Implementación de un algoritmo D* modificado para planificación de trayectorias

Carlos Alberto Balderrama-García, Ulises Orozco-Rosas, Kenia Picos*

CETYS Universidad, Centro de Innovación y Diseño (CEID), Av. CETYS Universidad No. 4. El Lago, C.P. 22210, Tijuana B.C., México
carlos.balderrama@cetys.edu.mx, ulises.orozco@cetys.mx, kenia.picos@cetys.mx

Resumen

En este trabajo se busca una solución eficiente al problema de planificación de trayectorias a través de la implementación de los algoritmos A* y D* basados en heurísticas. Este artículo documenta el proceso de implementación, así como los resultados de la implementación de los algoritmos en un ambiente virtual diseñado específicamente para probar algoritmos de planificación de trayectorias. Para el algoritmo D* se utiliza un método para la replanificación diferente al del algoritmo D* original, debido a la complejidad que representa su implementación. Se explican también algunas consideraciones que se deben tener en cuenta al momento de implementar los algoritmos A* y D* para conseguir resultados satisfactorios. Tras la ejecución de los algoritmos se observa que A* es el algoritmo que da resultados más rápidamente (debido a que no necesita verificar cambios en el mapa durante la ejecución). No se observan cambios drásticos en los tiempos de ejecución entre el algoritmo D* y su variante propuesta, incluso con el algoritmo de replanificación más sencillo que el original, aunque este sigue siendo más lento en tiempo de ejecución que el algoritmo A* para mapas que no cambian. Utilizar un algoritmo de replanificación más sencillo puede facilitar la implementación del programa y dar resultados efectivos donde el aumento en los tiempos de ejecución es negligible, pero los tiempos de ejecución pueden aumentar drásticamente para entornos más grandes.

Palabras clave— Algoritmo A*, Algoritmo D*, Búsqueda heurística incremental, Planificación de trayectorias, Teoría de grafos.

Abstract

In this work, an efficient solution to the path planning problem is sought through the implementation of heuristic-based path planning algorithms A and D*. This article details the implementation process as well as the results of implementing said algorithms in a virtual environment specifically designed for testing path planning algorithms. For the D* algorithm, a different replanning algorithm is used than the original implementation, and its performance is evaluated and compared with the original implementation. Some special considerations were taken into account in order to achieve satisfactory results. After the execution of the algorithms, it is observed that A* is the fastest algorithm (because it does not need to verify changes in the map during execution). No drastic changes are observed in the execution times of the D* algorithm and its modified version (even with the new replanning algorithm), although it is still slower than the original A* algorithm in maps that do not change during execution. Using a simpler replanning algorithm like the one used in this article may facilitate the implementation process and give effective results with negligible increases in execution times for small test environments, while in significantly larger test environments, the execution times can increase drastically.*

Keywords— *A* Algorithm, D* Algorithm, Incremental heuristic search, Path planning, Graph theory.*

1. INTRODUCCIÓN

Un problema recurrente en el área de las ciencias computacionales es el de encontrar trayectorias óptimas en un grafo. El implementar algoritmos eficientes para planificación de trayectorias, aunque puede parecer sencillo a simple vista, suele dar más problemas de los esperados debido a diversas causas, entre las que pueden estar grafos que están cambiando constantemente, o sencillamente que el grafo es demasiado grande para poder analizarlo en busca de trayectorias óptimas con algoritmos no muy bien planteados.

Aun sabiendo que se le puede dar una solución a los problemas de planificación de trayectorias con algoritmos sencillos como las búsquedas en amplitud (BFS, por sus siglas en inglés) y en profundidad (DFS, por sus siglas en inglés), estos algoritmos no son lo suficientemente eficientes para ser considerados soluciones efectivas. Lo anterior es el motivo por el que en muchos casos se eligen algoritmos como el de Dijkstra u otros algoritmos de búsqueda basados en la heurística, como A* y D* [1].

