

PROYECTO 1: ESTRATEGIAS DE BÚSQUEDA



Grupo:

Diego Vázquez Campos

Raúl Rodríguez Torres

José Ramón Rodríguez Hernández

ÍNDICE

1.	Descripción del entorno, simulación e implementación.....	0
1.1.	Descripción del problema	2
1.2.	Consideraciones de moralidad.....	2
1.3.	Descripción del sistema basado en agente (función del agente).....	3
1.4.	Descripción del informe de simulación.	4
2.	Funciones heurísticas contempladas	6
2.1.	Función Euclídea	6
2.2.	Función Manhattan (Lineal)	6
3.	Estructura de datos para la contención de nodos en el árbol de búsqueda.	7
4.	Evaluación	9
4.1.	Estudio experimental, tabla de resultados	9
4.2.	Visualización de los casos.....	9
4.3.	Conclusiones.....	11

1. Descripción del entorno, simulación e implementación.

1.1. Descripción del problema

Se plantea la situación en la que un sistema automático sea capaz de determinar el recorrido mínimo (camino óptimo) de un punto origen a un destino concreto mediante la implementación de algoritmos de búsqueda, así como la evaluación de diversas funciones heurísticas.

En principio, el programa debe contar con una interfaz gráfica que represente el escenario, con su punto de partida, su punto inicial, los elementos que intervienen y el recorrido óptimo que ha sido determinado.

Para ello, es importante percibir el entorno en parámetros $M \times N$, identificando la diferencia entre celdas libres y obstáculos. De esta manera, el coche podrá actuar en base a su norte, sur, este y oeste, para determinar cuáles son sus posibles movimientos.

El sistema automático estará asociado a un coche inteligente, que se enfrentará a un entorno con obstáculos que deberá evitar. Pero los obstáculos no son el único elemento al que el coche deberá enfrentarse. Éste deberá ser capaz de diferenciar entre obstáculos y personas, y su modo de actuar dependerá del tipo de elemento al que tenga delante.

La definición del comportamiento del coche para las personas describiría la siguiente respuesta: El coche debería incluir esas personas en su interior, y transportarlas consigo. Pero el coche tiene una capacidad, por lo cual, cuando el coche esté lleno, a partir de ese instante considerará a las personas como obstáculos, tratando de evitar pasar por encima de ellas.

En caso de que el coche deba actuar frente a un obstáculo, directamente debería ser capaz de determinar el recorrido a tomar esquivando los obstáculos.

1.2. Consideraciones de moralidad

El modus operandi para los casos donde la conducta actúa diferente para las componentes definidas como personas, y aunque en el programa la simulación está representada como se describía en la descripción del problema, en la práctica el coche considera multitudes de casos que queda implícito en su conducta.

La idea sería considerar un marco teórico-ético que deberá tener el programa para poder discernir entre dos opciones perjudiciales de la forma en que lo haría un conductor en persona.

Un ejemplo de esos casos que se deberían considerar es el de la toma de una decisión cuando el coche se vea en la obligación de tener que girar para no atropellar un peatón imprevisto a costa de sacrificar la vida de los pasajeros, o el de otros conductores por interferir en carriles adyacentes. Podrían fijarse unas normas en base a la información que disponga el coche para poder tomar la decisión:

- Disponer del conocimiento de la velocidad a la que va.
- Considerar la seguridad de la que dispone el piloto, como airbags. En ese caso se priorizaría la seguridad del peatón.
- Según el tipo de terreno por el que se circula (asfaltado o no asfaltado)
- Considerar si el peatón está cumpliendo con las normas de tráfico, en caso de estar incumpléndolas priorizaría la seguridad del piloto.
- Conocimiento de la condición piloto. Si el piloto o uno de los pasajeros resulta ser una mujer embarazada, debería priorizar la seguridad de sus pasajeros.

