Informe Inteligencia Artificial

Coche automático

Alejandro González González
Aitor Hernández Sánchez
Adrián Melian Fernández

Índice:

Primera semana:

Objetivo: Planteamiento de un sistema automóvil independiente basado en agentes de sensores y realizando acciones a través de efectores para llevar a cabo un objetivo.

Tabla de los Agentes del sistema:

Percepciones	**Acciones**	**Objetivos**	**Entorno**
Cámaras	Acelerar	Seguridad	Peatones
Sensores Acústicos/Proximidad	Frenar	Eficiencia	Señales
Geolocalización	Girar	Ecológico	Coches
Sensores Temperatura/ Seguimiento/ Luz/Radio	Alertas Acústicas/Emergencia/		
	Encendido/Apagado de luces		

Planteamiento: El sistema automóvil propuesto será de un vehículo que tenga la capacidad de percibir el entorno que le rodea y, que sea capaz de tomar la decisión correcta para las distintas situaciones que se le presenta. Para ello utilizaremos el lenguaje C++ creando un programa donde a partir de un entorno virtual se pondrá a prueba las distintas acciones que realizará dicho vehículo.

Solución: Para esto crearemos las distintas clases que nos harán falta para el entorno virtual al igual que el comportamiento del coche, estas son:

```
class map {
// donde están incluidos los atributos del propio mapa, los obstáculos y las personas//
}
```

```
class car {
// será la clase que proporcionará el movimiento al coche (izquierda, derecha, arriba y abajo)//
}
```

Segunda Semana:

Desarrollo de código: En este apartado realizaremos una explicación más detallada del diseño del código, dividido en los tres archivos:

- *main.cpp*
- *map.cpp/map.hpp*
- *car.cpp/car.hpp*

car.cpp/car.hpp:

Esta clase, como bien hemos comentado más arriba, realizará los movimientos naturales de un coche, para ello tomamos su posición como un eje de coordenadas *(x,y)* dentro de un map.cpp.

```
class car{
private:
      int x;
      int y;
public:
  car():x(0),y(0){}
  ~car(){}
  int mov_este(int i);
  int mov_norte(int i);
  int mov_oeste(int i);
  int mov_sur(int i);
  int pos_x();
  int pos_y();
```

```
};
```

map.cpp/map.hpp:

Para el desarrollo del mapa se ha utilizado un `std::<vector::pair<int,int>>`, que simulará la matriz a partir de rellenando cada una de las posiciones *(x, y),* según estén ocupadas por un obstáculo, una persona o el propio coche. Por el momento, sólo está disponible los obstáculos como prototipo, a medida que se vaya avanzando se implementará el resto de atributos de esta clase.

```
class map {

private:
	std::vector <std::pair<int, int>> map_;
	int N_, M_;
	std::vector <std::pair<int, int>> obstacles;
public:
	map(int m, int n);
	~map();
	int get_pos(std::pair<int, int> pos_xy);
	void handmade_obst();
	void random_obst();
	void write(std::ostream& os) const;
};
```

A su vez, se ha añadido el método `void ramdom_obst(),` en donde pondrá aleatoriamente los obstáculos y las personas, por el momento está en fase experimental hasta encontrar la manera de elaborarlo.

Tercera Semana:

random_obst(): La semana pasada comenzamos a desarrollar el código de las distintas clases, sin embargo no fueron terminadas del todo, aquí incluimos como fue desarrollado el método para asignar los obstáculos aleatoriamente:

```
void map::random_obst(){
      int obstáculos=0,m=0,n=0;
      std::pair<int,int> coordenadas;
      std::default_random_engine generator;
      std::uniform_int_distribution<int> distribution(0, map_.size()-2);
      std::uniform_int_distribution<int> horizonte(0, N_);
      std::uniform_int_distribution<int> vertical(0, M_);
      obstaculos=distribution(generator);
      for(int i=0;i<=obstaculos;i++){
      m=vertical(generator);
      n=horizonte(generator);
      coordenadas.first=m;
      coordenadas.second=n;
      obstacles.push_back(coordenadas);
      }
}
```

Fase de simulación:

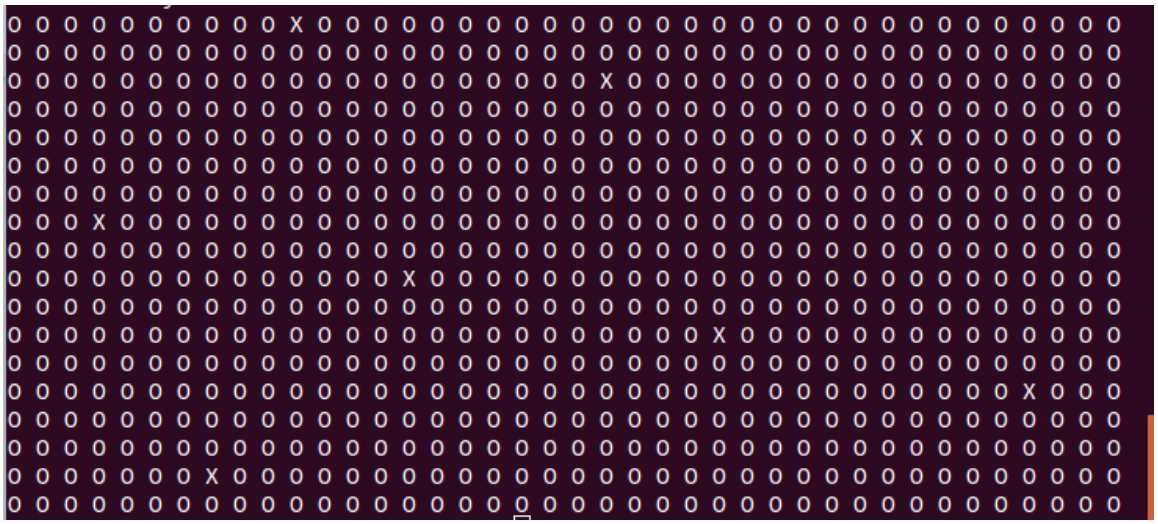
Por último, en la fase de simulación o de impresión de la rejilla se ha encontrado un problema de optimización, y es que durante la impresión del vector `map_` tiene que estar comprobando continuamente en el vector `obstacles` cada vez que se quiere imprimir una casilla.

```
void map::write(std::ostream& os) const{
  for(auto it=obstacles.begin();it != obstacles.end();it++){
```

```
      for(int i=0;i<=M_;i++){
            for(int j=0;j<=N_;j++){
            std::pair<int,int> coor=std::make_pair(i,j);
            if(*it==coor){
                  os << "X ";
            }
            else{
                  os << "O ";
            }}
      }}
}
```

Observación: se buscará la manera de solucionar el problema de impresión.

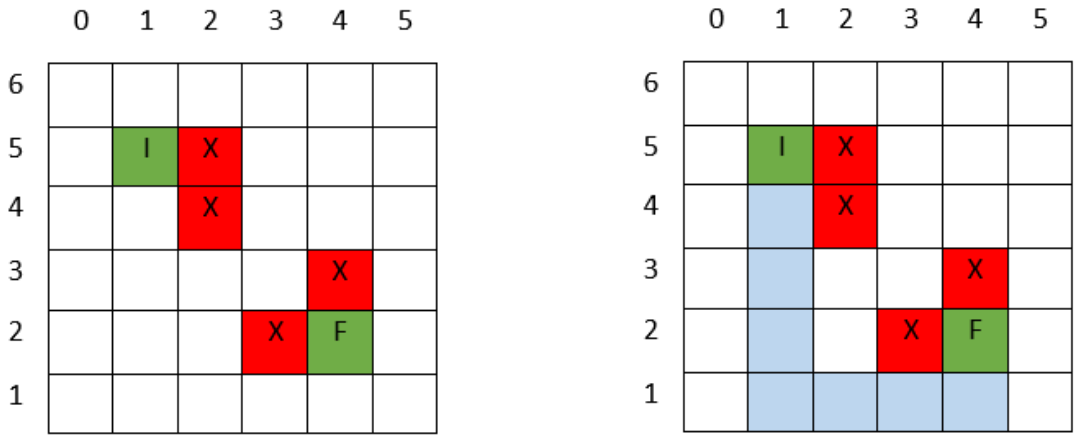
Ejemplo de simulación con 9 obstáculos marcados como 'X' y, las posiciones libres como '0':



Cuarta Semana:

Algoritmo A*: El algoritmo A* es usado para encontrar la ruta más cercana para ir de un lugar a otro (llamados nodos), es el más usado debido a que es sencillo y rápido.

Ejemplo del algoritmo:



El algoritmo funciona de la siguiente manera:

1. Se retorna el nodo inicial como solución si es igual al nodo final.
2. Si no, se adiciona el nodo inicial a la lista abierta
3. Mientras la lista no esté vacía, se recorre cada nodo que haya en la lista abierta y se toma el que tenga el costo total más bajo.
4. Si el nodo obtenido es igual al nodo final, se retornan todos los nodos sucesores al nodo encontrado.
5. Si no, se toma el nodo y se elimina de la lista abierta para guardarse en la lista cerrada y se buscan todos los nodos adyacentes al nodo obtenido y se adicionan a la lista abierta a menos que el nodo se encuentre en la lista cerrada o que el nodo sea sólido
6. Si el nodo adyacente ya se encuentra en la lista abierta se verifica que el costo sea menor, si es menor se cambian los valores de costo, sino se ignora

7. Se vuelve al paso 3 y se repite hasta que el punto 4 sea verdadero o que la lista abierta quede vacía.

Clase A*: el algoritmo de búsqueda estará dividida en dos clases una clase búsqueda y una clase nodo.

busqueda.cpp/busqueda.hpp:

Para la realización de la clase búsqueda es necesario, como comentamos en el apartado anterior, dos listas una abierta y otra cerrada para guardar los nodos surgentes, esto lo realizaremos con un vector donde ser irán almacenando dichos nodos.
Los atributos de dicha clase están descritos de la siguiente manera:

```
class busqueda{
    private:
        int costoIrDerecho = 1;
        int N,M;
        std::vector<nodo> listaAbierta = std::vector<nodo>();
        std::vector<nodo> listaCerrada = std::vector<nodo>();
        map mapa;

    public:
        busqueda();
        void adicionarNodoAListaAbierta(nodo nodo);
        std::vector<nodo> encontrarCamino(int x,int y);
        std::ostream write(std::ostream& os);
        std::vector<nodo> encontrarNodosAdyacentes(nodo nodoActual, nodo
nodoFinal);
        std::vector<std::pair<int,int>> camino;
        bool esIgual(nodo,nodo);

};
```

No hay mucho que mencionar en este apartado, ya que el algoritmo irá comparando las posiciones de los nodos *(x, y)* hasta llegar al nodo final con el mínimo costo posible. Como existen obstáculos y personas dentro de la simulación el atributo `std::vector<nodo> encontrarNodosAdyacentes(nodo nodoActual, nodo nodoFinal);` que examinará los nodos teniendo en cuenta el vector de obstacles y avanzará por el camino según le

convenga *(izquierda, derecha, arriba, abajo).* Por último, `std::vector<nodo> busqueda::encontrarCamino(int x,int y);` que será donde recorrerá el camino y calculará los costos.

nodo.cpp/nodo.hpp:

Es una clase creada para calcular los costos de los nodos, pues posee las dos funciones heurísticas que se utilizarán (pudiendo elegir una u otra) para calcular el costo estimado del nodo actual al nodo final.

```
class nodo{
private:
    nodo* NodoPadre;
    nodo* NodoFinal;
    float costeG;
public:
      int i;
      int j;
    nodo(){}
    nodo(nodo* nodopadre, nodo* nodofinal,int x,int y, float costo);
    float Calcularcosto();
      bool esIgual(nodo Nodo);
      nodo getnodopadre();
      nodo igualar(nodo Nodo,nodo Nodo1);

    float costeTotal;
};
```

Estas heurísticas están implementadas dentro de la función de `float Calcularcosto()`, que devuelve un float con ese costo estimado.
Esta función será llamada dentro del constructor parametrizado, cuando se calcule el coste total (`costeTotal = costeG + Calcularcosto()`) donde `float costeG` es el coste actual que lleva ese nodo. Este calculo se hará siempre y cuando no se haya llegado al nodo final.
El resto de funciones y atributos no merecen especial atención pues son bastante intuitivos.

Experiencia Computacional:

		Heurística1			Heurística2		
Dimensión	**Obstáculos**	**Longitud del camino**	**CPU**	**Nodos generados**	**Longitud del camino**	**CPU**	**Nodos generados**
20x20	10%	21	0.03125	158	21	0.03125	158
20x20	20%	21	0.03125	139	22	0.03125	139
20x20	30%	27	0.0156	127	27	0.015625	115
20x20	40%	28	0.0312	106	29	0.015625	112
20x20	50%	No se puede llegar porque hay demasiados obstáculos					
…	…						
30x30	10%	32	0.265625	352	32	0.2500	417
30x30	20%	36	0.328125	339	34	0.2500	299
30x30	30%	47	0.34375	310	36	0.265625	272
30x30	40%	42	0.15625	202	40	0.328125	272
30x30	50%	No se puede llegar porque hay demasiados obstáculos					
…	…						
50x50	10%	51	4.0000	857	51	4.00000	857
50x50	20%	51	4.3125	756	51	4.23438	756
50x50	30%	51	1.4843	424	51	3.48438	610
50x50	40%	51	0.9687	314	51	1.75000	420
50x50	50%	No se puede llegar porque hay demasiados obstáculos					
…	…						
100x100	10%	110	285.985	3474	110	414.02	3940
100x100	20%	110	305.066	3059	110	370.119	3279
…	…						

Conclusiones:

De la experiencia computacional podemos extraer principalmente que ambas funciones heurísticas tienen un crecimiento exponencial en el tiempo de ejecución, en cuanto al mapa generado.
Es por esto por lo que ya con un mapa de 50x50 y unos 250 obstáculos (10%), el tiempo de ejecución empieza a ser notable.

También podemos observar que más obstáculos significa también mayor tiempo de ejecución, aunque la heurística 2 redujo considerablemente el tiempo con más obstáculos.

Comparando las dos heurísticas, podemos ver que la primera es más eficiente, tanto en tiempo de ejecución como en los nodos generados. Algo que a primera vista no hubiésemos dado por supuesto, ya que tuvimos en mente que a la hora de trabajar en el entorno de un mapa como una matriz el algoritmo se manejaría mejor con la estimación del camino más corto dado por las celdas que lo separan y por la restricción del propio movimiento(imposibilidad de moverse de manera diagonal entre celdas), y no, como en este caso vimos, que se maneja mejor con la estimación dada por la primera heurística que si calculaba la línea recta entre punto inicial y final, haciendo referencia a un movimiento diagonal.

Repositorio en GitHub: https://github.com/alu0101015729/IA.git