Escuela Superior
de Ingeniería y Tecnología
Universidad de La Laguna

ESTRATEGIAS DE BÚSQUEDA

Inteligencia Artificial

Descripción breve

En este informe nos vamos a plantear un problema de estrategias de búsqueda, con el objetivo de determinar una trayectoria para coches autónomos

Yeixon Reinaldo Morales González
Luciano Sekulic Gregoris
Adrián Epifanio Rodríguez Hernández

Escuela Superior
de Ingeniería y Tecnología
Universidad de La Laguna

Índice

1. Introducción

¿Qué es una estrategia de búsqueda?

Una estrategia de búsqueda se define como el conjunto de procedimientos y operaciones que un usuario realiza con el fin de obtener la información necesaria para resolver un problema.

Descripción del problema a resolver

En esta práctica nos vamos a plantear un problema de estrategias de búsqueda, con el objetivo de determinar una trayectoria para coches autónomos.

El entorno será rectangular de *M x N* de celdas libres y ocupadas, en el cual el coche se moverá una casilla cada vez a sus correspondientes vecinas circundantes libres que no estén ocupadas (con obstáculos). El coche contendrá un sensor que le indique si sus casillas adyacentes (Norte, Sur, Este, Oeste) están libres u ocupadas por obstáculos. Además, no estará permitido que el vehículo se mueva en diagonal. Por otro lado, el coste de cada movimiento sin importancia de la dirección los trataremos como unitario, es decir, todos los movimientos dentro del entorno de soluciones tendrán el mismo coste.

Nuestro cometido es resolver este problema para que el coche autónomo llegue a su destino final sin estrellarse con ningún obstáculo, utilizando los algoritmos A* y greedy con estrategias las heurísticas de la distancia euclídea y la distancia de manhattan para encontrar el camino óptimo para el coche desde la posición inicial hasta su posición final. Los resultados dados en dichas pruebas nos servirán para su posterior evaluación y comparación para encontrar el mejor en cada caso.

Formulación del problema como un espacio de estados

Se puede definir el problema como una combinación de estados en los que cada estado representa la posición del coche en un mapa con sus obstáculos. Es decir, podríamos representar el problema como una matriz en los que cada posición del mapa con un obstáculo puede ser representada como un 1 y cada posición vacía como un 0. Para nuestro caso particular además de representar con un 0 las posiciones vacías y con un 1 los obstáculos, hemos decidido representar con un 3 la casilla de salida y con un 4 la casilla de llegada y con un 2 cada casilla sin obstáculos por la que ha pasado el coche. Por lo tanto, definimos nuestro conjunto de estados como todas las posibles posiciones del mapa a las que puede llegar nuestro coche. Por lo tanto, suponiendo un mapa vacío nuestro entorno sería de *M x N* estados.

Función objetivo:

$$Min\ z\ =\ \sum_{i}^{MxN} x_i$$

sujeto a:

$$\underline{x\ \in \{0,1\}\ \ i\ =\ 0,1,2\ldots MxN}$$

donde:

$x_i\ =\ 0\ \rightarrow No\ x_i\ pertenece\ a\ la\ solución$
$x_i\ =\ 1\ \rightarrow Si\ x_i\ \ pertenece\ a\ la\ solución$

Definimos el estado inicial como el estado en el que el coche se encuentra en la casilla de salida sin haber realizado ningún movimiento. El estado final es el estado en el que el coche ha llegado a la casilla de meta o después de haber recorrido todo su conjunto de estados no ha encontrado solución posible.

Operadores:

- Movimiento al Norte: movimiento del coche hacia arriba (y - 1) en el mapa si no está esa posición ocupada por un obstáculo.
- Movimiento al Sur: movimiento del coche hacia abajo (y + 1) en el mapa si no está esa posición ocupada por un obstáculo.
- Movimiento al Este: movimiento del coche hacia la derecha (x + 1) en el mapa si no está esa posición ocupada por un obstáculo.
- Movimiento al Oeste: movimiento del coche hacia la izquierda (x - 1) en el mapa si no está esa posición ocupada por un obstáculo.