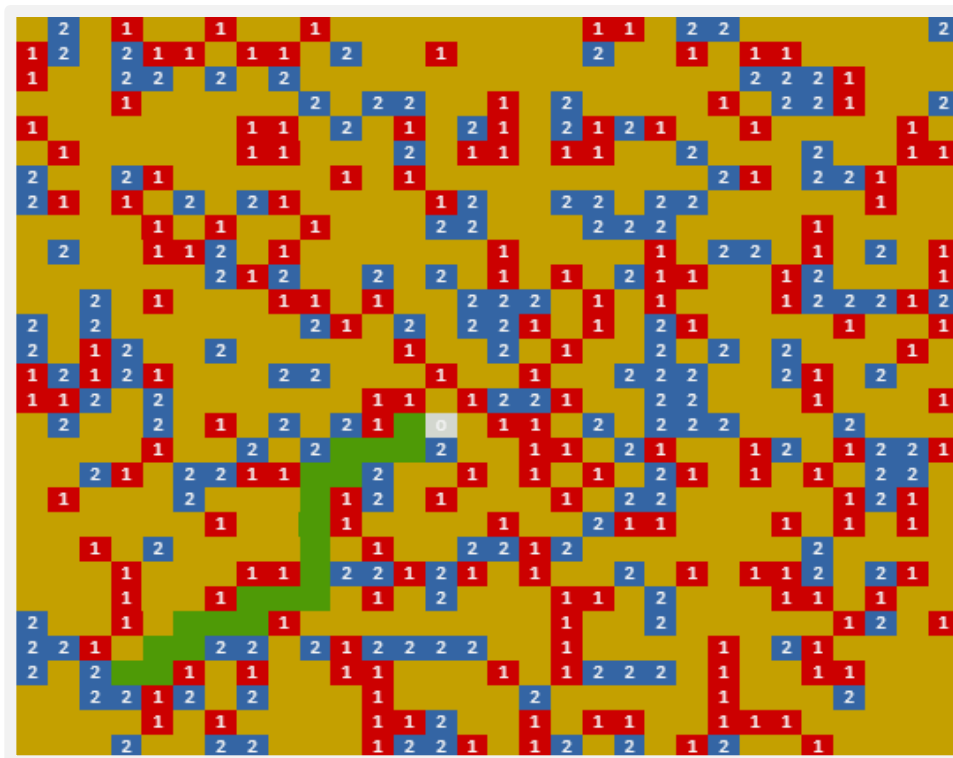
Además de lo antes mencionado, se intentaría minimizar el daño para alguno de los dos individuos.

1.3. Descripción del sistema basado en agente (función del agente)

Secuencia de percepciones	Acción
[N, No es obstáculo, Es mejor opción, no es callejón]	Moverse a Norte
[S, No es obstáculo, Es mejor opción, no es callejón]	Moverse a sur
[E, No es obstáculo, Es mejor opción, no es callejón]	Moverse a Este
[O, No es obstáculo, Es mejor opción, no es callejón]	Moverse a Oeste
[N, No es obstáculo No es mejor opción]	Mirar Sur, Este u Oeste
[S, No es obstáculo, No es mejor opción]	Mirar Norte, Este u Oeste
[E, No es obstáculo, No es mejor opción]	Mirar Sur, Norte u Oeste
[O, No es obstáculo, No es mejor opción]	Mirar Sur, Norte o Este
[N, Es obstáculo o es callejón]	Mirar Sur, Este u Oeste
[S, Es obstáculo o es callejón]	Mirar Norte, Este u Oeste
[E, Es obstáculo o es callejón]	Mirar Sur, Norte u Oeste
[O, Es obstáculo o es callejón]	Mirar Sur, Norte o Este
[N, No es obstáculo, Es mejor opción], [N, E y O, son obstáculos o callejones]	Marcar Norte como callejón y retroceder
[S, No es obstáculo, Es mejor opción], [S, E y O, son obstáculos o callejones]	Marcar Norte como callejón y retroceder
[E, No es obstáculo, Es mejor opción], [N, S y E, son obstáculos o callejones]	Marcar Norte como callejón y retroceder
[O, No es obstáculo, Es mejor opción], [N, S y O, son obstáculos o callejones]	Marcar Norte como callejón y retroceder
[N, Es persona, Es mejor opción, coche no lleno]	Moverse a Norte, suma persona a bordo
[S, Es persona, Es mejor opción, coche no lleno]	Moverse a sur, suma persona a bordo
[E, Es persona, Es mejor opción, coche no lleno]	Moverse a Este, suma persona a bordo
[O, Es persona, Es mejor opción, coche no lleno]	Moverse a Oeste, suma persona a bordo
[N, Es persona, Es mejor opción, coche lleno]	Mirar Sur, Este u Oeste
[S, Es persona, Es mejor opción, coche lleno]	Mirar Norte, Este u Oeste
[E, Es persona, Es mejor opción, coche lleno]	Mirar Sur, Norte u Oeste
[O, Es persona, Es mejor opción, coche lleno]	Mirar Sur, Norte o Este