En este trabajo se busca una solución a este problema por medio de la implementación de los algoritmos A* y D*, así como una versión modificada del algoritmo D*, en un ambiente virtual que simule el espacio de trabajo de un robot móvil. Para esto se proponen los siguientes objetivos:

- Dicho ambiente debe tener una dimensión de 10×10 unidades cuadradas con la opción de ser discretizado con una resolución variable (1.0, 0.5, 0.25, 0.125, 0.0625), esto significa que cada unidad puede ser dividida en subunidades cuadradas cuyo lado sea igual a la resolución.
- El ambiente debe permitir que el robot móvil gire y avance en diferentes direcciones, dependiendo de si el usuario elige utilizar la vecindad de Von Neumann

* Autor para la correspondencia (*Corresponding author*).

(norte, sur, este y oeste) o la vecindad de Moore (norte, sur, este, oeste, noreste, noroeste, sureste, suroeste).

- Finalmente, el ambiente virtual debe ser capaz de leer archivos de valores separados por comas (CSV, por sus siglas en inglés) donde se especifique en cada renglón la información de los **obstáculos que deberán ser colocados en el entorno**. El robot móvil no deberá ser capaz de pasar a través de un punto del mapa que se encuentre marcado como obstáculo. Cada línea del archivo CSV deberá contener las coordenadas de la esquina inferior izquierda de un obstáculo rectangular (x e y), seguido de sus dimensiones l y w (en los ejes x e y, respectivamente).

Para poner a prueba el ambiente virtual, se deben **implementar los algoritmos de planificación de trayectorias A* y D***, en programas que permitan especificar la resolución a utilizar, el tipo de vecindad, un archivo CSV con los obstáculos a utilizar, las coordenadas de origen del robot móvil, y las coordenadas de la meta del robot móvil.

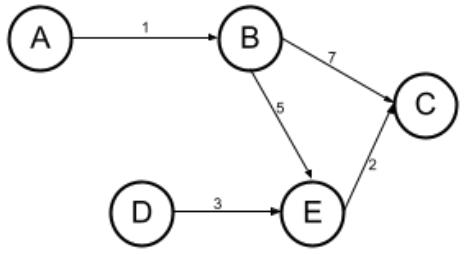
2. FUNDAMENTOS

A continuación, se presenta una descripción general de los conceptos y algoritmos utilizados en este trabajo.

2.1 Grafo

Un grafo es, en pocas palabras, una estructura de datos que se representa a través de un conjunto de vértices (nodos) unidos por aristas. Un grafo puede ser dirigido o no dirigido, así como también puede ser ponderado o no ponderado. Un grafo dirigido es un grafo en el que las aristas que conectan dos vértices son unidireccionales, es decir, que solo es posible realizar un recorrido entre ambos vértices en una sola dirección. Un grafo ponderado es un grafo en el que existe un costo para recorrer una arista que conecta dos vértices. Con el fin de ilustrar los conceptos anteriores, en la Figura 1 se puede observar un grafo ponderado dirigido.

Figura 1. Grafo dirigido ponderado.



2.2 Algoritmo de Dijkstra

El algoritmo de Dijkstra es un algoritmo voraz (*greedy*) utilizado para resolver el problema de encontrar las trayectorias óptimas con un solo nodo inicial en grafos dirigidos (aunque el algoritmo puede ser utilizado también en grafos no dirigidos) y ponderados con pesos de aristas positivos [2].

Para que el algoritmo funcione es necesario llevar un registro de todos los nodos (vértices) del grafo, donde se debe registrar un nodo antecesor y el costo del vértice entre el nodo y su antecesor (se consideran valores infinitos si el nodo no ha sido inspeccionado en ningún momento, y un valor registrado de cero para el nodo inicial). El algoritmo consiste en, comenzando en un nodo inicial, examinar cada uno de los vecinos no visitados del nodo actual e ir registrando en cada paso el costo mínimo para llegar del nodo inicial hasta cada uno de los nodos vecinos. Esto se hace al momento de examinar cada vecino: se revisa si el peso de la arista que conecta el nodo actual con su vecino sumado al valor registrado del nodo actual es menor que el costo que se tiene registrado para llegar hasta el nodo vecino. Si el costo es menor, se registra el peso de la arista sumado con el valor registrado del nodo actual como el nuevo costo mínimo del vecino, y se asigna el nodo actual como el antecesor del vecino. Esto se repite con cada uno de los vecinos del nodo actual.

