10.4018/978-1-930708-25-9.ch010.

[21] Javaid, Muhammad Adeel, Understanding Dijkstra's Algorithm (April 10, 2013). Available at SSRN: https://ssrn.com/abstract=2340905 or http://dx.doi.org/10.2139/ssrn.2340905

[22] Handy Permana, Silvester Dian, et al. Comparative Analysis of Pathfinding Algorithms A *, Dijkstra, and BFS on Maze Runner Game (2018). Available at: (PDF) Comparative Analysis of Pathfinding Algorithms A *, Dijkstra, and BFS on Maze Runner Game (researchgate.net)

[23] Weibel, C., Chirila, D. V., et al. (2020). Comparing Path-Finding Algorithms. Available at BP2_Group_1_Pathfinding__new___4_.pdf (ruc.dk)

[24] Rodola, G. (n.d.). *psutil documentation*. Retrieved from https://psutil.readthedocs.io/

[25] Python Software Foundation. (n.d.). *tracemalloc — Trace memory allocations*. Retrieved from https://docs.python.org/3/library/tracemalloc.html

[26] Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). Introduction to Algorithms (3rd ed.). MIT Press.

Anexos

Anexo 1– Creación del objeto cúbico.

```
def __init__(self, position = (0,0,0), model = 'cube' ):
    super().__init__(
        parent = scene,
        position = position,
        model = model,
        origin_y = 0.5,
        texture = textre,
        _always_on_top = True,
        eternal = False,
        color = color.white,
        meta = False
    )
```

Anexo 2 – Definición de escenario en Ursina

```
def stage(maze, start, end):
    global size_field
    size_field = len(maze)
    Sky(texture='sky')
    window.fullscreen = True
    camera.orthographic = True
    EditorCamera()
    obstacles(maze,start, end)
    run = False
```

Anexo 3 – Creación de personaje NPC en ursina.

```
class Bot(FrameAnimation3d):
    def __init__(self, x, y, z, rotate ):
        super().__init__(
            fps = 10,
            frame_times = 10,
            autoplay = False,
            loop = True,
            position=(x,y,z),
            rotation=(0, rotate, 0),
            parent=scene,
            scale= 5,
            texture = 'b1',
```

```
        _always_on_top = True,
        speed = 1,
        collider = "box"
    )
```

Anexo 4 - Carga de imagen cómo terreno en Ursina

```
terreno =  Terrain(heightmap='1g',skip=4,)
terrain = Entity(model = terreno,
    scale=(size_t,1,size_t),
    texture = "cesped",
    position = (size_t/2, -1,size_t/2),
    collider = "mesh",
    scale = size_t,
    scale_y = 50
)
```

Anexo 5 – Algoritmo de Análisis

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
# Primero, normalizamos los datos para que las métricas tengan el mismo rango
df = pd.read_csv("Directorio\Folder")
df_normalized = df.copy()

# Normalizar cada columna relevante (CPU, Memoria, Tiempo, Numero de bloques
computados)
columns_to_normalize = ['CPU', 'Memoria', 'Tiempo', 'Numero de bloques computados']
df_normalized[columns_to_normalize] = df[columns_to_normalize].apply(lambda x: (x -
x.min()) / (x.max() - x.min()))

# Si la columna "Ruta encontrada" es un valor booleano o binario (0 o 1), no es necesario
normalizarla
# Suponiendo que 1 es positivo (se encontró la ruta) y 0 es negativo, sumaremos la columna
directamente

# Asignar un peso a cada métrica según su importancia
weights = {
   'CPU': 0.2,  #20% de importancia
   'Memoria': 0.2,
   'Tiempo': 0.20,
   'Numero de bloques computados': 0.2,
   'Ruta encontrada': 0.2
}

# Calcular la puntuación ponderada para cada algoritmo
df_normalized['Puntuacion'] = (
   df_normalized['CPU'] * weights['CPU'] +
   df_normalized['Memoria'] * weights['Memoria'] +
   df_normalized['Tiempo'] * weights['Tiempo'] +
   df_normalized['Numero de bloques computados'] * weights['Numero de bloques
computados'] +
   df['Ruta encontrada'] * weights['Ruta encontrada']
)

# Agrupar por algoritmo y calcular la puntuación media
df_resultado = df_normalized.groupby('Algoritmo')['Puntuacion'].mean().reset_index()

# Ordenar para encontrar el mejor algoritmo
```

```
df_resultado = df_resultado.sort_values(by='Puntuacion', ascending=True)  # Asumiendo que menor puntuación es mejor
print(df_resultado)
# Mostrar el mejor algoritmo
mejor_algoritmo = df_resultado.iloc[0]
print(f"El mejor algoritmo es: {mejor_algoritmo['Algoritmo']} con una puntuación de {mejor_algoritmo['Puntuacion']:.4f}")

# Ordenar el DataFrame para visualizar mejor
df_resultado = df_resultado.sort_values(by='Puntuacion', ascending=True)  # Menor puntuación es mejor

# Crear un gráfico de barras
plt.figure(figsize=(10, 6))
plt.barh(df_resultado['Algoritmo'], df_resultado['Puntuacion'], color='skyblue')

# Añadir etiquetas y título
plt.xlabel('Puntuación Ponderada')
plt.ylabel('Algoritmo')
plt.title('Comparación de Algoritmos según Puntuación Ponderada')

# Añadir valor de la puntuación en las barras
for index, value in enumerate(df_resultado['Puntuacion']):
    plt.text(value, index, f'{value:.4f}', va='center')

path_graf =  ("Directorio\Folder\images\\")
# Mostrar el gráfico
plt.savefig(path_graf+"full.png")
# Mostrar el gráfico
#plt.show()
df_resultado
# Ordenar el DataFrame para visualizar mejor
df_resultado = df_resultado.sort_values(by='Puntuacion', ascending=True)  # Menor puntuación es mejor

#Se elimina el algoritmo DFS del dataframe para una mejor visualización
df_resultado = df_resultado[~df_resultado['Algoritmo'].isin(['DFS'])]
# Crear un gráfico de barras
plt.figure(figsize=(10, 6))
plt.barh(df_resultado['Algoritmo'], df_resultado['Puntuacion'], color='skyblue')

# Añadir etiquetas y título
plt.xlabel('Puntuación Ponderada')
plt.ylabel('Algoritmo')
plt.title('Comparación de Algoritmos según Puntuación Ponderada')
```

```
# Añadir valor de la puntuación en las barras
for index, value in enumerate(df_resultado['Puntuacion']):
    plt.text(value, index, f'{value:.4f}', va='center')

# Mostrar el gráfico
plt.savefig(path_graf+"sin_DFS.png")
# Mostrar el gráfico
plt.show()
```