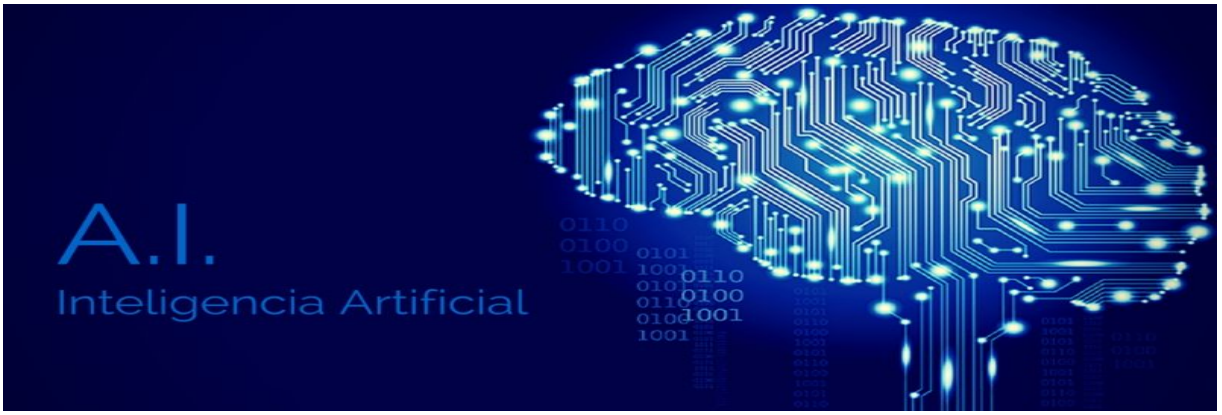
Informe:

Inteligencia Artificial

Práctica de Búsqueda

Renisha BharatKumar Lachhani Punjabi
alu0101028026@ull.edu.es
Sergio Delgado López
alu0100893601@ull.edu.es



Índice:

|1. Introducción.

Dentro de este informe vamos a tratar con los dos objetivos principales de la práctica: definir cómo sería un coche autónomo comercial en un futuro no muy lejano, atendiendo a su comportamiento y toma de decisiones, y por otro lado, describir de manera breve el software usado para resolver el problema de búsqueda.

El problema de búsqueda se basa en un coche inteligente que tiene que recoger *x* número de pasajeros y llevarlos hasta un punto final *P*, evitando los obstáculos y límites del mapa.

|2. Descripción del coche autónomo y entorno.

2.1. Tabla.

En primer lugar debemos definir lo que es un agente inteligente. Un agente es cualquier cosa capaz de percibir su medioambiente con la ayuda de sensores y actuar en ese medio utilizando actuadores.