Al realizarse el proceso anterior en un nodo, el nodo se marca como visitado y se elige el nodo vecino con el costo mínimo desde el nodo actual (el que tenga la arista con el menor peso) como el nuevo nodo actual, siempre y cuando este no esté marcado como un nodo visitado. El algoritmo se repite hasta que ya no haya más nodos que visitar. La trayectoria óptima se puede obtener si se examina el nodo antecesor del nodo meta, y el antecesor de este de manera sucesiva hasta llegar al nodo inicial. Se garantiza que la trayectoria dada por este método es la trayectoria menos costosa (la más óptima) entre el nodo inicial y el nodo meta en el grafo.

Debido a que no importa el orden en que se revisen los nodos, la manera más sencilla de ejecutar el algoritmo es utilizando una cola de prioridad, en donde se agregan los vecinos del nodo actual. La lista es ordenada utilizando el valor registrado de distancia mínima de cada nodo, y se eligen los que tienen las distancias más cortas primero, para agregar sus vecinos a la lista y continuar con el proceso.

La desventaja de este método es que se requiere tener un conocimiento total del grafo antes de comenzar la ejecución del algoritmo, además de que en ocasiones se realiza procesamiento innecesario de información por siempre elegir las aristas con el menor peso, a pesar de que puede haber rutas más cortas (por más contraintuitivo que parezca) si se sigue la arista con el mayor peso. También, si el grafo cambia en algún momento previo a la ejecución del algoritmo y este cambio se desconoce, el algoritmo no es capaz de reaccionar ante tal cambio y puede dar una trayectoria imposible de recorrer o que no es la más óptima para el nuevo grafo.

2.3 Algoritmo A*

El algoritmo A* (pronunciado como “a-estrella”) es una variante del algoritmo de Dijkstra, que incorpora una estimación heurística del costo de llegar al nodo meta para reducir el número de estado explorados por el algoritmo [3].

El algoritmo funciona de exactamente la misma manera que el algoritmo de Dijkstra, pues la única diferencia significativa entre ambos algoritmos es el valor utilizado para ordenar la cola de prioridad a la que se agregan los nodos vecinos [3]. La función utilizada para ordenar la cola

de prioridad está dada por un valor global y uno local, donde el valor local está dado por la suma de la distancia mínima recorrida para llegar desde el nodo inicial hasta el nodo actual y de un estimado de la distancia del nodo actual hasta el nodo meta [3] (en muchos casos se suele utilizar la distancia euclidiana entre los nodos, si se habla de un grafo que representa un mapa).

Esta estimación de la distancia entre el nodo actual y el nodo meta es la heurística que permite ordenar de una mejor manera la cola de prioridad. En otras palabras, el uso de esta función implica que la cola de prioridad está ordenada por estimaciones del costo óptimo desde el nodo inicial hasta el nodo meta [4]. La ventaja de este algoritmo es que, mientras más cerca está el nodo actual del nodo meta, menos nodos se tienden a explorar (en comparación con el algoritmo de Dijkstra). Si la estimación heurística es fiable, el algoritmo garantiza que se encontrará la trayectoria más corta entre el nodo inicial y el nodo meta. Sin embargo, es importante destacar que utilizar el algoritmo A* puede no ser la mejor solución en ocasiones, pues se complica encontrar una función que de una estimación heurística eficiente de evaluar y que sirva como una guía fiable [3].

2.4 Algoritmo D*

El algoritmo D* (pronunciado como "d-estrella") es un algoritmo de búsqueda basado en el algoritmo A*. La diferencia de este algoritmo con el algoritmo A* original es que este permite reaccionar ante cambios en los pesos de las aristas del grafo, de ahí su nombre "D*", que hace referencia a un algoritmo A* **dinámico**, en el sentido de que es capaz de reaccionar ante cambios en los pesos de las aristas mientras es ejecutado. Este algoritmo permite también obtener las trayectorias óptimas entre dos nodos del grafo [5]. También es necesario mencionar que, a diferencia de A*, D* inicia la búsqueda de la trayectoria óptima desde el nodo meta, y no desde el nodo de inicio.

