Centro de Investigación en Matemáticas A.C.
Maestría en Ciencias de la Computación y Matemáticas Industriales
Programación y Algoritmos I (2019)

Proyecto final. Algoritmo A*

Marco Antonio Esquivel Basaldua

Ir de un punto A a un punto B es un problema recurrente en aplicaciones como la robótica, la computación y la algoritmia se han encargado con el tiempo de dar solución a este problema siendo el algoritmo *A** el que lo hace de manera más óptima. Cuando avanzar un paso en cierta dirección es más costoso que dar un paso en otra dirección se necesita de implementar, además, el algoritmo *Dijkstra*.

En este trabajo se presenta la implementación de los algoritmos *A** y *Dijkstra* para encontrar el camino de un punto A a un punto B en un espacio cuadriculado en dos dimensiones. En caso de que un camino exista entre ambos puntos se reportará el óptimo, es decir el que sea menos costoso de recorrer. Si no existe forma de unir ambos puntos se dará aviso del resultado.

Introducción

El algoritmo A* es un algoritmo de búsqueda en grafos presentado por Peter E. Hart, Nils J. Nilsson y Bertram Raphael en el año 1968. Su función es encontrar el camino de menor costo entre un nodo de inicio y uno de destino tal como los algoritmos *Breath First Search* (*BFS*) y *Depth First Search* (*DFS*) pero utilizando una función de heurística para optimizar la búsqueda sacando provecho a la información sobre la ubicación del nodo de destino. Estos tres algoritmos (*BFS*, *DFS* y *A**) son considerados algoritmos de búsqueda **completos**, ya que de existir un camino que una el nodo de inicio con el nodo de destino, éste será encontrado.

Para explicar el funcionamiento del algoritmo *A**, primero se describen brevemente los algoritmos *BFS* y *DFS* además del algoritmo de *Dijkstra* ya que su funcionamiento fue utilizado en la realización de este proyecto.

Breath First Search (BFS)

Llamado en español *Búsqueda en anchura*, es un algoritmo de búsqueda no informada en el que se exploran todos los vecinos del nodo de inicio. En seguida, para cada uno de los vecinos se exploran sus respectivos vecinos adyacentes. Este proceso se continua hasta dar con el nodo de destino o terminar de explorar todo el mapa o grafo.

Utilizando este algoritmo en un espacio de dos dimensiones, la búsqueda sería equivalente a imaginar el recorrido que hace una perturbación sobre la superficie de un estanque de agua calmada.

La complejidad computacional del algoritmo se puede expresar como $O(|V| + |E|)$, donde $|V|$ es el número de vértices y $|E|$ es el número de aristas. El razonamiento es porque en el peor caso, cada vértice y cada arista será visitado por el algoritmo.

Depth First Search (DFS)

Llamado búsqueda en anchura, es un algoritmo de búsqueda no informada, al igual que el *BFS* pero con la diferencia que la búsqueda se realiza de manera ordenada expandiendo todos y cada uno de los nodos que va localizando de forma recursiva. Cuando ya no existen más nodos que visitar en el camino actual, el algoritmo regresa. Se sigue este proceso hasta dar con el nodo de destino o terminar de recorrer todos los nodos del mapa o grafo.

Imagínese la implementación de este algoritmo en un mapa de dos dimensiones como el recorrido radial que realiza la luz de un faro (considérese que el faro existe en ese espacio de dos dimensiones). Los lugares iluminados por la luz son los lugares explorados por el algoritmo.

En el peor caso la complejidad del algoritmo es $O(b^m)$, siendo b el factor de ramificación (número promedio de ramificaciones por nodo) y m la máxima profundidad del espacio de estados.

A*

Este algoritmo realiza una búsqueda tanto en anchura como en profundidad utilizando la información dada sobre el nodo de destino. A partir de los vecinos del nodo de origen, se evalúa la distancia de cada uno de los nodos hacia el nodo de destino, se prioriza entonces continuar la búsqueda a partir del nodo que haya reportado una menor distancia, en otras palabras, a partir del nodo mas cercano al destino. De acuerdo a la aplicación del algoritmo y al movimiento permitido en la consideración de los vecinos de un nodo, la distancia medida puede ser la distancia euclidiana o la distancia manhattan.