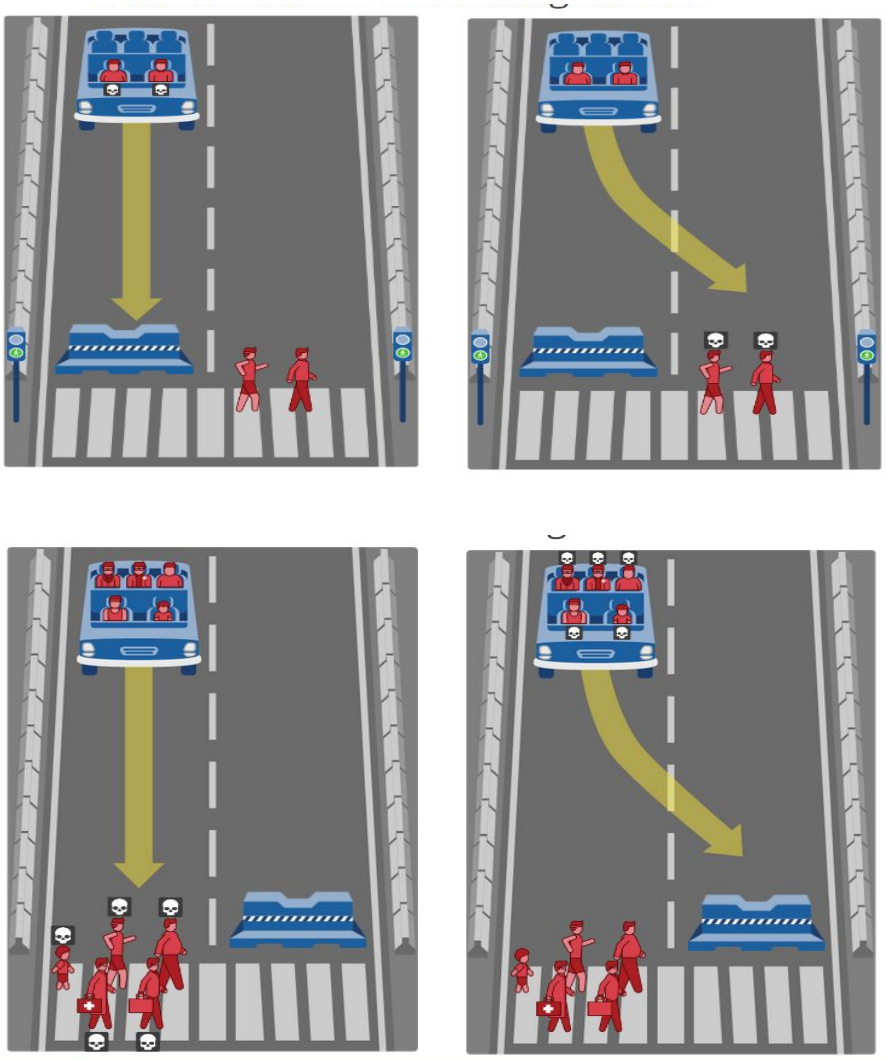
Supongamos un modelo de coche autónomo, y definimos las distintas PERCEPCIONES, ACCIONES, OBJETIVOS y el ENTORNO:

Percepciones	**Acciones**	**Objetivos**	**Entorno**
1. Cámara: ver señales, persona, obstáculos (3D para medir la profundidad). 2. Sensor de lluvia 3. Sensor de luminosidad 4. Sensor de proximidad 5. Sensor de golpe 6. Sensor de temperatura 7. Sensor de seguimiento 8. Acústicos 9. Ultrasonido 10. Geolocalización 11. Radio	1. Acelerar 2. Frenar/Detener 3. Cambiar Dirección (intermitentes) 4. Encender luces 5. Apagar luces 6. Alertas Acústicas 7. Alertas de Emergencia 8. Asistente Virtual : informar y proporcionar herramientas al usuario	1. Evitar un accidente 2. Eficiencia (combustible, max/min beneficio) 3. Ecológico	1. Peatones 2. Señales 3. Obstáculos

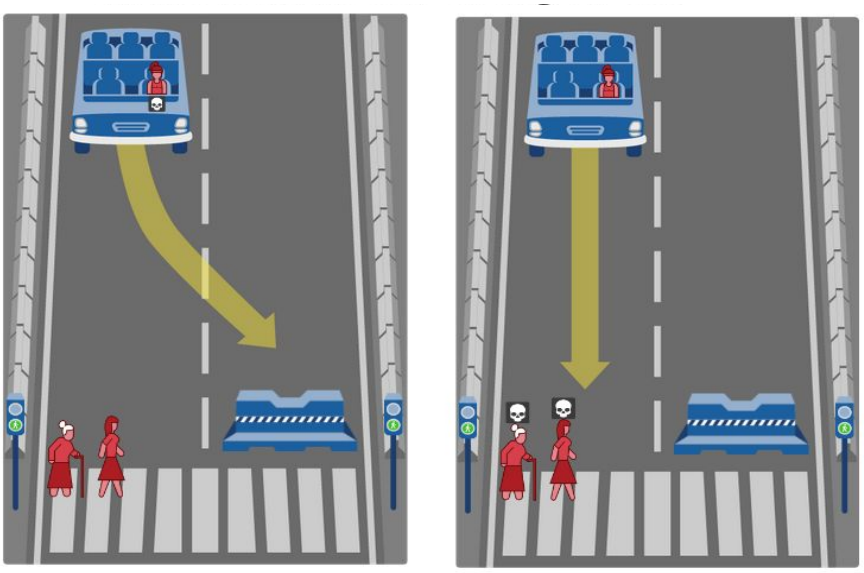
2.2. Tabla de decisiones.