La naturaleza del entorno de trabajo (Descripción REAS)				
Agente	M. de rendimient.	Entorno	Actuadores	Sensores
Coche autónomo	Seguridad	Carreteras	Dirección	Cámaras
	Rapidez	Tráfico	Acelerador	Sónar
	Legalidad	Peatones	Freno	Velocímetro
	Comodidad	Pasajeros	Bocina	GPS
	Eficiencia	Obstáculos	Señal	Tacómetro
			Visualizador	Control de aceler.
				Sens. Del motor
				Controlador E/S

1.4. Descripción del informe de simulación.



Ante la imposibilidad de recurrir a una interfaz gráfica cómoda para un usuario común, se decidió abordar la visualización de la resolución en la terminal. Y debido a ello, se dio mucha importancia a la estética de la presentación para que, a pesar de las limitaciones de la misma, el programa fuera capaz de presentar una visual agradable, y legible.

La apariencia que se eligió para cada elemento fue la de bloques rojos para obstáculos y azules para personas. A parte, se aprovecho el espacio para representar dichos elementos con el mismo número con el que el programa trabaja internamente. El bloque gris “o”, representa al coche, que a medida que avanza deja una estela verde, marcando así el recorrido mínimo.

Al inicio del programa pregunta qué heurística se desea utilizar y una vez acabado el recorrido, aparecen tres datos debajo del cuadro que ofrecen la información necesaria para determinar su eficiencia.

a) Menú inicial

```
-----
Bienvenido a la práctica de heurística
-----

Dispones de las siguientes opciones:
-----

1 - Casos Predefinidos
2 - Manual
3 - Fichero
0 - Exit
-----

Opcion: █
```

Ofrece la posibilidad de elegir entre generar elementos de forma aleatoria para tamaños de malla predefinidos. Introducir la cantidad de forma manual. O fijar coordenadas concretas para cada elemento a través de un fichero de texto.

b) Casos 1 y 2

Tamaños de la malla	Casos predefinidos
<pre> ----- Tipo de malla ----- 1 - 20 x 20 2 - 30 x 30 3 - 50 x 50 4 - 100 x 100 0 - Exit ----- Opcion: </pre>	<pre> ----- Porcentaje de obstaculos: 1 - 20% 2 - 30% 3 - 40% ----- 1 -¿Que tipo de calculo quieres usar? manhattan(0), euclidia(1): 1 </pre>
	<pre> ----- Introducción manual ----- Seleccione el origen: X: 0 Y: 0 Seleccione el destino: X: 10 Y: 10 Numero de obstaculos: 20 -¿Que tipo de calculo quieres usar? manhattan(0), euclidia(1): 1 </pre>

c) Introducción por fichero

Detecta automáticamente el fichero en la misma carpeta de construcción, el cual debe tener el nombre de "datos.txt"

Formato del fichero	
<pre> 1 60 60 17 2 1 2 1 3 14 5 2 4 12 6 2 5 15 15 2 6 1 1 1 7 10 10 2 8 11 11 2 9 14 14 2 10 15 15 2 11 16 14 2 12 17 13 2 13 10 12 2 14 9 12 2 15 8 12 2 16 5 4 1 17 30 30 1 </pre>	<p>La primera línea define el tamaño de la malla (60 x60) Seguido del número de coordenadas a leer en las siguientes líneas "17"</p> <p>En cada línea se define la coordenada como:</p> <ul style="list-style-type: none"> - Primera columna define la componente x. - La segunda columna define la componente y. - La tercera columna define el tipo de elemento: 2 para persona, 1 para obstáculo, etc

d) Resultado final

Al se mostrará la malla con el recorrido mínimo barrido en movimiento, y al acabar, se mostrarán los datos de eficiencia justo debajo.

```

-----
Nodos generados: 76
-----
Longitud del camino: 21
-----
Tiempo de calculo: 0.000708
-----
Pulse enter para continuar...