El algoritmo es funcionalmente equivalente a A*, pero es más eficiente en ambientes complejos y en expansión, pues los cambios locales en el mundo no tienen un impacto significativo en la trayectoria obtenida, y evita los costos computacionales elevados de realizar un retroceso (*backtracking*).

D* realiza la planificación utilizando el algoritmo de Dijkstra, y utiliza la información ya procesada del grafo para realizar la replanificación cuando es necesario [6]. Esto último es el principal punto fuerte del algoritmo, pues permite explotar similitudes en las búsquedas para resolver problemas más rápidamente, en lugar de comenzar el proceso desde el inicio. Utiliza las partes idénticas del árbol de búsqueda previo y corrige para actualizarlo [7].

3. METODOLOGÍA

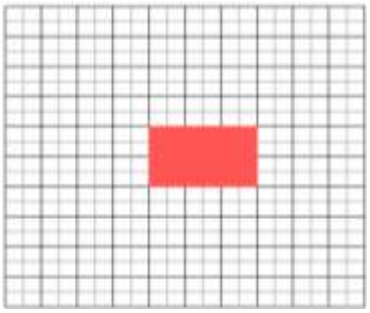
Para la implementación del programa propuesto en los objetivos de la investigación se utilizó el lenguaje de programación Python en su versión 3.7.4. A continuación, se detalla el proceso de desarrollo de los programas.

3.1 Implementación de un ambiente virtual para planificación de trayectorias

Para la implementación del ambiente virtual para el robot móvil, se escribió un módulo de Python capaz de manipular el grafo utilizado para la búsqueda, mientras se muestra en una ventana el mapa organizado como una cuadrícula.

Se utilizó el módulo *turtle* de Python para manejar la parte visual. El mapa se muestra en una cuadrícula de 10×10 unidades, cuyo tamaño en la pantalla depende del tamaño de la ventana. Para calcular el tamaño se restan cuarenta píxeles (para dejar un espacio con los bordes de la pantalla) en cada dimensión y se divide el resto entre diez. Los obstáculos se muestran en la pantalla como un rectángulo de color rojo. En la Figura 2 se puede apreciar el entorno virtual.

Figura 2. Entorno virtual mostrando un obstáculo.



Para realizar los movimientos se utiliza un vector con compensaciones (*offsets*) para poder conocer hacia qué nodo de la cuadrícula moverse. Este vector con compensaciones cambia dependiendo de la vecindad elegida. En el caso de utilizar la vecindad de Von Neumann, las compensaciones utilizadas son $(0,1)$, $(1,0)$, $(0,-1)$ y $(-1,0)$. En el caso de utilizar la vecindad de Moore, se utilizan las mismas que en Von Neumann y además se agregan: $(-1,1)$, $(1,1)$, $(1,-1)$ y $(-1,-1)$. Los vecinos se obtienen sumando dichas compensaciones a la posición del nodo actual.

El entorno cuenta, además, con la capacidad de trazar rutas en el mapa (siguiendo una línea verde con puntos en los extremos) si se le proporciona un vector conteniendo las coordenadas de cada uno de los nodos en la trayectoria.

3.2 Implementación del algoritmo A*

Dado que la manera en que el algoritmo es explicado de manera matemática, fue necesario escribir un pseudocódigo que mostrara de manera más clara el procedimiento a seguir para encontrar la trayectoria óptima de dos nodos. Siguiendo las especificaciones del algoritmo A* se llegó al pseudocódigo mostrado en el Código 1.

En el pseudocódigo se incluyen dos funciones importantes: *distance(Nodo, Nodo)* y *heur(Nodo).* La función *distancia* se utiliza para obtener el peso de la arista que une dos nodos dados. En el caso de esta implementación se utiliza la distancia euclidiana entre dos puntos en el mapa. De forma similar, la función *heur* se utiliza para obtener una estimación heurística de la distancia entre un nodo dado y el nodo meta, que en este caso también es la distancia euclidiana entre el nodo dado y la meta.

Debido a que se utilizaron coordenadas cartesianas con punto decimal para especificar la ubicación del robot móvil, fue necesario realizar una conversión entre el sistema de coordenadas y el sistema de índices para poder manipular matriz que almacena los datos del entorno. Esto fue implementado en el sistema del ambiente virtual.