- **Caso A:**
  Elegir entre un coche con personas chocar contra personas cruzando o obstáculo.

  El coche con fallo de frenos se choque contra el obstáculo, así sólo morirían las personas que estén dentro del coche pero al menos el coche pararía para no seguir atropellando a más personas. La opción de que el coche se choque contra las dos personas no tendría sentido ya que el coche no pararía después de chocar contra ellos y seguiría y podría chocar contra más personas y matar a las que están dentro.



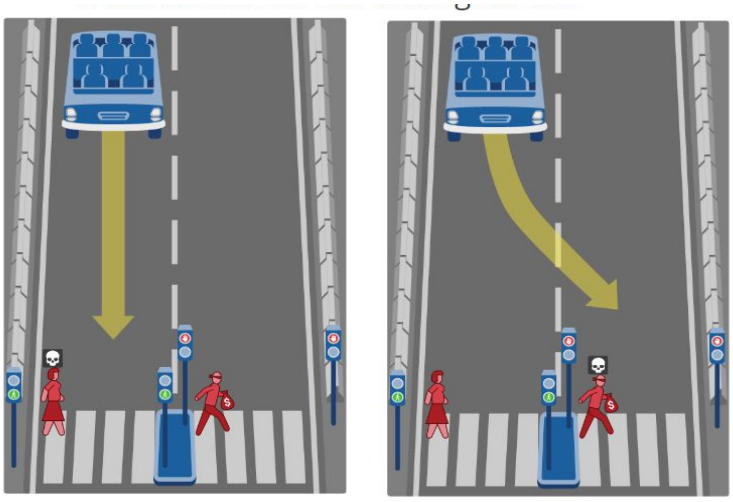
En este caso no debe importar la señal porque que las personas estén cruzando en rojo o verde el problema seguiría siendo que después de chocar contra esas personas el coche no pararía y seguirá chocando con más personas posiblemente, pués sería mejor que choque contra el obstáculo.



En éste caso tampoco importaría el número de personas que estén fuera o dentro del coche si el coche no se va a desacelerar o frenar.

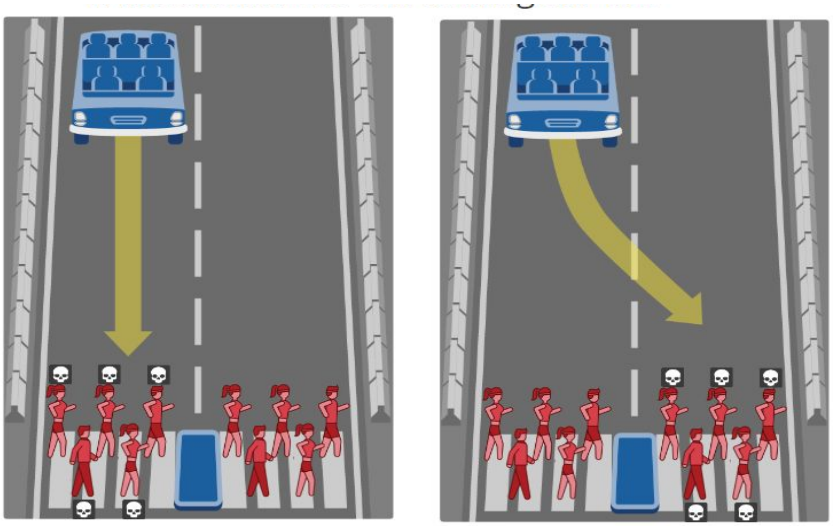
- **Caso B:**
  Elegir que un coche vacío se atropelle entre una persona que cruza con señal verde y otro con señal rojo.