```

2. Funciones heurísticas contempladas

Por definición, la función heurística se define como la estimación utilizada para la obtención de un resultado que utilizaremos para el cálculo del algoritmo A* siendo su función:

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

Donde, $g(n)$ define la distancia recorrida hasta el momento/nodo "n" (dicho de otro modo, es el coste del camino desde el punto inicial/ origen, hasta el nodo n) y $h(n)$ define el resultado de hacer la estimación para la distancia que falta por recorrer en el momento/nodo "n" (o dicho de otro modo, es el coste más barato desde dicho nodo n hasta la meta/nodo final).

2.1. Función Euclídea

Se define como la raíz cuadrada de la suma de los cuadrados de la diferencia de la componente "x" de la coordenada del nodo objetivo menos la componente "x" de la coordenada del nodo final, y por el otro lado, la componente "y" de la coordenada del nodo actual, menos la componente "y" de la coordenada del nodo final.

$$h_1(n) = \sqrt{(x_f - x_0)^2 + (y_f - y_0)^2}$$

```
float car_t::euclidiana(int x, int y)
{
    return (sqrt(((get_x_destiny() - x)*(get_x_destiny() - x))+((get_y_destiny() - y)*(get_y_destiny() - y))));
}
```

Tal estimación aporta como resultado en nuestro algoritmo un movimiento que prioriza, ante todo, un acercamiento en diagonal.

2.2. Función Manhattan (Lineal)

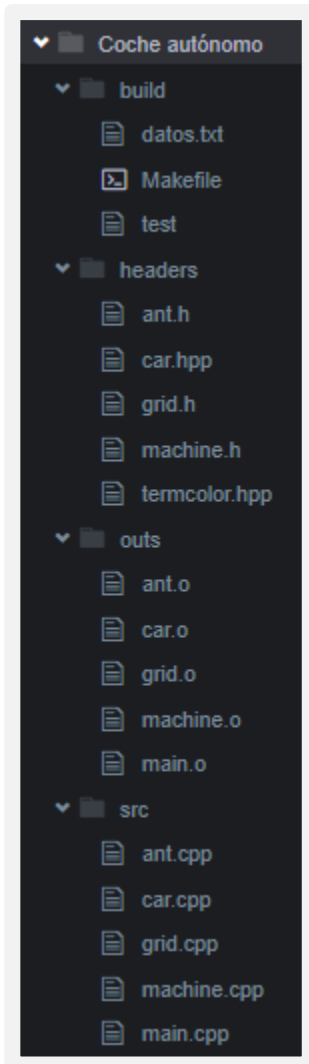
Se define como la suma de los valores absolutos al resultado de hacer la diferencia de la componente "x" de la coordenada del nodo objetivo menos la componente "x" de la coordenada del nodo final, y por el otro lado, la componente "y" de la coordenada del nodo actual, menos la componente "y" de la coordenada del nodo final.

$$h_2(n) = |x_f - x_0| + |y_f - y_0|$$

```
float car_t::manhattan(int x, int y)
{
    return ((abs(get_x_destiny() - x)) + (abs(get_y_destiny() - y)));
}
```

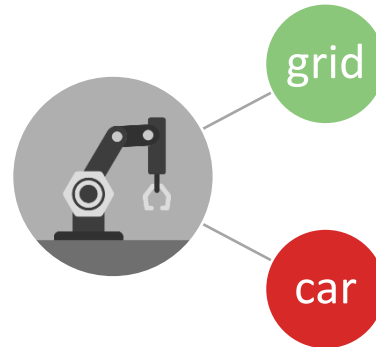
Tal estimación aporta como resultado en nuestro algoritmo un resultado próximo a un movimiento en "L"

3. Estructura de datos para la contención de nodos en el árbol de búsqueda.



El programa consta de 3 clases principales, siendo una de ellas, una clase con herencia de una cuarta llamada ant.cpp.