Este algoritmo es comúnmente utilizado en las ciencias de la computación debido a su optimalidad ya que encuentra, en caso de existir, el camino con menor costo hacia un nodo de destino visitando la menor cantidad de nodos.

La complejidad computacional del algoritmo está íntimamente relacionada con la calidad de la heurística que se utilice en el problema. En el caso peor, con una heurística de pésima calidad, la complejidad será exponencial, mientras que en el caso mejor, con una buena $h'(n)$, el algoritmo se ejecutará en tiempo lineal. Para que esto último suceda, se debe cumplir que:

$h'(x) \leq g(y) - g(x) + h'(y)$

donde h' es una heurística óptima para el problema, como por ejemplo, el coste real de alcanzar el objetivo.

Dijkstra

El algoritmo de *Dijkstra* es utilizado cuando ir de un nodo a otro en el grafo o mapa tiene diferentes costos. Imagínese que se tiene que ir de un punto A a un punto B, se podría pensar que la distancia más corta para llegar a B es en línea recta pero qué tal que esa línea recta esté atravesada por una inmensa montaña. En este escenario podría ser más sencillo rodear la montaña que atravesarla para llegar de A a B, cuya distancia podría ser mayor pero representaría un costo menor. Utilizando este algoritmo se estarían tomando en cuenta los pesos o costos de ir de un nodo al siguiente para dar prioridad a la búsqueda del camino.

Metodología

Dado un mapa cuadriculado en dos dimensiones se desea ir de un punto **Hunter** a un punto **Prey** en un modelo *presa-cazador*. El objetivo es determinar el camino óptimo que debe recorrer el cazador (a partir del punto **Hunter**) hasta la presa ubicada en la posición **Prey**. El costo para el cazador de avanzar un espacio (hacia arriba, abajo, izquierda, derecha o en diagonales) es de una unidad, sin embargo existen obstáculos en el mapa cuya posición no puede ser explorada ni atravesada por el cazador (llamados obstáculos de *tipo 0*), otro tipo de obstáculos que añaden un peso adicional de una unidad el ser atravesadas (obstáculos de *tipo 1*) y otros más que agregan un costo de 1.5 (obstáculos de *tipo 1.5*). Un ejemplo de la configuración de este mapa puede apreciarse en la figura 1.

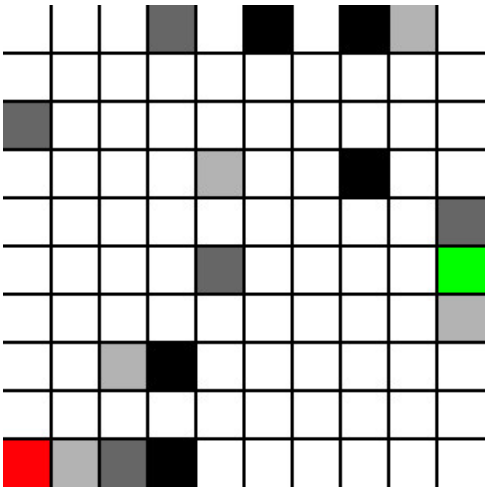

Figura 1. Mapa de 10x10 con obstáculos y cazador y presa ubicados.

Los espacios en blanco representan posiciones libres, los cuadros en negro son obstáculos tipo 0, los espacios en gris representan obstáculos de tipo 1 y 1.5 siendo los más claros los de tipo 1. El cazador se encuentra en la posición roja y su objetivo es llegar a la presa en la posición verde.

Debido a la existencia de espacios con diferente peso (obstáculos de *tipo 1* y *tipo 1.5*) se implementa el algoritmo *A** en conjunto al algoritmo *Dijkstra* para encontrar el camino óptimo. Aplicados al ejemplo de la figura 1 se obtienen los resultados que se muestran en la figura 2.