En este caso sí habría que elegir, debería chocar contra la persona que cruza con señal rojo ya que esa persona no debería de estar cruzando en ese momento.



- **Caso C:**
  Elegir que un coche se choque contra el mismo número de personas por los dos caminos.

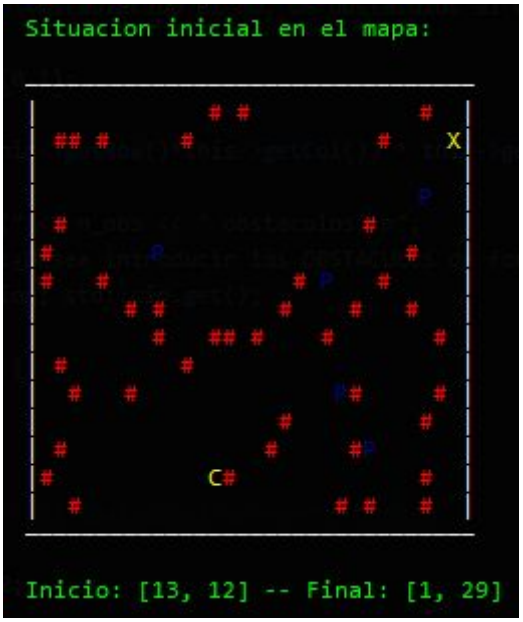
  Si por los dos caminos se encuentra con el mismo número de personas pues sería mejor que siga recto.



- **En los casos B y C la mejor opción para el coche sería no elegir entre esos dos caminos sino detectar por los lados de derecha e izquierda para detectar si puede chocar contra un obstáculo o girar por esos lados sin chocar contra personas u otros vehículos envés de chocar contra las personas en los dos caminos. En el caso de que no puede girar por la derecha o izquierda, debe escoger la solución indicada para cada caso B y C.**

2.3. Entorno de simulación.

En nuestro entorno de simulación podemos encontrarnos con tres objetos: obstáculos, personas y un coche. Este programa simula un tablero (matriz) con dimensiones *m x n* con los bordes destacados y vacía por dentro para meter nuestros objetos. Tenemos una posición inicial para el coche que se muestra con la letra ‘C’ y una posición final demostrada por el símbolo ‘X’. El coche mueve por el tablero evitando los obstáculos demostrados por el símbolo ‘#’ recogiendo todas las personas demostrados por el símbolo ‘P’ que se encuentre por el camino siguiendo la función heurística aplicada.



|3. Descripción del software.

Para implementar el software necesario que resuelve el problema de búsqueda hemos decidido programa en el conocido lenguaje C++, por la principal razón de que es el lenguaje que más dominamos tras numerosos proyectos realizados.

3.1. Estructura.

3.1.1. Tablero.

La clase *Tablero* implementada en el fichero *Tablero.h* será la más extensa de nuestro código, ya que en ella será donde realicemos la mayoría de las operaciones. Dichas operaciones implican la impresión del mapa, colocación de los objetos del entorno (personas, obstáculos, límites), etc.

```
class Tablero {

private:

  std::vector<std::vector<char>> mapa_;
  Coche car_;
  std::vector<pasajeros> pasajeros_;
```

En esta tenemos atributos como el mapa que es una matriz de caracteres, un propio objeto Coche, un vector de pasajeros por si en el futuro se quisiera implementar algo relacionado con los pasajeros, y por último también tenemos la posición tanto inicial como final del coche.