El funcionamiento se refleja en el siguiente diagrama.



Lo que quiere representar es la importancia de la clase **machine**, cuyo fin es el de hacer una gestión como mediador entre la rejilla (la clase **grid**) y el coche (la clase **car**). De esa forma hacemos posible, la formación del programa con un orden, velando por la eficiencia, y evitando así, la asignación de métodos de forma caótica.

A continuación, vamos a hacer un resumen de cada una de esas clases, obviando el funcionamiento del resto de clases de las que depende.

a) Machine

```
class machine_t
{
private:
    car_t coche_;
    grid_t rejilla_;

public:
    machine_t(car_t car, grid_t rejilla);
    ~machine_t();

    void start(void);

    void random_obstacles(int cantidad);
    void random_obstacles(float porcentaje);
    void manual_obstacles(int opcion);

    void eyecar();

    void write(void);
};
```

Machine es una clase formada por dos atributos: El coche y la rejilla por la que se moverá el coche.

Es el encargado de modificar la rejilla (generando sus obstáculos y personas) y a su vez de dar comienzo al movimiento del coche a través de su método *start()*.

Como el coche solo debe preocuparse de su posición actual, de conocer su destino y de moverse, machine es el encargado de ofrecerle toda la información que necesite saber.

b) Grid

```
class grid_t
{
private:
    std::vector<std::vector<int>> > grid_;
    int height_;
    int width_;

public:
    //constructores
    grid_t();
    grid_t(int x, int y);
    grid_t(int x, int y, int colour);
    grid_t(const grid_t &grid);
    virtual ~grid_t();

    //getters
    int get_height(void) const;
    int get_width(void) const;
    int get(int x, int y) const;

    //setters
    void set_height(int height);
    void set_width(int width);
    void set(int x, int y, int colour);
    void resize(int height, int width);
    void resize(int height, int width, int colour);
};
```

La rejilla es la matriz que representa nuestro escenario. Cada posición en la misma contiene un número, que es el que define qué es lo que se encuentra en esa posición

- 0) Para huecos
- 1) Para obstáculos
- 2) Para personas
- 3) Para la representación de la estela.
- 6) Callejón sin salida, o casilla que conduce a un callejón sin salida.

c) Car

```
class car_t: public ant_t
{
private:
    std::vector<int> destiny_;
    std::vector<std::vector<int>> > possible_move_;
    std::vector<int> status_;
    std::vector<std::vector<int>> > recorrido_;
    bool callejon_;
    std::vector<int> callejon_v_;
    int contador_estrella;
    int nodos_generados;

public:
    car_t();
    car_t(int x, int y);
    ~car_t();

    void insertar_recorrido(int x, int y);
    void extraer_recorrido();
    void set_contador_estrella(int x);
    void set_nodos_generados();
    int get_nodos_generados();

    // setters
    void set_destiny(int x, int y);
    void set_status(std::vector<int> v);

    // getters
    int get_x_destiny() const;
    int get_y_destiny() const;
    int get_size_recorrido() const;
    std::vector<int> get_destiny() const;
    int get_recorrido_x(int i) const;
    int get_recorrido_y(int i) const;
    int get_callejon_v_x();
    int get_callejon_v_y();

    // Poda
    void pruning_neighbours(void);
    void set_callejon_false();
    bool get_callejon();
    void set_callejon_true();

    //Security
    std::vector<std::vector<int>> > check_compass(int width, int height);
    std::vector<std::vector<int>> > check_ut(int width, int height, std::vector<std::vector<int>> &v);
    bool check(int width, int height, std::vector<int> v);
    bool check_destiny();

    //Auxiliary
    std::vector<int> north();
    std::vector<int> south();
    std::vector<int> west();
    std::vector<int> east();

    //Principal
    int move(int width, int height, int opt);
    float euclidiana(int x, int y);
    float manhattan(int x, int y);

    void print_data_test();
};
```

En dicha clase es donde reside la mayor complejidad del programa. No solo por la herencia de la clase ant, que es quien abastece de mecanismos para trabajar con coordenadas y movimientos. Sino que, además, es la clase encargada de **evaluar los nodos adyacentes** (mediante el método `check_compass()`), realizar el **algoritmo A***, estimando la distancia con las **heurísticas**, para ir guardando los nodos determinados como mejores opciones en un array "recorrido_" y así, almacenar la sucesión de nodos que forman el **camino mínimo**.