2. Entorno de simulación y programación

Al ejecutar el programa, nos enseña un menú, en el cual podemos seleccionar múltiples opciones:

```
=====   Welcome to the "Minimum Route Calculator For An Autonomous Car" simulator.   =====

 1. Introduce the game specs by keyboard.
 2. Load a data file.
 3. Generate random obstacles
 4. Introduce obstacles by keyboard
 5. Print map on screen.
 6. Print solution on screen.
 7. Save solution to file
 8. Save data to file for using it again
 9. Generate Solution
 10. Show trace
 11. Choose heuristic function
 12. Choose search algorithm
 13. Use openList solution
 0. Exit
```

1- Esta opción le pide al usuario las dimensiones del mapa, la posición inicial y final.

```
Introduce the number of rows: 200

Introduce the number of columns: 200

Introduce the start X position: 1

Introduce the start Y position: 1

Introduce the finish X position: 200

Introduce the finish Y position: 200
```

2- Esta opción permite cargar ficheros con un mapa.

```
Please enter the file name: ../inputs/Map2.txt
```

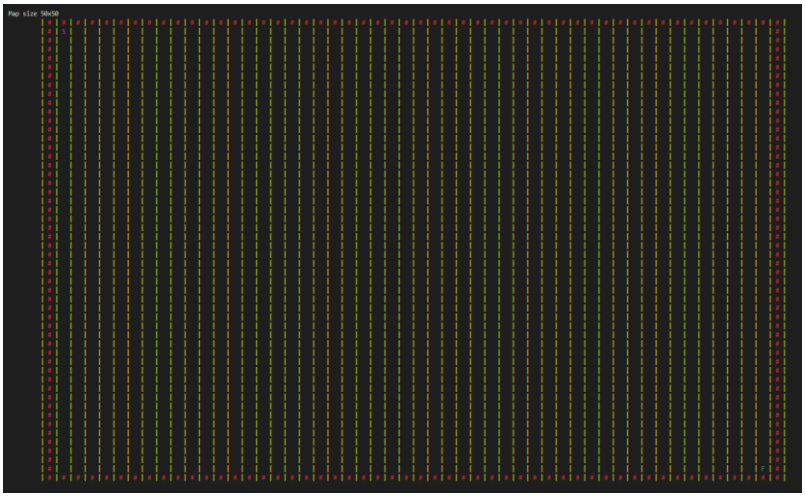
3- Permite meter obstáculos en posiciones random, indicando la cantidad que desees.

```
Please enter the ammount of obstacles: 1520
```

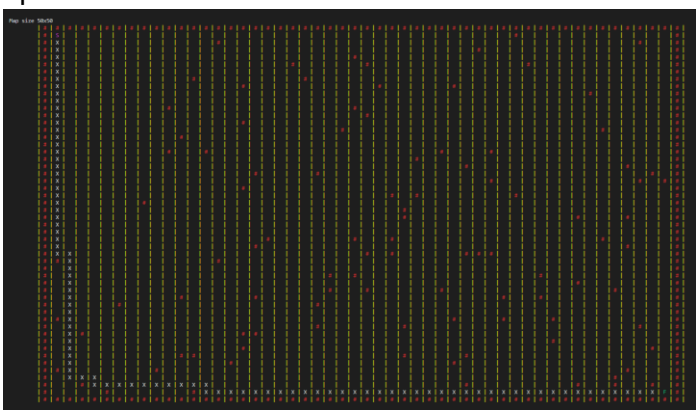
4- Permite meter obstáculos en las posiciones indicadas y la cantidad mediante teclado.

```
Please enter the ammount of obstacles: 5
5 remaining obstacles, introduce the x coordinate: 25
5 remaining obstacles, introduce the y coordinate: 5
4 remaining obstacles, introduce the x coordinate: 45
4 remaining obstacles, introduce the y coordinate: 42
3 remaining obstacles, introduce the x coordinate: 85
3 remaining obstacles, introduce the y coordinate: 79
2 remaining obstacles, introduce the x coordinate: 264
2 remaining obstacles, introduce the y coordinate: 45
1 remaining obstacles, introduce the x coordinate: 94
1 remaining obstacles, introduce the y coordinate: 52
```

5- Muestra el mapa generado, con o sin obstáculos.



6- Muestra el mapa con la solución generada, para ello tienes que generarla previamente en la opción 9.