Cabe decir que en toda la implementación usamos un *struct* para almacenar las posiciones i y j.

Atendiendo ahora a la funciones de los elementos del entorno, como se ve en la figura siguiente, tanto la colocación del coche, de los obstáculos o de los pasajeros, puede realizarse de forma automática y aleatoria, o manual.

```
void colocarCoche ();
void rellenarObstaculos ();
void rellenarPersonas ();
```

3.1.2. Coche.

Dentro de esta clase es donde reside todo lo relacionado con el coche autónomo.

```
class Coche {

private:

  char simbolo_coche_;
  int i_;
  int j_;
  int personas_recogidas_;

  posicion inicio_;
  posicion fin_;

  Camino path_;
```

Como atributos, esta clase tiene el símbolo con el que se representará el coche dentro del tablero, y su posición actual, también dentro del tablero. Por otro lado también tiene el número de personas que recogerá mientras realiza el camino, al igual que el *tablero* el coche también almacena su posición inicial y final.

Por último y no menos importante, el coche almacena un objeto Camino, que es el objeto con la clase encargada de calcular todo el camino con las funciones de búsqueda implementadas.

El resto de la clase, se podría considerar una clase almacén ya que esta solo recurrirá a las otras clases para hacer los cálculos. Al igual que todas las clases, ésta tiene todos los métodos necesarios de escritura y lectura.

3.1.3. *Struct* de pasajeros.

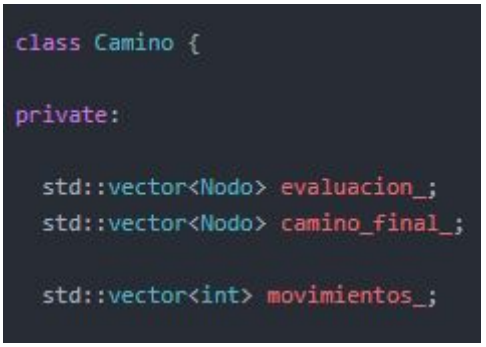
```
struct pasajeros{

  int posi;
  int posj;
  bool recogido;

};
```

Este tipo de dato *struct* se encuentra dentro de la clase tablero con el fin de tener almacenadas las posiciones de los pasajeros que el coche tendrá que recoger de forma autónoma.

Estos datos podrán sernos útiles a la hora de calcular los algoritmos de movimientos.

Como se vio anteriormente, dentro de *Tablero* tenemos un tipo de dato vector, que tendrá como tamaño a tantos pasajeros como haya en el problema, almacenando la posición de cada uno y su estado.

3.1.4. Camino.

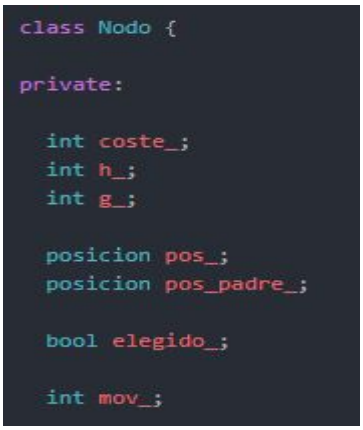


Esta clase compone el grueso de todo el código fuente de la práctica, ya que contiene todos los cálculos necesarios para llevar a cabo la búsqueda de la solución al problema con el algoritmo A*.

Como se puede observar en la figura, los atributos que tiene esta clase son tres vectores, dos de ellos son de objetos Nodos, de los cuales hablaremos a continuación, y el otro es un vector de enteros que servirá para almacenar el recorrido que ha de realizar el coche en el mapa.

El vector denominado *evaluacion_* es el más importante, puesto que es el que va a trabajar con todos los nodos generados y evaluados para concluir en el camino a seguir por el coche, usando el ya comentado A*.

3.1.5. Nodo.



Esta clase se trata de un clase almacen usada típicamente en otros muchos programas del estilo.