4. Evaluación

4.1. Estudio experimental, tabla de resultados

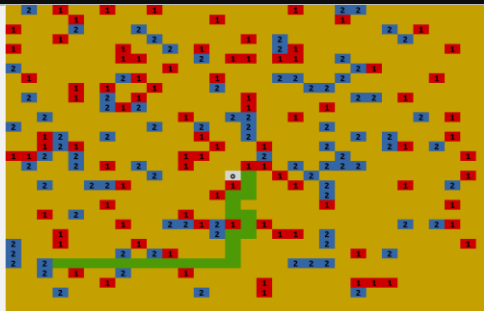
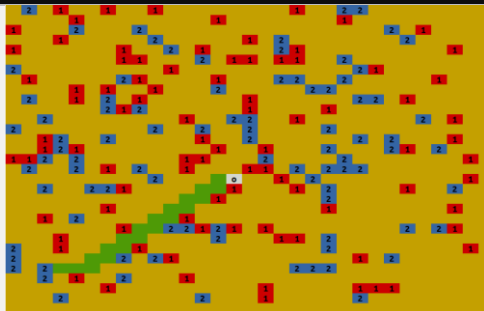
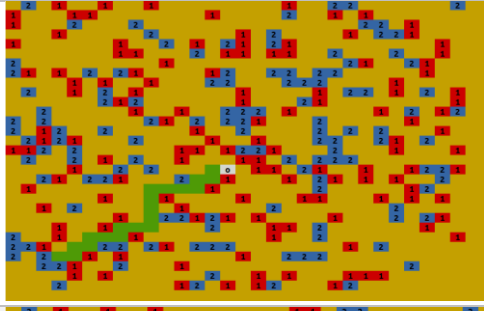
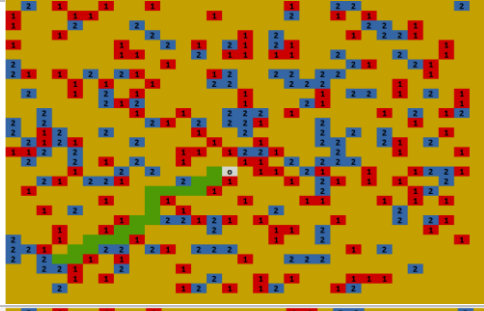
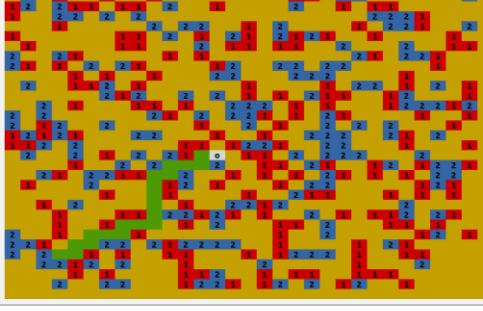
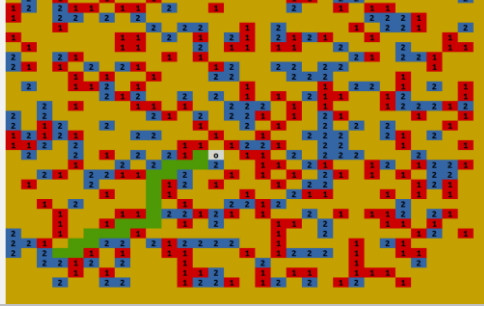
		Longitud		CPU		Nodos Generados	
Dimensión	Obstcls.	Manhattan	Euclidiana	Manhattan	Euclidiana	Manhattan	Euclidiana
20x20	20%	20	20	0,832 ms	0,863 ms	66	66
	30%	20	20	0,84 ms	1,039 ms	62	62
	40%	16	16	0,76 ms	0,765 ms	55	47
30x30	20%	26	22	3,029 ms	2,445 ms	93	68
	30%	22	22	2,923 ms	1,431 ms	81	65
	40%	22	22	2,551 ms	2,282 ms	63	58
50x50	20%	19	23	2,231 ms	1,642 ms	69	77
	30%	19	23	2,152 ms	2,43 ms	67	79
	40%	19	23	2,389 ms	2,988 ms	69	81
100x100	20%	35	33	3,546 ms	3,248 ms	123	111
	30%	33	33	3,018 ms	3,193 ms	103	105
	40%	37	35	3,764 ms	3,289 ms	113	101