7- Te permite guardar el mapa en un fichero de texto con la solución calculada.

```
Please enter the output file name: ../outputs/Map6.txt
```

8- Te permite guardar el mapa, pero sin solución para si luego deseas trabajar con ese mismo mapa.

```
Please enter the output file name: ../outputs/Map7.txt
```

Escuela Superior
de Ingeniería y Tecnología
Universidad de La Laguna

9- Genera la solución del mapa.

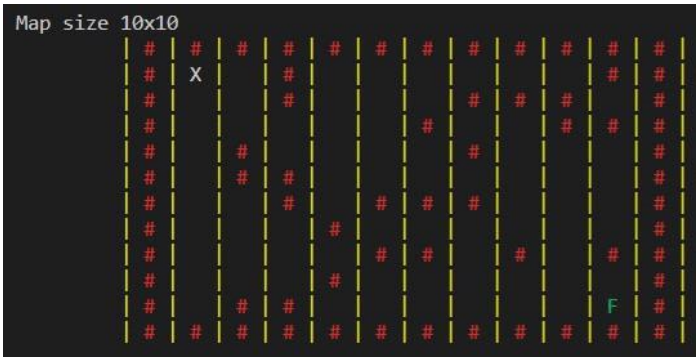
```
CPU Time: 0.006923 seconds.

Closed List: 108 nodes.

Open List: 223 nodes.

Path Length: 97 nodes.
```

10- Muestra la traza que realiza para llegar al final.



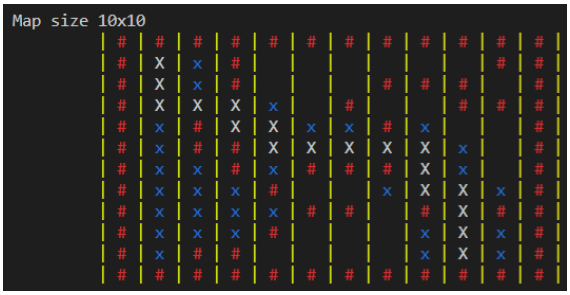
11- Permite elegir qué algoritmo usar para solucionar el mapa.

```
Please select the search algorithm you want to run
1. A-Star algorithm
2. Greedy algorithm
0. Cancel
```

12- Permite elegir qué función se va a usar en el algoritmo.

```
Please select the heuristic function you want to run
1. Euclidean distance
2. Manhattan distance
0. Cancel
```

13- Muestra todas las expansiones que ha realizado el algoritmo.



Leyenda:

Amraillo '|': Representan los bordes de las celdas.
Rojo '#': Representa los obstáculos.
Blanco 'X': Camino óptimo.
Azul 'x': Camino analizado.

Argumentación sobre el entorno de programación escogido

En primer lugar, hemos utilizado dos entornos de desarrollo integrado (IDE), ***Visual Studio Code*** (VSC) por parte de Yeixon Reinaldo y Luciano Sekulic y el ***Sublime Text*** por parte de Adrián Epifanio. Cada uno ha utilizado el IDE con el que se encontraba más cómodo y al que está más familiarizado para facilitar el trabajo.

Hemos decidido trabajar en varias ramas de desarrollo simultáneas y para poder programar de forma más ágil. Para ello hemos utilizado la plataforma GitHub para mantener un control de las distintas versiones del código. Cada vez que alguno de nosotros realizaba algún cambio o mejora en dicho código se subía a esta plataforma mencionando una breve descripción del cambio o mejora implementados en el código.

En cuanto al lenguaje de programación escogido, una de las principales razones de peso para decantarnos por C++, fue la familiarización que todos los componentes del grupo tenemos con este lenguaje de programación debido al uso que le hemos dado en proyectos anteriores. Por otro lado, fue clave su capacidad de orientación a objetos, el poder programar a alto nivel.

También tuvimos en cuenta la posibilidad que nos aporta este lenguaje de crear estructuras de datos complejas con una gran facilidad, definir operaciones sobre datos complejos y relacionarlos, la creación de templates (aunque en nuestro programa coincidimos que no eran necesarias) e implementar múltiples patrones de diseño. Por otro lado, debido a que es un lenguaje multiplataforma, nos ha permitido trabajar desde distintos sistemas operativos ya que no todos los componentes del grupo trabajamos exclusivamente con sistemas operativos de Linux o Windows, tiene infinidad de parámetros de optimización y permite acceder directamente a memoria las veces que desees, controlada por el propio usuario.