Consta de una serie de atributos como *coste_, h_ y g_*, que serán los que almacenan los datos necesarios para llevar a cabo el cálculo del camino con el A*. Por otro lado también cada Nodo almacena su posición, y la posición del nodo padre que lo ha generado.

Cabe decir que el booleano *elegido_*, se trata simplemente de un atributo para comprobar cuando ha sido elegido para formar parte del camino por el algoritmo.

El resto de la clase se podría resumir en que cada uno de los atributos tiene sus métodos correspondientes de lectura y escritura, propios de las clases de este estilo.

3.2. Métodos destacados de cada clase.

En este apartado hablamos de los métodos utilizados en las clases implementadas para el programa y cuál es la función de cada uno.

3.2.1. Tablero.

1. **Tablero();**
2. **Tablero(int, int);**
3. **Tablero(const Tablero&);**
4. **~Tablero();**

Los tres primeros métodos que se encuentran son el constructor por defecto, constructor con parámetros y constructor de copia respectivamente, el cuarto es el destructor.

El constructor por defecto se utiliza para inicializar el objeto de la clase Tablero. Al constructor con parámetros le pasamos como parámetros dos enteros, el número de filas y columnas, para crear nuestro tablero.

El paso por valor es paso por copia, y para crear el objeto local de la clase **Tablero**, a partir del parámetro real *mapa_*, se utilizará un constructor llamado "*constructor de copia*". Este constructor acepta como parámetro un objeto de la clase **Tablero** (su misma clase) para crear el nuevo objeto. Este *constructor de copia* existe por defecto en todas las clases de C++, aunque no se defina explícitamente. Este constructor por defecto asigna uno a uno los miembros del nuevo objeto *car_* igualándolos con los del objeto ya existente. De ahí que los dos objetos, compartan en realidad, la misma memoria.

El destructor sirve para eliminar el objeto creado en la clase **Tablero** después de que se termine utilizándolo.

5. **void redimensionar ();**

Éste método se utiliza par redimensionar el tamaño del tablero, al principio el tamaño del tablero está puesto por defecto a 20x60 pero pregunta si quiere seguir con este tamaño o se puede cambiar manualmente por el usuario insertando los valores de las filas y columnas deseadas.

6. **void colocarCoche ();**

   En éste método se coloca el coche en su posición inicial y se puede hacer manualmente pidiendo al usuario las coordenadas donde le gustaría situar al coche , si las coordenadas no son válidas se vuelve a preguntar hasta que lo sean, o aleatoriamente por el programa teniendo en cuenta el tamaño del tablero.

7. **void rellenarObstaculos ();**

   Se rellenan los obstáculos manualmente pidiendo al usuario el número de obstáculos que le gustaría insertar y después las coordenadas de cada uno sin poder repetir las coordenadas de un obstáculo ya establecidos o aleatoriamente por el programa estableciendo el número de obstáculos al 0.1 del tablero. Por la pantalla se demostrarán con el símbolo ‘X’ sobre el tablero.

8. **void rellenarPersonas ();**

   Tiene la misma función que el método *rellenarObstaculos()* pero en vez de obstáculos serán personas y se representarán por la pantalla sobre el tablero con el símbolo ‘P’.

9. **void moverCoche (void);**

   Como el nombre indica, este método se encarga de mover el coche llamando a la función mover(std::vector<std::vector<char> >&); de la clase Coche pasando como parámetro el tablero.

10. **friend std::ostream& operator<< (std::ostream&, const Tablero&);**

    Método de sobrecarga para el operador de flujo de salida, que nos imprimirá por pantalla nuestro tablero con todos los elementos del entorno en las posiciones establecidas.

3.2.2. Coche.

1. **Coche();**
2. **Coche(const Coche&);**
3. **~Coche();**

Los dos primeros métodos son los constructores. El primer constructor sitúa al coche en la posición i=0 y j=0 si no se fija manualmente y el segundo constructor sitúa al coche en las coordenadas pasadas como parámetros.
El tercero es el destructor.