4.2. Visualización de los casos

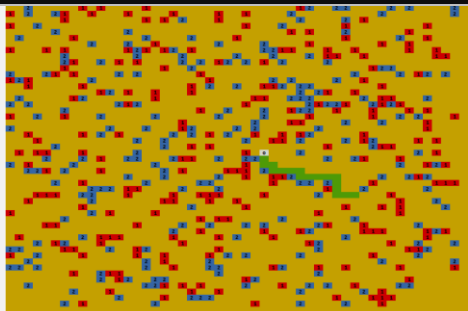
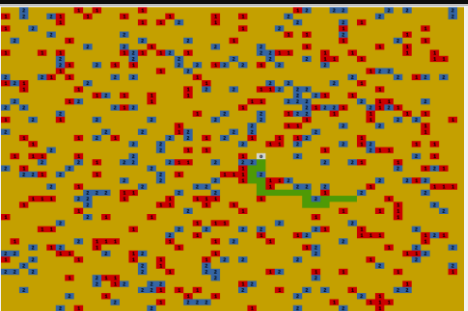
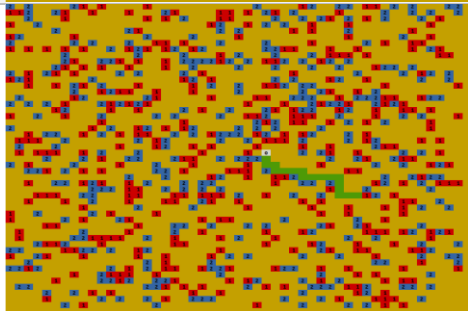
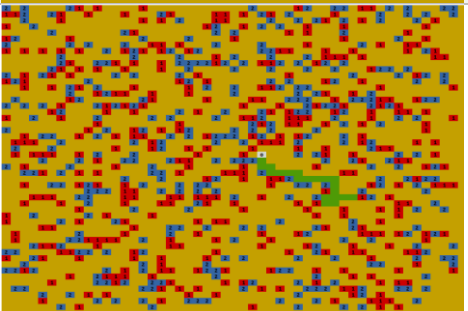
a) 20 x 20