3. Metodología de trabajo

La resolución de esta práctica se ha llevado a cabo en grupo de manera completamente telemática, el grupo está formado por:

- Adrián Epifanio Rodríguez Hernández
- Yeixon Reinaldo Morales González
- Luciano Sekulic Gregoris

En un primer lugar comenzamos por el desarrollo del entorno en el que se iba a ejecutar nuestro programa. Para ellos definimos la estructura de datos que íbamos a implementar (Clases “Car”, “Game”, “Map” y “Sensor”). Una vez decididas las clases y atributos correspondientes a cada una de ellas procedimos al reparto de tareas. En este punto, Yeixon se encargó del desarrollo de las clases Car y Sensor, Luciano de la clase Game y Adrián de la clase Map.

Una vez finalizada la definición e implementación básica de dichas clases se procedió a la unión de las distintas ramas de trabajo mediante la plataforma de GitHub para comenzar las pruebas conjuntas, previo a esto cada clase había sido probada y testeada para comprobar su correcto funcionamiento.

Tras la primera revisión del código con la profesora, se decidió hacer ciertas mejoras sobre el entorno de ejecución de nuestro programa, en este caso se propuso la utilización de colores a la

hora de imprimir el mapa por pantalla o la posibilidad de permitir la lectura y escritura de datos mediante un fichero en lugar de introducirlos manualmente.

Para las tareas definidas previamente el reparto procedió de forma que nuestro compañero Luciano se encargó de la lectura desde un fichero de entrada y diseño del mismo. Adrián realizó la escritura del mapa en un fichero de salida y la escritura de los datos del problema a un fichero de salida para permitir la posibilidad de volver a cargar dicho mapa sin tener que introducir los obstáculos manualmente en dichas posiciones, dicho fichero de salida debía seguir el diseño del fichero de entrada que el programa reconoce diseñado por Luciano. Por último, Yeixon se encargó del desarrollo de la clase Colorize para permitir que al imprimir por pantalla se reconociera de forma más rápida el camino seguido futuramente por los algoritmos y los obstáculos junto con la casilla de salida y la de llegada.

En cuanto a la implementación de los distintos algoritmos el reparto fue el siguiente, Adrián se encargó del algoritmo A* y Luciano del algoritmo Greedy, teniendo en cuenta que se necesitaban una función heurística para guiarlos, posteriormente nuestro compañero Yeixon implementó, más adelante unimos los algoritmos y las heurísticas mediante GitHub, previamente probando su funcionamiento, para comprobar que algoritmo es el adecuado en cada caso.

4. Algoritmos de Búsqueda

Para el desarrollo de nuestra práctica, decidimos implementar 2 algoritmos distintos, en primer lugar, un algoritmo greedy, aun sabiendo que la solución que nos proporciona este algoritmo no tiene que ser la óptima y un algoritmo A-estrella. Decidimos realizar la implementación de al menos dos algoritmos para poder comparar los resultados de uno y otro y decidir cuál de los dos podría ser más eficiente en cada caso.

Pseudocódigo Algoritmo Greedy:

```
establecer mapa de trabajo
crear nodo raíz
insertar nodo raíz al árbol
si (nodo final rodeado de obstáculos)
      devolver falso
fin si
var = falso
mientras (var != verdadero)
      nodo = mejorNodoPosible
      si (coordenadas de coche en arbol[nodo] == coordenadas final)
            posicionSolucion = nodo
            devolver verdadero
      si no si (arbol[nodo] ha sido visitado)
            devolver falso
      si no
            expandir(arbol[nodo])
      fin si
mientras (var != verdadero)
```

Pseudocódigo Algoritmo A-estrella:

```
establecer mapa de trabajo
crear nodo raíz
insertar nodo raíz al árbol
si (nodo final rodeado de obstáculos)
      devolver falso
fin si
var = falso
mientras (var != verdadero)
      nodo = mejorNodoPosible
      si (coordenadas de coche en arbol[nodo] == coordenadas final)
            posicionSolucion = nodo
            devolver verdadero
      si no si (arbol[nodo] ha sido visitado)
            devolver falso
      si no
            expandir(arbol[nodo])
      fin si
mientras (var != verdadero)
```