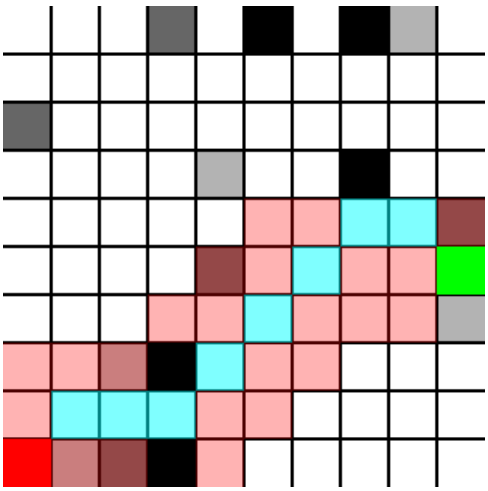

Figura 2. Camino del cazador hacia la presa del ejemplo en la figura 1.

En la figura 2 los espacios en azul representan uno de los caminos óptimos (puede existir más de uno) que puede tomar el cazador para llegar a su presa, los espacios en rojo son el espacio explorado antes de localizar a la presa, su transparencia permite observar cuales y cuántos obstáculos fueron explorados y si fue necesario atravesar alguno de ellos para llegar al destino.

En este ejemplo se reporta un costo de camino óptimo de 9 unidades.

Mapa y obstáculos

Un mapa, como los mostrados en las figuras 1 y 2, es representado dentro del código mediante una matriz cuadrada de valores `double` en la que los espacios vacíos y sin explorar son representados por el valor 1, los obstáculos *tipo 0* son representados con el valor -10, los de *tipo 1* con el valor 2 y los de *tipo 1.5* con el valor 2.5. Mientras el algoritmo se ejecuta los valores de los espacios que se van explorando van tomando el valor correspondiente a la suma de los espacios que se exploraron hasta llegar a ese punto, para esto se hace uso de la programación dinámica.

La ubicación de los obstáculos en el mapa se determina de manera aleatoria, sin embargo el usuario es el responsable de determinar el tamaño del mapa (siendo este siempre un cuadrado) y el porcentaje del mismo que será dedicado a cada uno de los obstáculos.

En el ejemplo de las figuras 1 y 2 se tiene un mapa de tamaño 10 con un porcentaje del 5% para cada tipo de obstáculo (se puede elegir un porcentaje distinto por obstáculo). El código se encuentra blindado para que la suma de los porcentajes de los obstáculos no sea mayor al 100%.

El usuario es también responsable de decidir las posiciones del cazador y la presa. Para ello primero se realiza e imprime el mapa con los obstáculos, así el usuario puede usar su configuración como guía para colocarlos. Las coordenadas del cazador y la presa se dan al tipo de una matriz, comenzando por el número de la fila de arriba hacia abajo y después el número de la columna de izquierda a derecha, la numeración comienza desde cero. Así, para el ejemplo anterior, la ubicación del cazador es $(9, 0)$ y la de la presa e $(5, 9)$.

El código se encuentra blindado para no ubicar al cazador y a su presa en espacios válidos, es decir que se encuentren dentro de los límites del mapa y que el espacio seleccionado no sea un obstáculo de cualquier tipo.

POO y clases

Este proyecto utiliza la programación orientada a objetos (POO) del lenguaje c++. Para su implementación se cuenta con dos clases: `Draw_map` y `Hunting`.

La clase `Draw_map` es la encargada de crear el mapa con los obstáculos. Para su instanciamiento se requiere indicar como parámetros de entrada el porcentaje de cada uno de los obstáculos así como el tamaño del mapa. Los métodos implementados en esta clase son los siguientes.

Método	Descripción
`void fillMapa();`	Es el encargado de llenar la matriz (que representa el mapa) de acuerdo a lo solicitado por el usuario.
`void graphCairo();`	Una vez generada la matriz, este método se encarga de dibujar la cuadrícula del mapa y guardarla en el archivo "Map.png".
`vector<vector<double> > get_mapa();`	Este es un "getter" por el cual se puede recuperar la matriz que representa el mapa.

La clase `Hunting` lleva a cabo la implementación del algoritmo *A** y genera el mapa final como el mostrado en la figura 2. En su instanciamiento se requiere dar como entradas la matriz del mapa de obstáculos (se obtiene de la clase `Draw_map`) y las posiciones del cazador y la presa los cuales son ambos un par de valores enteros. Los métodos utilizados por esta clase son los siguientes.