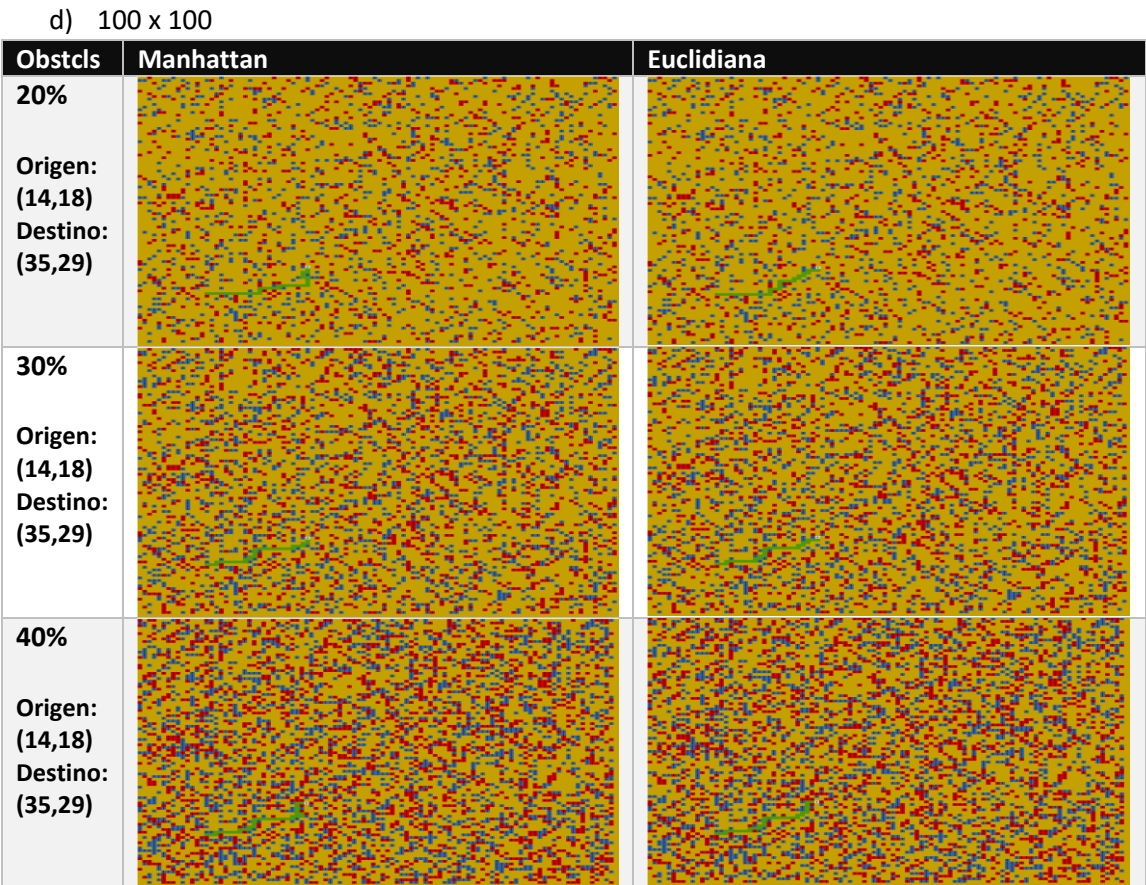
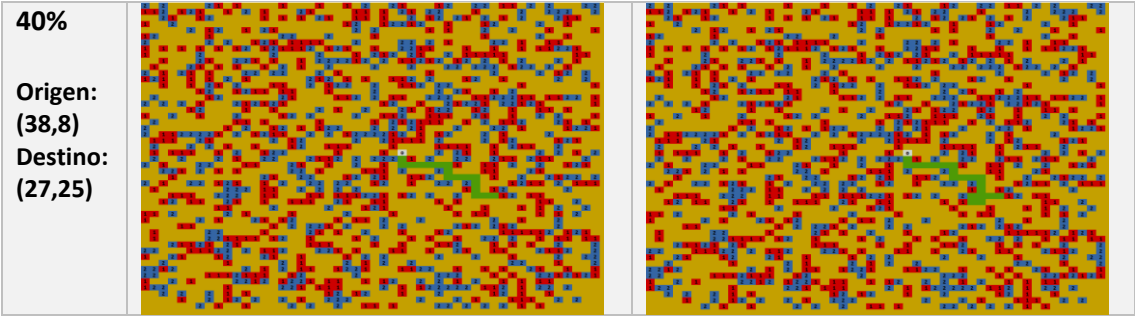
Obstcls	Manhattan	Euclidiana
20% Origen: (0,0) Destino: (12,7)		
30% Origen: (0,0) Destino: (12,7)		
40% Origen: (2,1) Destino: (10,6)		

b) 30 x 30

Obstcls	Manhattan	Euclidiana
20% Origen: (3,3) Destino: (14,3)		
30% Origen: (3,3) Destino: (14,3)		
40% Origen: (3,3) Destino: (14,3)		

c) 50 x 50

Obstcls	Manhattan	Euclidiana
20% Origen: (38,8) Destino: (27,25)		
30% Origen: (38,8) Destino: (27,25)		



4.3. Conclusiones

Como podemos observar en el estudio experimental realizado, los resultados son claramente favorecedores hacia la función heurística Euclidiana, ya que en la práctica ofrece un tiempo menor y la cantidad de costo consumido en los desplazamientos de nodo a nodo también ofrece una cifra, aunque cercana, debajo de la que resulta de aplicar la heurística Manhattan.