Código 1. Pseudocódigo para implementación del algoritmo A*.

```
For node in nodes
  node.State = NEW
Start = nodes[0]
Start.global = heur(Start)
Start.local = 0
PriorityQueue.Add(Start)

while PriorityQueue.NotEmpty
  A = PriorityQueue.Pop()
  A.State = VISITED
  for B in A.Neighbors
    if A.local + distance(A,B) < B.local
      B.parent = A
      B.local = A.local + distance(A,B)
      B.global = B.local + heur(Start)

    if B.State = NEW
      B.State = OPEN
      PriorityQueue.Add(B)
  PriorityQueue.Sort()
```

Para el proceso de inicialización, se observa que se utilizan distancias globales locales infinitas para asegurar que el nodo sea procesado por lo menos una vez, al comparar los valores locales de los nodos para saber si es necesario actualizarlos y agregarlos a la cola de prioridad. En general, el proceso de inicialización es muy simple, como se mostró en el pseudocódigo del Código 1.

Para el proceso de búsqueda, se siguen pasos sencillos: extraer el nodo A y marcarlo, actualizar cada nodo hijo del nodo A si se forma una trayectoria más corta, y agregar a la cola de prioridad a los nodos hijos que no estén abiertos o se hayan visitado antes.

La trayectoria obtenida está dada por la secuencia que se forma al seguir el nodo padre del nodo meta, y continuar el proceso hasta llegar al nodo de inicio. De implementarse correctamente, se sabe que se obtiene la ruta óptima entre el nodo de inicio y el nodo meta.

3.3 Implementación del algoritmo D* (expansión de nodos solamente)

Para la parte de expansión de nodos del algoritmo A* se utilizó como base el pseudocódigo propuesto por Anthony Stentz [5], mostrado en el Código 2.

Fue necesario definir las funciones *c*, *MIN-STATE*, *GET-K-MIN*, *DELETE* e *INSERT* para permitir que el algoritmo funcionara de manera correcta. La función *c* es la función de costo entre dos vértices, que, como ya se mencionó en repetidas ocasiones, es la distancia euclidiana entre los nodos en el mapa (retorna un costo infinito si uno de los dos nodos proporcionados como parámetros representa un obstáculo). La función *MIN-STAT* y *GET-K-MIN* están relacionadas estrechamente. *MIN-STATE* retorna el primer elemento en la cola de prioridad, ordenando con base a un valor que será llamado *k*, mientras que *GET-K-MIN* retorna el valor *k* mínimo en la cola de prioridad, es decir, el del elemento retornado por *MIN-STATE*.

Código 2. Pseudocódigo para la implementación de la parte de expansión del algoritmo D*.

```
def PROCESS-STATE()
  X = MIN-STATE()
  if X = NULL then return -1
  k_old = GET-KMIN(); DELETE(X)
  if k_old < h(X) then
    for each neighbor Y of X:
      if h(Y) <= k_old and h(X) > h(Y) + c(Y, X) then
        b(Y) = Y; h(X) = h(Y) + c(Y, X)
  if k_old = h(X) then
    for each neighbor Y of X:
      if t(Y) = NEW or
        (b(Y) = X and h(Y) != h(X) + c(X, Y)) or
        (b(Y) != X and h(Y) > h(X) + c(X, Y)) then
        b(Y) = X; INSERT(Y, h(X) + c(X, Y))
  else
    for each neighbor Y of X:
      if t(Y) = NEW or
        (b(Y) = X and h(Y) != h(X) + c(X, Y)) then
        b(Y) = X; INSERT(Y, h(X) + c(X, Y))
      else
        if b(Y) != X and h(Y) > h(X) + c(X, Y) then
          INSERT(X, h(X))
        else
          if b(Y) v X and h(X) > h(Y) + c(Y, X) and
            t(Y) = CLOSED and h(Y) > k_old then
            INSERT(Y, h(Y))
  return GET-KMIN()
```

Las funciones *DELETE* e *INSERT* sirven para eliminar y agregar nodos a la cola de prioridad, respectivamente. *DELETE* solo marca un nodo como cerrado y lo retira de la cola, mientras que *INSERT* se encarga de actualizar los valores *k* de un nodo, dependiendo de si este está marcado como nuevo, abierto o cerrado. Si el nodo no está marcado como abierto, lo marca como tal y lo agrega a la lista. Por otra parte, si ya está marcado como abierto, solo actualiza su información en la cola de prioridad, en lugar de agregarlo nuevamente. Al finalizar el proceso, se reordena la cola de prioridad con base en los valores *k* actualizados.