Método	Descripción
`void A_star();`	Implementación del algoritmo *A**.
`void graphCairo();`	Este método se encarga de dibujar la cuadrícula del mapa junto con las posiciones del cazador y la presa, el espacio explorado y el camino encontrado. El resultado se guarda en el archivo "Hunt Path.png".
`double dist_eucl(pair<int,int>);`	Calcula la distancia euclidiana desde un punto de entrada hacia la ubicación de la presa.
`void first_search(int,int,int,int);`	Explora los vecinos del cazador.
`void second_search(int,int,int,int);`	Explora a partir de los vecinos del cazador.
`void writePath(vector<pair<int,int> >);`	Escribe en el archivo de texto "Path Direction.txt" los puntos que debe seguir el cazador para llegar a la presa.

Heurística

Se mencionó con anterioridad que el algoritmo *A** utiliza la información sobre la localización de la presa para encontrar mucho más rápido el camino óptimo que vaya del cazador a la presa. Esta información, llamada también heurística, es la distancia euclidiana a partir de la posición que se esté considerando en ese momento hacia la ubicación de la presa.

La optimización del tiempo de búsqueda se lleva a cabo utilizando colas de prioridad de la **STL** del lenguaje c++. Cada vez que se exploran los vecinos de una ubicación, éstos son guardados en una cola de prioridad cuyo ordenamiento se realiza con respecto al valor negativo de la distancia euclidiana hacia la ubicación de la presa. La razón de utilizar el valor negativo es porque la cola de prioridad pone en primer posición el valor más grande encontrado, en la implementación del algoritmo *A** nos interesa priorizar en cuanto a la menor distancia euclidiana, dar un valor negativo a esta distancia soluciona este problema.

Mientras se va explorando el espacio, los puntos cuyos vecinos serán explorados son los que se obtienen a partir de la cola de prioridad.

Modo interactivo

Una vez que se indica las posiciones de cazador y presa y se muestra el camino del cazador hacia la presa, es posible interactuar con el mapa desplazando la presa a una posición adyacente a su ubicación actual. Para realizar esto, se usa la disposición del teclado matricial numérico (figura 3). Suponiendo que la ubicación actual de la presa es la posición del numero 5, presionando alguna de las teclas restantes indica el desplazamiento de la presa a una posición vecina de acuerdo a la disposición de los números en el teclado. Por ejemplo si se presiona el valor 8 se está indicando a la presa que se mueva una posición hacia arriba de su ubicación actual, presionar el número 3 indica a la presa el desplazamiento una unidad en diagonal hacia abajo y hacia la derecha.

Figura 3. Teclado numérico matricial.

Para que los resultados tengan efecto, una vez que se presione la tecla indicando la posición a la que se quiere avanzar, se debe presionar la tecla *Enter*. Se recomienda abrir la imagen "Map.png" ya que los cambios generados serán mostrados sobre ella.

En el caso en que se indique avanzar a una posición inválida (ubicación fuera del mapa, o ubicación de un obstáculo) se desplegará un mensaje indicando que se ingreso una posición invalida y se invitará al usuario a introducir otro valor.

Para detener la ejecución del programa se debe presionar la tecla 0.

Resultados

Además del ejemplo presentado en las figuras 1 y 2, en seguida se muestran los resultados obtenidos para mapas de tamaño 20 con distintos escenarios en cuanto a los porcentajes de los diferentes tipos de obstáculos.

Mapa sin obstáculos

Al indicar el tamaño del mapa igual a 20 y los porcentajes de los obstáculos en 0 se obtiene el mapa de obstáculos de la figura 3.