Como se puede observar, el pseudocódigo del algoritmo A-estrella y el algoritmo greedy son prácticamente el mismo. La diferencia entre ambos algoritmos viene a la hora de expandir el nodo, en el caso de greedy se expande el nodo con menor función heurística y esta función está únicamente formada por la distancia desde la posición del coche hasta el final. Por otra parte, en el algoritmo A-estrella la distancia está formada por la distancia desde el coche hasta el final más la distancia que ha recorrido desde la casilla de salida hasta la posición actual del coche. Esto produce que el algoritmo A-estrella nos proporcione siempre en el caso de que el problema tenga solución, la solución óptima para la resolución de dicho problema, el inconveniente es que es mucho más costoso tanto de implementar como de calcular con el algoritmo A-estrella a con un algoritmo greedy, por eso, en muchos casos nuestro grupo considera que dada la mínima diferencia entre la solución mediante el algoritmo A-estrella y greedy, consideramos que greedy pueda ser más rentable en cuanto a costo de implementación y tiempo de cálculo. No obstante, todo dependerá de las necesidades del problema.

Estructuras de datos

En cuanto a las estructuras de datos utilizadas para el diseño de estos algoritmos, nosotros hemos optado por emplear herencias junto con un polimorfismo dinámico para permitirnos así cambiar de un algoritmo a otro en tiempo de ejecución sin necesidad de restablecer el programa o reiniciarlo por completo. Para ello hemos creado una clase padre *SearchAlgorithm* en la que implementamos como atributos un árbol de nodos “*tree_*” que es un vector de nodos. Cada nodo del árbol está compuesto por un par de unsigned que representa la posición del vehículo en ese nodo, un float para almacenar la distancia heurística f(n),un unsigned par almacenar la posición del padre del nodo en el árbol, otro unsigned para la distancia acumulada desde la salida hasta llegar a ese nodo (para el cálculo de f(n) en el algoritmo A-estrella), y un booleano para indicar si dicho nodo ha sido o no previamente visitado.

Además de el vector de nodos, otro de los atributos de nuestra clase *SearchAlgorithm* es un unsigned que almacena la posición del nodo final en en el árbol con la solución del problema si este lo tuviese. También tenemos un atributo de la clase Map que almacena el mapa con el que trabajará el algoritmo para no sobreescribir el mapa original. Por último tenemos un vector de unsigned para representar nuestra lista cerrada de nodos que almacena las posiciones de los nodos pertenecientes a la lista cerrada en el árbol.

Por otra parte, esta clase padre además de tener los métodos de get y set de los atributos mencionados también posee varios métodos, posee un método encargado de generar la lista cerrada de nodos una vez finalizado el algoritmo, otro para almacenar la lista abierta en el mapa original para poder visualizarlo posteriormente, un método para imprimir la traza que sigue nuestro algoritmo en el caso de que este encuentre solución al problema. También cuenta con un método que es el encargado de elegir cuál será el siguiente nodo a expandir comprobando cual es el nodo con menor valor de su función heurística y que no haya sido visitado previamente.

Por último, cuenta con un método virtual llamado *runAlgorithm* que será empleado para el polimorfismo dinámico en las clases hijas *A-starAlgorithm* y *GreedyAlgorithm*.

Funciones Heurísticas

Al igual que con los algoritmos, decidimos emplear dos funciones heurísticas distintas para poder comparar los valores y tiempos de CPU de una con la otra. En este caso nos decantamos por el uso de la función euclídea y la función de manhattan. Decidimos emplear estas dos funciones porque funcionan de forma similar y son válidas para la resolución de nuestro problema.

La función euclídea nos permite calcular una ruta más “directa” hacia la casilla final mientras que la de manhattan suele avanzar primero en una dirección y luego en otra. Finalmente, la longitud del camino entre ambas con el algoritmo A-estrella debe ser la misma ya que este algoritmo solo nos proporciona soluciones óptimas.