4. **void mover (std::vector<std::vector<char> >&, int);**

Ésta función se encarga de mover el coche por el tablero pasado como parámetro y en las direcciones indicadas que recibe igualmente por parámetro.

5. **void calcularCamino (std::vector<std::vector<char> >, posicion, posicion, int);**

Esta función del coche es la más importante, ya que es la que recurre a su instancia de la clase camino para calcular el recorrido.

Como se puede observar, los parámetros son la matriz de caracteres del mapa, las dos posiciones: inicio y final, y un entero que nos indicará qué función heurística se desea usar.

6. **friend std::ostream& operator<< (std::ostream&, const Coche&);**

Al igual que en la clase anterior, ésta sobrecarga del operador salida muestra el coche como tal, con la una peculiaridad de que recurre a su propio atributo del símbolo que usará.

3.2.3. Camino.

1. **Camino();**
2. **Camino (const Camino&);**
3. **~Camino();**

4. **void calculo (std::vector<std::vector<char> >, posicion, posicion, int);**

    Esta función encierra el grueso de todo el algoritmo, ya que a través de los parámetros, se calcularán todos los nodos posibles en la generación del camino, distinguiendo entre aquellos nodos que son padres para no volver a recorrerlos, y aquellos que no son posibles, como los obstáculos.

    Dentro de esta función también se establecen los valores estimados por funciones heurísticas para los nodos que se van a evaluar.

5. **Nodo evaluarFrontera (posicion);**

    Esta función se encarga de simplemente de evaluar los nodos posibles, y de todos ellos escoger uno que será el que se expanda en la siguiente iteración.

6. **void establecerH_1 (posicion, Nodo&);**
7. **void establecerH_2 (posicion, Nodo&);**

Estas dos funciones son las encargadas de estimar la distancia entre el nodo *X* y la posición final. Con dicho cálculo podremos elegir el nodo mejor que nos lleve hasta la solución.

3.2.4. Nodo.

1. **Nodo& operator= (const Nodo&);**
2. **bool operator== (const Nodo&) const;**

    Estos dos métodos son los que más podemos destacar de la clase Nodo, y son la sobrecarga de los operadores ‘=’ y ‘==’ que nos sirven para hacer muchas comprobaciones dentro del algoritmo de búsqueda.

|4. Tabla de estadísticas.

Parámetros		**Heurística 1**		**Heurística 2**	
Dimensión	% de Obs.	Nodos generados	Longitud del camino	Nodos generados	Longitud del camino
20 x 20	10%	32	10	26	10
20 x 20	30%	6	3	6	3
50 x 50	10%	18	7	19	7
50 x 50	30%	109	26	105	30
100 x 100	10%	3098	98	304	98
100 x 100	30%	534	54	328	56
250 x 250	10%	1799	77	322	77
250 x 250	30%	9919	226	10601	301
500 x 500	10%	16872	457	9604	367
500 x 500	30%	6721	188	3526	188
750 x 750	10%	20888	413	8018	413
750 x 750	30%	35809	447	18682	463
1000 x 1000	10%	Inviable			
1000 x 1000	30%				

|5. Funciones heurísticas.

Para resolver este problema hemos utilizado las dos siguientes funciones heurísticas:

1) La distancia Euclidiana : en este problema, de espacio bidimensional, la distancia euclidiana entre el punto origen (P) y el punto final (Q) de coordenadas *(p1, p2)* y *(q1, q2)* se calcula de la siguiente manera :

   $d_E = (P, Q) = sqrt((q1-p1)^2 + (q2-p2)^2)$.

2) La distancia rectilínea : mide la distancia de dos puntos con una serie de giros de 90º, siendo P el nodo de origen con coordenadas *(p1, p2)* y Q el nodo final con coordenadas *(q1, q2)*, se calcula de la siguiente manera :

   $d_{ab} = | p1- q1 | + | p2 -q2 |$

|6. Experiencia computacional.