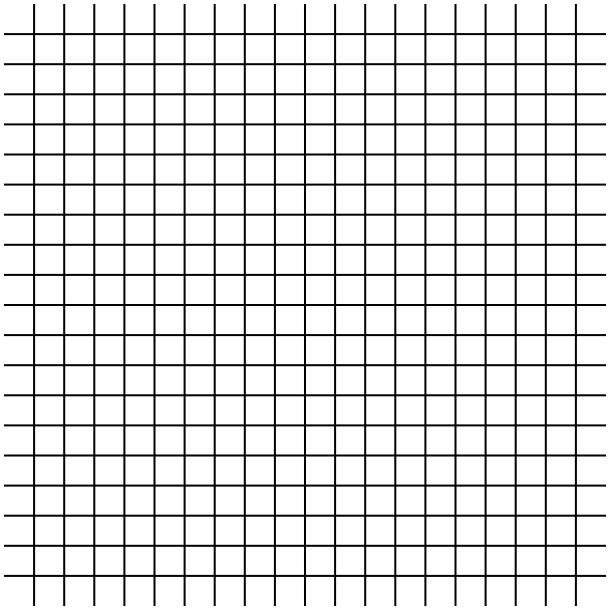

Figura 4. Mapa de 20x20 sin obstáculos.

A continuación se deciden las posiciones que ocuparán el cazador y la presa. En este caso se eligen la posición $(0, 0)$ para el cazador y $(19, 19)$ para la presa. La presa es localizada con un costo de camino de 19 unidades. El mapa generado se muestra en la figura 4. En la figura 5 se muestran las instrucciones y mensajes por consola para generar los mapas.

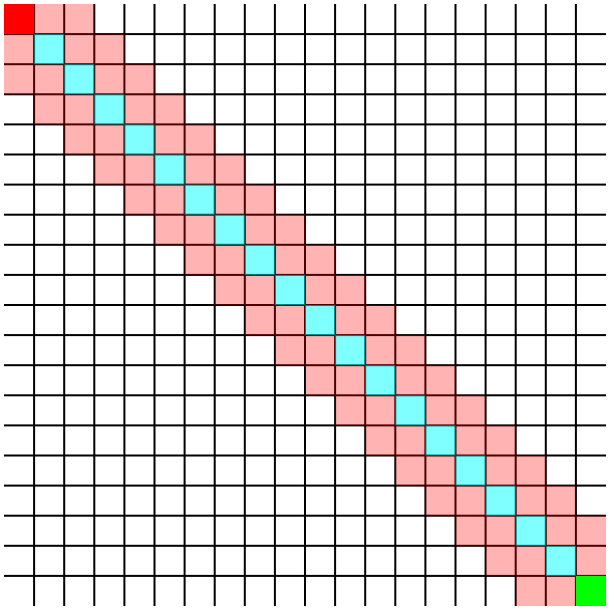

Figura 5. Mapa del camino generado.

```
El mapa de búsqueda es un cuadrado
Ingrese el tamaño de uno de sus lados
20
Ingrese en porcentaje la cantidad de obstáculos tipo 0, tipo 1 y tipo 1.5 separados por un espacio
0 0 0

Mapa con los obstàculos generado

Ingrese las coordenadas (tipo matriz) del cazador separados por un espacio
0 0
Ingrese las coordenadas (tipo matriz) de la presa separados por un espacio
19 19
La presa fue localizada
Costo del camino para llegar a ella: 19
```

Figura 6. Instrucciones y mensajes por consola.

Mapa con obstáculos en 5% 10% y 20%

Al indicar el tamaño del mapa igual a 20 y los porcentajes de los obstáculos en 5% para los de *tipo 0*, 10% para los de *tipo 1* y 20% para los de *tipo 1.5*, se obtiene el mapa de obstáculos de la figura 6.

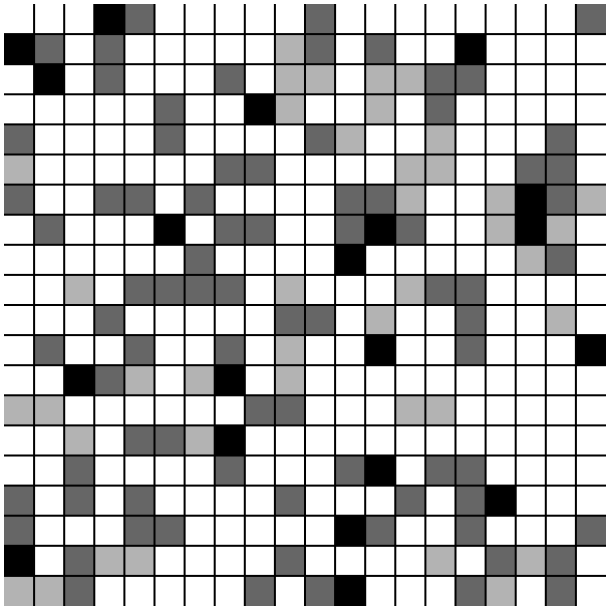

Figura 7. Mapa de 20x20 con obstáculos en 5%, 10% y 20%.