3.4 Implementación del algoritmo D* (con replanificación de trayectorias)

Debido a la complejidad del algoritmo D* en este trabajo se presenta una versión modificada del mismo, específicamente en la parte de replanificación de trayectorias. Es decir que, si no hay cambios en el mapa, el algoritmo mostrado en la sección anterior retorna de manera exitosa la trayectoria óptima entre los dos nodos proporcionado.

Aunque el método utilizado para la replanificación de trayectorias desafía el propósito de utilizar el algoritmo D*, pues se descarta la información ya procesada para poder replanificar rutas, sigue dando resultados efectivos que no tienen un efecto notorio para efectos de este trabajo,

considerando las dimensiones del grafo que se está recorriendo.

El algoritmo utilizado consiste simplemente en ejecutar el proceso de expansión comenzando en el nodo meta y deteniéndose al llegar al nodo de inicio. Por la manera en que se implementó, el algoritmo siempre retornará una trayectoria, aún si esta pasa a través de obstáculos. Este comportamiento se usa de manera ventajosa, pues se puede seguir la trayectoria para determinar si alguno de los nodos que la conforma es un obstáculo. Si, efectivamente, uno de los nodos es un obstáculo, se actualiza la información del grafo, se reinicia toda la información procesada y se inicia el proceso de expansión desde cero, esta vez yendo desde la meta hasta el nodo anterior al obstáculo detectado.

Si inmediatamente después de realizar la expansión nuevamente, al seguir la trayectoria existe otro obstáculo, significa que la replanificación no pudo encontrar una trayectoria entre ambos nodos, y el proceso de búsqueda termina con resultados negativos. En caso contrario, si se encuentra una trayectoria entre ambos nodos, se puede dar por terminado el proceso con resultados positivos.

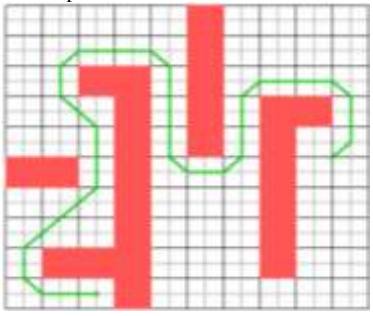
4. RESULTADOS

A continuación, se presentan los resultados obtenidos por los algoritmos A*, D* y D* modificado en términos de longitud de trayectoria y tiempo de ejecución. Los resultados se obtuvieron en una computadora portátil con un procesador Intel Core i7-8850H (6 núcleos, 12 procesadores lógicos) a una velocidad de reloj de 4.20 GHz y 32 GB de memoria RAM tipo DDR4 a una velocidad de reloj de 2667 MHz. Es importante mencionar que todos los cálculos se ejecutaron exclusivamente en el procesador de la computadora.

4.1 Algoritmo A*

En la Figura 3. Ruta óptima encontrada para un entorno complejo con múltiples obstáculos utilizando A*. se muestra la ruta óptima encontrada para un entorno complejo con múltiples obstáculos. La Figura 3 muestra el resultado de la ejecución del algoritmo A*. La ruta encontrada es la ruta óptima entre los nodos de inicio y de meta definidos. Se observa que, al cambiar la vecindad utilizada y la resolución, la ruta puede cambiar de manera considerable, y en algunos casos, hasta dejar de existir.

Figura 3. Ruta óptima encontrada para un entorno complejo con múltiples obstáculos utilizando A*.



4.2 Algoritmo D*

Al igual que con el algoritmo A*, el algoritmo D* entrega resultados satisfactorios en todos los casos. La parte de expansión funciona de la manera que se planteó inicialmente. Se observa, sin embargo, que existen pequeñas diferencias entre la trayectoria generada por el algoritmo A* y la generada por D*.