Tras ejecutar varias pruebas nos hemos dado cuenta de que la función de manhattan aporta mejores resultados en cuanto a tiempo de CPU y sobre todo a cantidad de nodos generados. Creemos que esto es debido a que, en nuestro caso particular del problema, todos los movimientos tienen el mismo coste. Esto ocurre porque la medida de la hipotenusa de un triángulo siempre es menor a la suma de los dos catetos, pero como en este caso hay que sumarle la cantidad de nodos recorridos hasta el momento pues la suma de ambas siempre es mayor a el cálculo de la hipotenusa desde una casilla anterior por lo que expande una cantidad de nodos bastante mayor a la función de manhattan. Por ello consideramos que en el algoritmo A-estrella es mejor emplear la función de manhattan ya que a todos los movimientos valer la misma cantidad no tenemos ese problema.

Por otra parte, mediante el uso del algoritmo greedy ocurre algo similar, la diferencia es que mediante greedy la diferencia de nodos generados usando una función heurística u otra no es tan notable, ambas funciones generan aproximadamente la misma cantidad de nodos, la euclídea genera aproximadamente un 5-10% más que la de manhattan mientras que con el algoritmo A-estrella hemos comprobado que a veces genera hasta más de tres veces más nodos.

5. Evaluación Experimental

Tabla 1 (sin obstáculos)

Mapa	Solución	Algoritmo	Función Heurística	Tiempo CPU	Lista Abierta	Lista Cerrada
10 x 10	17	A-Estrella	D. Manhattan	0.000441	36	18
10 x 10	17	Greedy	D. Euclídea	0.000404	52	18
10 x 10	17	Greedy	D. Manhattan	0.000368	36	18
50 x 50	97	A-Estrella	D. Manhattan	0.006347	196	98
50 x 50	97	Greedy	D. Euclídea	0.006996	292	98
50 x 50	97	Greedy	D. Manhattan	0.00616	196	98
100 x 100	197	A-Estrella	D. Manhattan	0.025561	396	198
100 x 100	197	Greedy	D. Euclídea	0.024816	592	198
100 x 100	197	Greedy	D. Manhattan	0.023602	396	198
200 x 200	397	A-Estrella	D. Manhattan	0.098304	796	398
200 x 200	397	Greedy	D. Euclídea	0.09925	1192	398
200 x 200	397	Greedy	D. Manhattan	0.095259	796	398
500 x 500	997	A-Estrella	D. Manhattan	1.24798	1996	998
500 x 500	997	Greedy	D. Euclídea	1.27992	2992	998
500 x 500	997	Greedy	D. Manhattan	1.25768	1996	998
1000 x 1000	1997	A-Estrella	D. Manhattan	31.9745	3996	1998
1000 x 1000	1997	Greedy	D. Euclídea	31.9366	5992	1998
1000 x 1000	1997	Greedy	D. Manhattan	32.1369	3996	1998

Como se puede observar en la tabla 1, hemos omitido los cálculos del algoritmo A-estrella con función euclidiana ya que cuanto menor número de obstáculos tenga el mapa más nodos genera y mayor es el tiempo de cómputo. Por lo tanto, consideramos que no proporciona una solución rentable para un mapa vacío o con poca cantidad de obstáculos.

Además, se puede apreciar que para poca cantidad de obstáculos el algoritmo greedy con función de heurística de manhattan es tan válido como el de A-estrella. Como se puede observar, nuestro entorno es capaz de soportar mapas superiores a 1000 x 1000 aunque su tiempo de cómputo crece exponencialmente al tamaño del mapa.

Tabla 2, tamaño muy pequeño (10 x 10)

Mapa	Obstáculos	Solución	Algoritmo	Función Heurística	Tiempo CPU	Lista Abierta	Lista Cerrada
10 x 10	25%	21	A-Estrella	D. Euclídea	0.00206	156	119
10 x 10	25%	21	A-Estrella	D. Manhattan	0.000701	52	38
10 x 10	25%	21	Greedy	D. Euclídea	0.00041	49	26
10 x 10	25%	23	Greedy	D. Manhattan	0.000471	40	23
10 x 10	50%	N.A	A-Estrella	D. Euclídea	0.000724	43	43
10 x 10	50%	N.A	A-Estrella	D. Manhattan	0.000568	43	43
10 x 10	50%	N.A	Greedy	D. Euclídea	0.000541	43	43
10 x 10	50%	N.A	Greedy	D. Manhattan	0.000372	43	43
10 x 10	80%	N.A	A-Estrella	D. Euclídea	0.000475	1	1
10 x 10	80%	N.A	A-Estrella	D. Manhattan	0.000411	1	1
10 x 10	80%	N.A	Greedy	D. Euclídea	0.000357	1	1
10 x 10	80%	N.A	Greedy	D. Manhattan	0.000754	1	1