A continuación se deciden las posiciones que ocuparán el cazador y la presa. En este caso se eligen la posición $(10, 7)$ para el cazador y $(3, 15)$ para la presa. La presa es localizada con un costo de camino de 10 unidades. El mapa generado se muestra en la figura 7. En la figura 8 se muestran las instrucciones y mensajes por consola para generar los mapas.

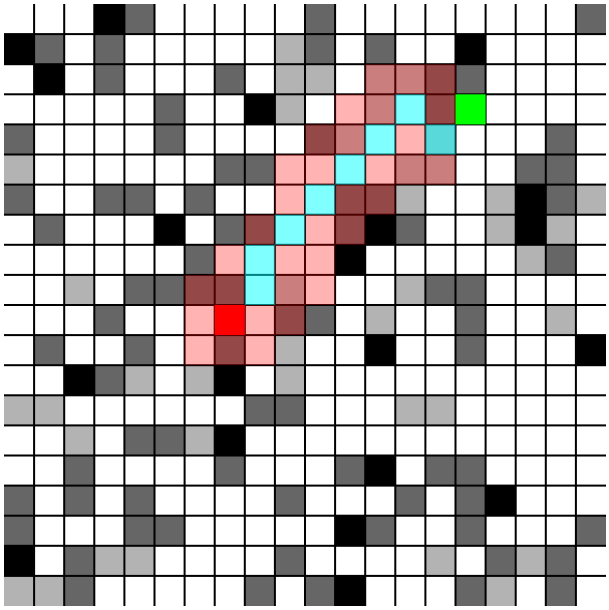

Figura 8. Mapa del camino generado.

```
El mapa de búsqueda es un cuadrado
Ingrese el tamaño de uno de sus lados
20
Ingrese en porcentaje la cantidad de obstáculos tipo 0, tipo 1 y tipo 1.5 separados por un espacio
5 10 20

Mapa con los obstàculos generado

Ingrese las coordenadas (tipo matriz) del cazador separados por un espacio
11 7
Ubicación inválida para el cazador. Vuelva a ingresar las coordenadas
10 7
Ingrese las coordenadas (tipo matriz) de la presa separados por un espacio
3 15
La presa fue localizada
Costo del camino para llegar a ella: 10
```

Figura 9. Instrucciones y mensajes por consola.

Mapa con obstáculos en 20% 10% y 5%

Al indicar el tamaño del mapa igual a 20 y los porcentajes de los obstáculos en 20% para los de *tipo 0*, 10% para los de *tipo 1* y 5% para los de *tipo 1.5*, se obtiene el mapa de obstáculos de la figura 9.

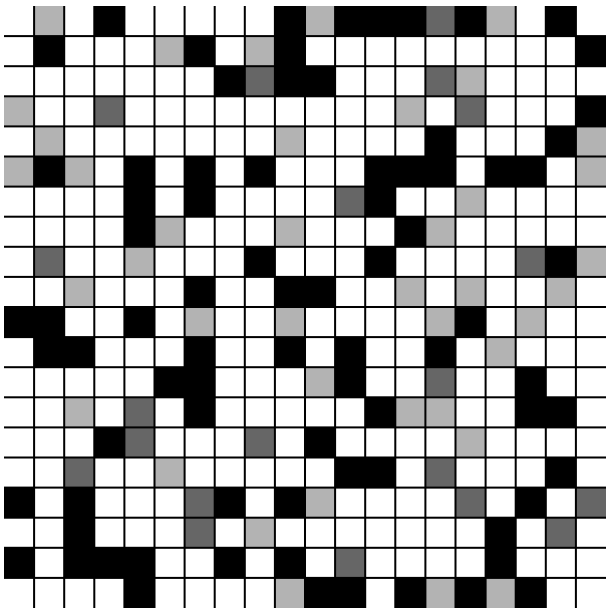

Figura 10. Mapa de 20x20 con obstáculos en 20%, 10% y 5%.