Al visualizar la trayectoria seguida por el robot móvil, este avanza hasta toparse con un muro (en la mayoría de los casos), y después gira hacia donde es conveniente. Si se compara la trayectoria generada por A* (Figura 3) con la generada por D* (Figura 4) se puede apreciar claramente que sí existe una diferencia entre la trayectoria generada al ir desde el nodo de inicio hasta el nodo meta (la trayectoria de A*) e ir desde el nodo meta hasta el nodo de inicio (la trayectoria de D*).

Figura 4. Ruta óptima encontrada para un entorno complejo con múltiples obstáculos utilizando D*.

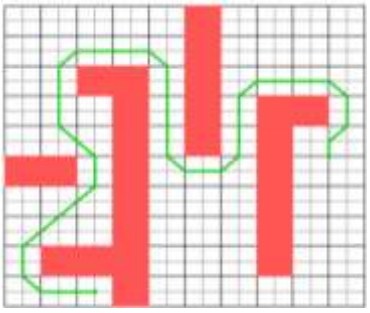


Figura 5. Ruta óptima encontrada para un entorno complejo con múltiples obstáculos utilizando D* (con un obstáculo colocado antes de iniciar la planificación).

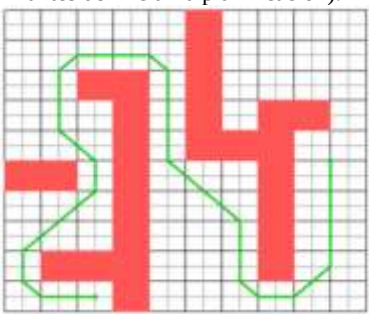
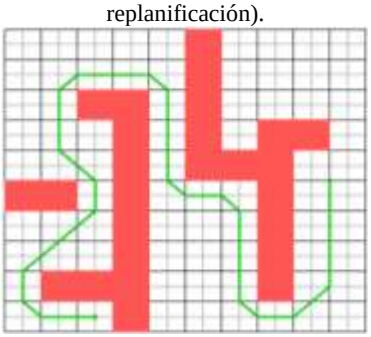


Figura 6. Ruta óptima encontrada para un entorno complejo con múltiples obstáculos utilizando D* (mostrando el resultado de la replanificación).



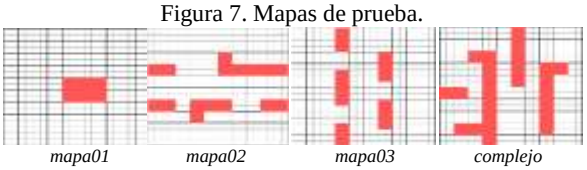
En la Figura 5 y Figura 6 se observa que la trayectoria generada sí cambia, pero sigue siendo la óptima si se coloca

un obstáculo en el mapa. En la Figura 5 se observa la trayectoria generada por D* si el obstáculo colocado en la posición (6,5) está en ese lugar desde el inicio de la ejecución del algoritmo, y en la Figura 6 se observa la trayectoria generada por D* cuando el obstáculo es colocado después de realizar la planificación inicial. Es visible de manera clara en qué momento ocurrió el proceso de replanificación.

4.3 Tiempos de ejecución y rutas encontradas

Se procedió a capturar los tiempos de ejecución de los tres algoritmos para diferentes configuraciones del entorno. Se utilizaron cuatro mapas con la información de los obstáculos, llamados *mapa01*, *mapa02*, *mapa03* y *complejo*, respectivamente. En la Figura 7 se pueden observar la configuración de los mapas de prueba.

Figura 7. Mapas de prueba.



mapa01 *mapa02* *mapa03* *complejo*

En la Tabla 1 se observan los tiempos de ejecución del algoritmo A* para los cuatro mapas mencionados anteriormente. Se observa que, opuesto a lo que dice la intuición, para rutas más largas hay tiempos de ejecución menores. Hace falta realizar pruebas más rigurosas para verificar que el tiempo de ejecución tiene este comportamiento en todos los casos.

Tabla 1. Tiempo de ejecución del algoritmo A*.