Para mapas de tamaño 10 x 10 consideramos que el algoritmo greedy con una cantidad de obstáculos del 25% o menor es tanto o más efectivo que el algoritmo A-estrella. Además, como se puede observar la función heurística de manhattan genera menos nodos que la función heurística euclídea por lo que recomendamos el uso de la función de manhattan para cualquier cantidad de obstáculos.

Además, como se puede ver en la tabla, mapas con más del 50% de obstáculos no encuentran solución (N.A) debido a que no existe ningún camino posible para llegar desde la casilla de salida a la casilla de final.

Tabla 3, tamaño pequeño (50 x 50)

Mapa	Obstáculos	Solución	Algoritmo	Función Heurística	Tiempo CPU	Lista Abierta	Lista Cerrada
50 x 50	25%	97	A-Estrella	D. Euclídea	0.03451	998	424
50 x 50	25%	97	A-Estrella	D. Manhattan	0.011373	313	173
50 x 50	25%	105	Greedy	D. Euclídea	0.007417	237	113
50 x 50	25%	115	Greedy	D. Manhattan	0.007577	225	123
50 x 50	50%	N.A	A-Estrella	D. Euclídea	0.000724	6	6
50 x 50	50%	N.A	A-Estrella	D. Manhattan	0.000568	6	6
50 x 50	50%	N.A	Greedy	D. Euclídea	0.000541	6	6
50 x 50	50%	N.A	Greedy	D. Manhattan	0.000372	6	6
50 x 50	80%	N.A	A-Estrella	D. Euclídea	0.000475	1	1
50 x 50	80%	N.A	A-Estrella	D. Manhattan	0.000411	1	1
50 x 50	80%	N.A	Greedy	D. Euclídea	0.000357	1	1
50 x 50	80%	N.A	Greedy	D. Manhattan	0.000754	1	1

Para mapas de tamaño pequeño, es decir, 50 x 50, consideramos que con una cantidad de obstáculos del 25%, el algoritmo A-estrella lo consideramos mejor, ya que hay una notable diferencia entre la longitud del camino generado por A-estrella y por greedy. Aun así, se puede observar que la función heurística de manhattan genera menos nodos que la función heurística euclídea por lo que recomendamos el uso de la función de manhattan para cualquier cantidad de obstáculos en este tipo de mapa.

Además, como se puede ver en la tabla, mapas con más del 50% de obstáculos no encuentran solución (N.A) debido a que no existe ningún camino posible para llegar desde la casilla de salida a la casilla de final.

Tabla 4, tamaño mediano (100 x 100)

Mapa	Obstáculos	Solución	Algoritmo	Función Heurística	Tiempo CPU	Lista Abierta	Lista Cerrada
100 x 100	25%	197	A-Estrella	D. Euclídea	2.241511	26744	15021
100 x 100	25%	197	A-Estrella	D. Manhattan	0.233834	2843	1683
100 x 100	25%	223	Greedy	D. Euclídea	0.033468	518	262
100 x 100	25%	267	Greedy	D. Manhattan	0.04652	604	338
100 x 100	50%	N.A	A-Estrella	D. Euclídea	0.000724	1	1
100 x 100	50%	N.A	A-Estrella	D. Manhattan	0.000568	1	1
100 x 100	50%	N.A	Greedy	D. Euclídea	0.000541	1	1
100 x 100	50%	N.A	Greedy	D. Manhattan	0.000372	1	1
100 x 100	80%	N.A	A-Estrella	D. Euclídea	0.000475	1	1
100 x 100	80%	N.A	A-Estrella	D. Manhattan	0.000411	1	1
100 x 100	80%	N.A	Greedy	D. Euclídea	0.000357	1	1
100 x 100	80%	N.A	Greedy	D. Manhattan	0.000339	1	1

Para mapas de tamaño mediano, es decir, 100 x 100, consideramos que con una cantidad de obstáculos del 25%, el algoritmo A-estrella lo consideramos mejor, ya que hay una notable diferencia entre la longitud del camino generado por A-estrella y por greedy. Aun así, se puede observar que la función heurística de manhattan genera menos nodos que la función heurística euclídea por lo que recomendamos el uso de la función de manhattan para cualquier cantidad de obstáculos en este tipo de mapa.