A continuación se deciden las posiciones que ocuparán el cazador y la presa. En este caso se eligen la posición $(16, 14)$ para el cazador y $(17, 1)$ para la presa. La presa es localizada con un costo de camino de 16 unidades. El mapa generado se muestra en la figura 10. En la figura 11 se muestran las instrucciones y mensajes por consola para generar los mapas.

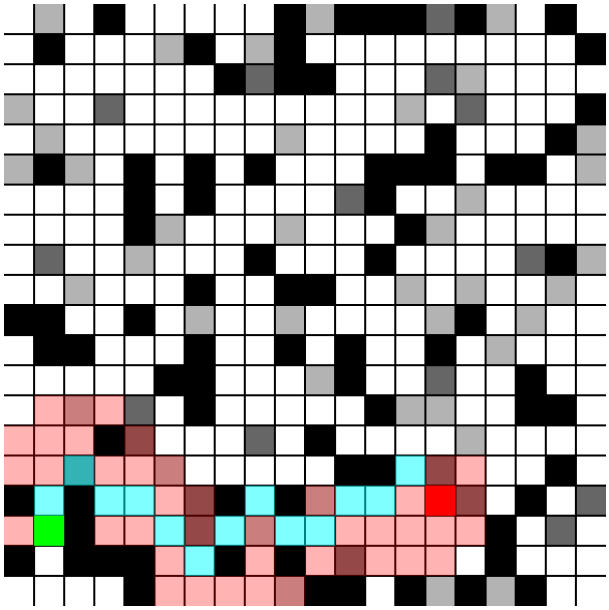

Figura 11. Mapa del camino generado.

```
El mapa de búsqueda es un cuadrado
Ingrese el tamaño de uno de sus lados
20
Ingrese en porcentaje la cantidad de obstáculos tipo 0, tipo 1 y tipo 1.5 separados por un espacio
20 10 5

Mapa con los obstàculos generado

Ingrese las coordenadas (tipo matriz) del cazador separados por un espacio
16 14
Ingrese las coordenadas (tipo matriz) de la presa separados por un espacio
17 1
La presa fue localizada
Costo del camino para llegar a ella: 16
```

Figura 12. Instrucciones y mensajes por consola.

Mapa con obstáculos tipo 0 en 50%

Al indicar el tamaño del mapa igual a 20 únicamente con obstáculos *tipo 0* en un 50% se obtiene el mapa de obstáculos de la figura 12.

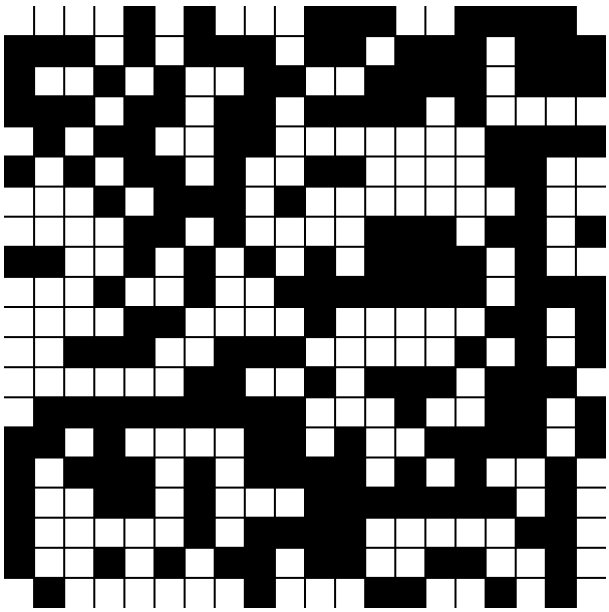

Figura 13. Mapa de 20x20 solo con obstáculos tipo 0 en 50%.