Mapa	Tiempo [ms]	Distancia [m]
map01	21.920	7.121
map02	15.970	11.950
map03	20.770	11.071
complejo	11.768	26.899

Con el algoritmo D* sin replanificación se obtienen tiempos de ejecución de tres a cuatro veces más lentos que con el algoritmo A*, esto posiblemente por la manera en que se maneja cada uno de los nodos (cada nodo contiene más información en la implementación de D* que de A*). Sin embargo, no se observa el mismo patrón que con el algoritmo A*, en donde los tiempos de ejecución disminuyen para distancias mayores. En la Tabla 2 se observan los resultados la ejecución del algoritmo con los mismos mapas y puntos de inicio y meta que con A*.

Tabla 2. Tiempo de ejecución del algoritmo D* sin replanificación (2da columna) y del algoritmo D* modificado con replanificación (3ra columna).

Mapa	Tiempo [ms]	Tiempo [ms]	Distancia [m]
map01	63.790	65.990	7.121
map02	54.970	50.990	11.950
map03	68.990	73.000	11.071
complejo	75.030	75.020	26.899

Finalmente, para el algoritmo D* con una rutina de replanificación modificada se observan tiempos muy parecidos a los obtenidos para la variante sin replanificación. No obstante, hay un ligero incremento en los tiempos de ejecución, esto posiblemente por las verificaciones que se realizan con la ruta a todo momento para replanificar si es necesario. Se observan en la Tabla 2 los tiempos de ejecución resultantes para los mismos mapas y puntos de inicio y meta que con los otros dos algoritmos.

5. CONCLUSIONES Y RECOMENDACIONES

Mediante las implementaciones realizadas se obtuvieron resultados efectivos, esto motiva la continuación en la investigación en temas de planificación de trayectorias óptimas, así como generar más investigación e información que permita comprender y ejecutar estos algoritmos de una manera efectiva, con objetivo de evitar problemas como el que se presentó al momento de escribir el algoritmo de replanificación.

Naturalmente, se obtuvieron los tiempos de ejecución menores para el algoritmo A*, pero esto se debe a que no tiene la capacidad de replanificar rutas, y no utiliza tiempo adicional para verificar que el mapa haya cambiado en ningún momento. Por otra parte, la implementación del algoritmo de replanificación diferente al original parece no afectar considerablemente el rendimiento, pues los cambios presentados se reflejan en una diferencia de cinco milisegundos en cada ejecución del algoritmo.

Es importante destacar que utilizar una implementación más sencilla del algoritmo de planificación de trayectorias puede evitar complicaciones en la implementación del algoritmo D* original, pero puede resultar en tiempos de ejecución mucho mayores en entornos más grandes que los probados en este trabajo. Se recomienda utilizar implementaciones más sencillas del algoritmo D* para este tipo de situaciones, tales como D* Lite o D* enfocado, dependiendo de las necesidades de la implementación.

REFERENCIAS

[1] U. Orozco-Rosas, K. Picos y O. Montiel, «Hybrid path planning algorithm based on membrane pseudo-bacterial potential field for autonomous mobile robots,» *IEEE Access*, vol. 7, nº 1, pp. 156787-156803, 2019.

[2] T. Cormen, C. Leiserson, R. Rivest y C. Stein, Introduction to Algorithms, 3 ed., Cambridge, Massachusetts: The MIT Press, 2009.

[3] S. LaValle, Planning Algorithms, Cambridge: Cambridge University Press, 2006.

[4] U. Orozco-Rosas, O. Montiel y R. Sepúlveda, «Mobile robot path planning using membrane evolutionary artificial potential field,» *Applied Soft Computing*, vol. 77, pp. 236-251, 2019.

[5] A. Stentz, «Optimal and efficient path planning for partially-known environments,» *Proceedings of the 1994 IEEE International Conference on Robotics and Automation*, Mayo 1994.

[6] H. Choset, *Robotic Motion Planning: A* and D* Search*, Carnegie Melon School of Computer Science, 2007.

[7] B. Williams, *Incremental Path Planning: Dynamic and Incremental A**, Massachusetts Institute of Technology, 2004.