Además, como se puede ver en la tabla, mapas con más del 50% de obstáculos no encuentran solución (N.A) debido a que no existe ningún camino posible para llegar desde la casilla de salida a la casilla de final.

Tabla 5, tamaño grande (200 x 200)

Mapa	Obstáculos	Solución	Algoritmo	Función Heurística	Tiempo CPU	Lista Abierta	Lista Cerrada
200 x 200	25%	N.A	A-Estrella	D. Euclídea	N.A	N.A	N.A
200 x 200	25%	397	A-Estrella	D. Manhattan	0.494119	3000	1704
200 x 200	25%	417	Greedy	D. Euclídea	0.124747	937	461
200 x 200	25%	481	Greedy	D. Manhattan	0.213131	1406	781
200 x 200	50%	N.A	A-Estrella	D. Euclídea	0.000724	1	1
200 x 200	50%	N.A	A-Estrella	D. Manhattan	0.000568	1	1
200 x 200	50%	N.A	Greedy	D. Euclídea	0.000541	1	1
200 x 200	50%	N.A	Greedy	D. Manhattan	0.000372	1	1
200 x 200	80%	N.A	A-Estrella	D. Euclídea	0.000475	1	1
200 x 200	80%	N.A	A-Estrella	D. Manhattan	0.000411	1	1
200 x 200	80%	N.A	Greedy	D. Euclídea	0.000357	1	1
200 x 200	80%	N.A	Greedy	D. Manhattan	0.000339	1	1

Para mapas de tamaño grande, es decir, 200 x 200, o tamaños superiores consideramos que con una cantidad de obstáculos del 25%, el algoritmo A-estrella lo consideramos mejor empleando manhattan, ya que hay una notable diferencia entre la longitud del camino generado por A-estrella y por greedy. Aun así, se puede observar que la función heurística de manhattan genera menos nodos que la función heurística euclídea por lo que recomendamos el uso de la función de manhattan para cualquier cantidad de obstáculos en este tipo de mapa. Bajo ningún concepto se recomienda el uso de función euclídea combinada con algoritmo A-estrella en este tipo de mapas ya que el tiempo de cómputo se dispara excesivamente y aportará un camino equivalente al generado por la función heurística de manhattan.

Además, como se puede ver en la tabla, mapas con más del 50% de obstáculos no encuentran solución (N.A) debido a que no existe ningún camino posible para llegar desde la casilla de salida a la casilla de final.

6. Conclusión

En las tablas de la Evaluación Experimental podemos apreciar que la cantidad de nodos recorridos es directamente proporcional con las dimensiones del mapa, al igual que con la cantidad de obstáculos que se generan, a mayor cantidad de obstáculos mayor es el camino recorrido.

Comparando las funciones heurísticas, podemos llegar a la conclusión de que ambas generan un camino óptimo, puesto que realizan un recorrido con la misma distancia. Sin embargo, se puede apreciar que la función heurística Euclídea genera muchos más nodos que la de Manhattan.

Además, se puede ver que los recorridos generados por A* con el uso de la función heurística, generan un camino lo más diagonal posible, de lo contrario a la de manhattan, que genera caminos rectilíneos moviéndose el máximo posible por las filas y/o columnas.

7. Bibliografía

- Euclidean: https://en.wikipedia.org/wiki/Euclidean
- Manahattan: https://es.wikipedia.org/wiki/Geometr%C3%ADa_del_taxista
- Heuristica: https://es.wikipedia.org/wiki/Heur%C3%ADstica_admisible
- Algoritmo A*: https://es.wikipedia.org/wiki/Algoritmo_de_b%C3%BAsqueda_A*
- Algoritmo Greedy: https://es.wikipedia.org/wiki/Algoritmo_voraz