Para este problema, de encontrar el camino más óptimo para llegar hasta el punto final hemos utilizado el algoritmo de búsqueda A*. Este algoritmo se clasifica entre los algoritmos de búsqueda en grafo y su función es encontrar el camino de menor coste desde su nodo origen hasta su nodo final sólo si se cumplen unas determinadas condiciones.

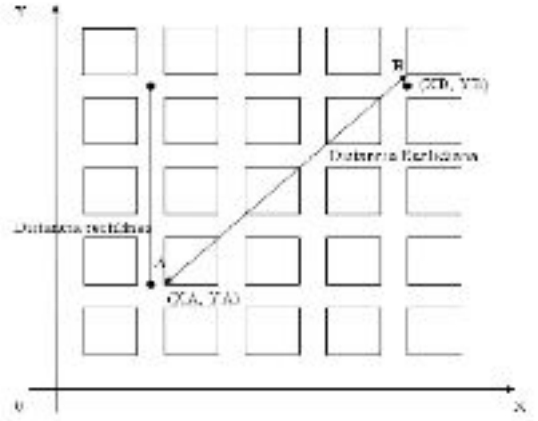
Dentro del algoritmo de búsqueda A* hemos utilizado la heurística de Manhattan Distance. Ésta heurística se utiliza cuando solo tenemos los 4 movimientos básicos posibles (derecha, izquierda, arriba, abajo) y ninguna otra.

El tiempo necesario para resolver el problema depende de la situación del problema, es decir, el tamaño del tablero, número de obstáculos y sus posiciones, posición del punto origen y el punto final, la distancia entre los dos puntos y la heurísticas utilizada. Por ejemplo:

1) El tablero podría ser de menor tamaño pero la distancia entre los dos puntos podría ser mayor y las posiciones de los obstáculos podría bloquear la mayoría de los caminos para llegar el punto final.

2) También podría ser que el tamaño del tablero podría ser mayor pero la distancia entre los puntos es menor y los obstáculos no bloquean la mayor parte del camino y se podría llegar en menos tiempo al punto final.

Para resolver el problema, hemos utilizado el cloud 9 para escribir el programa correspondiente en C++ y demostrar la simulación del entorno en la consola.



Norma rectilínea $d_{AB} = |X_A - X_B| + |Y_A - Y_B|$

Norma euclidiana $d_{AB} = [(X_A - X_B)^2 + (Y_A - Y_B)^2]^{1/2}$

|7. Conclusión.

Como conclusión a ésta práctica cabe destacar la cantidad de problemas que hemos tenido a la hora de implementar la parte final de los algoritmos de búsqueda, puesto que al principio no conocíamos el funcionamiento de dichos algoritmos y trabajamos de cierta manera, la cual tuvimos que adaptar para poder incluir los algoritmos, resultando en un programa incompleto pero que internamente lleva mucho trabajo, aunque de cara al usuario parezca que es un ejecutable muy pobre en cuanto a implementación.

Atendiendo ahora a la parte de algoritmos de búsqueda, podemos decir que hay muchas maneras de encontrar la distancia entre dos puntos de manera heurística, pero nosotros nos concentramos en dos de ellos.Para medir distancias se puede utilizar una norma rectilínea, o bien una euclidiana, lo que hemos hecho en nuestro caso para resolver este problema.

La primera tiene mayor aplicación, por ejemplo, en grandes ciudades, con trazos rectos perpendiculares y paralelos de calles y avenidas y donde la distancia

entre dos puntos no puede medirse como la recta que los une, sino como el mínimo número de calles que exista entre ambos.

Por el otro lado, la norma Euclidiana dice que la distancia entre dos puntos es la recta que los une. Esta norma tiene sentido, por ejemplo, en zonas rurales y urbanas con trazo irregular de calles.