A continuación se deciden las posiciones que ocuparán el cazador y la presa. En este caso se eligen la posición $(1, 16)$ para el cazador y $(19, 5)$ para la presa. En este caso la presa no fue localizada ya que se encuentra en un espacio inalcanzable por el cazador. El mapa generado se muestra en la figura 10. En la figura 11 se muestran las instrucciones y mensajes por consola para generar los mapas.

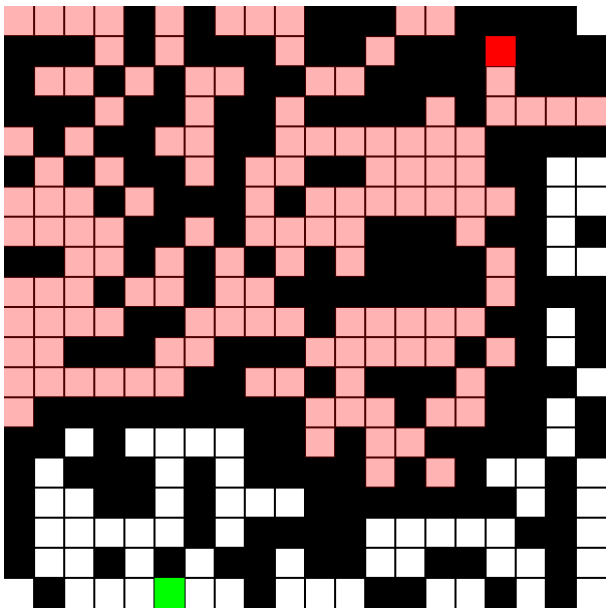

Figura 14. Mapa del camino generado.

```
El mapa de búsqueda es un cuadrado
Ingrese el tamaño de uno de sus lados
20
Ingrese en porcentaje la cantidad de obstáculos tipo 0, tipo 1 y tipo 1.5 separados por un espacio
50 0 0

Mapa con los obstàculos generado

Ingrese las coordenadas (tipo matriz) del cazador separados por un espacio
1 16
Ingrese las coordenadas (tipo matriz) de la presa separados por un espacio
19 5

La presa no puede ser alcanzada
```

Figura 15. Instrucciones y mensajes por consola.

Conclusiones

La aplicación de la programación dinámica en este proyecto permitió de trabajar unicamente sobre una matriz de valores *double* que se actualiza conforme se vaya explorando el mapa, sin embargo se conserva la matriz original de obstáculos para volver a dibujarla en usando las funciones de *Cairo* y en seguida dibujar el espacio explorado y el camino encontrado.

Se hizo uso de la distancia euclidiana en lugar de la distancia manhattan como parámetro de prioridad para los nodos visitados ya que se considera como válido avanzar posiciones en diagonal. En aplicaciones donde solo se puede avanzar hacia arriba, abajo, izquierda o derecha la distancia manhattan es un buen criterio de para priorizar la exploración.

Se puede buscar la manera de reducir las lineas de código usadas en este proyecto. La razón de su extensión es que existen bastantes evaluaciones *if*. Cada vez que se busca explorar en el mapa se tiene que evaluar si el espacio que se quiere evaluar está dentro del mapa y si se trata de un lugar válido para visitar.

Una posible aplicación para el algoritmo *A** que evalúa los espacios al estilo *Dijkstra* es para determinar rutas óptimas de forma automática en robótica, suponiendo que se tiene un robot en una posición A y se quiere que llegue a una posición B en un espacio con obstáculos. Una opción adicional para esta aplicación es evaluar el costo del robot para visitar cada uno de sus

nodos vecinos tomando en cuenta que tenga que girar para ir a puntos a sus costados y dar media vuelta para ir al espacio detrás de él o viajar en reversa.

Como punto adicional, el código genera un archivo de texto llamado "Path Directions.txt" donde se encuentran las coordenadas de los puntos en la cuadricula, ordenados, que el cazador debe seguir para llegar a su presa. En la aplicación de robótica, este archivo puede usarse como entrada para indicar a las instrucciones del robot qué camino debe seguir para llegar a su destino.