

# Práctica 1

## Documentación código BÚSQUEDA A\*



Christian Torres González

Inteligencia artificial

Practica 1: Búsqueda A\*

Inteligencia Artificial

Práctica 1

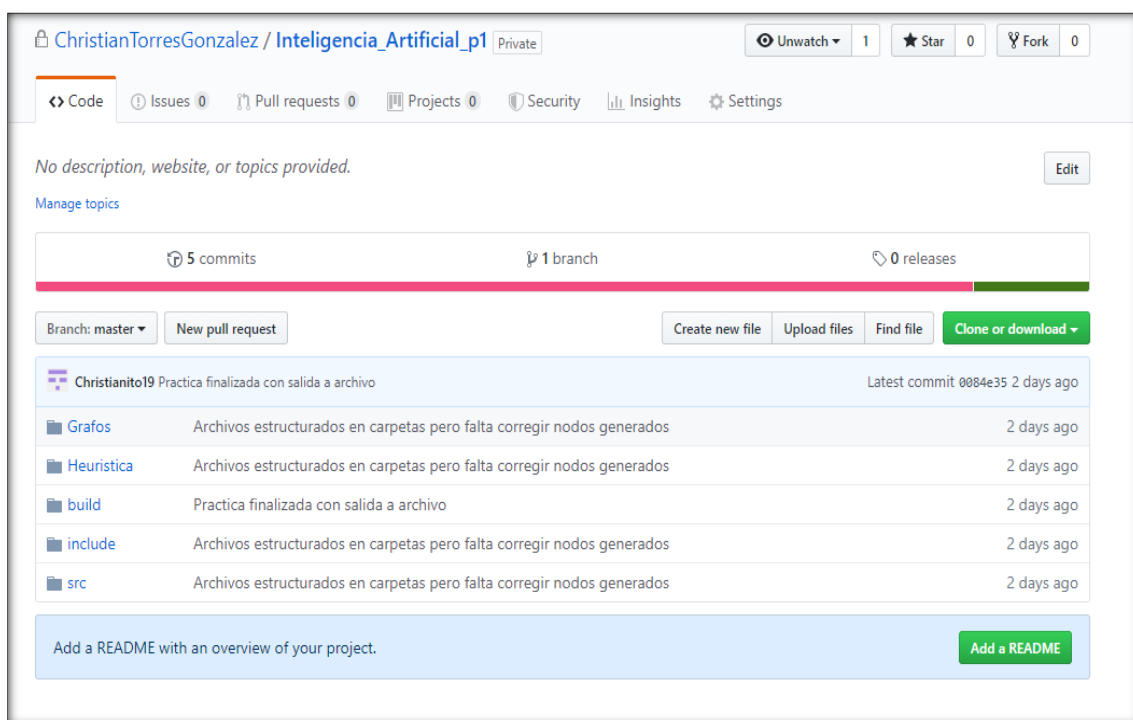
# Índice

1. Jerarquía de directorios
2. Funcionamiento del programa
3. Documentación de main.cpp
4. Documentación de grafo.hpp y grafo.cpp
  - a. Documentación triple.hpp y triple.cpp
5. Documentación heurística.hpp y heurística.cpp
6. Documentación árbol.hpp y árbol.cpp
  - a. Documentación nodo.hpp y nodo.cpp
7. Documentación búsqueda.hpp y búsqueda.cpp
8. Tabla de resultados

## 1. Jerarquía de directorios.

Antes de comenzar a hablar del funcionamiento del código y de las diferentes funcionalidades implementadas, expliquemos de que manera se van a almacenar los diferentes ficheros tanto de código como los grafos y sus correspondientes heurísticas.

De manera esquematizada podemos observar varios directorios Grafos, Heurística, include, src y build.



- **Grafos**: en este directorio se encuentran almacenados los diferentes grafos con los que iremos trabajando, de tal manera que cada vez que queramos trabajar con un grafo nuevo, simplemente tendremos que añadirlo a este directorio.
- **Heurística**: al igual que con el directorio anterior, es aquí donde almacenaremos las heurísticas pertenecientes a cada grafo. Al igual que en el caso anterior, cuando

incluimos un grafo en el directorio de grafos, añadiremos a este directorio su correspondiente heurística.

- **Build**: en este directorio se encuentran los archivos necesarios para la compilación del código, es decir, el fichero Makefile, y el ejecutable del programa.
- **Include**: en este directorio se encuentran almacenados todos los ficheros cabeceras de los diferentes ficheros de implementación utilizados para la elaboración de la práctica.
- **Src**: finalmente, es en este directorio en donde se encuentran almacenados los ficheros donde se encuentra el código de la búsqueda implementado en sus respectivas clases y funciones.

## 2. Funcionamiento del programa

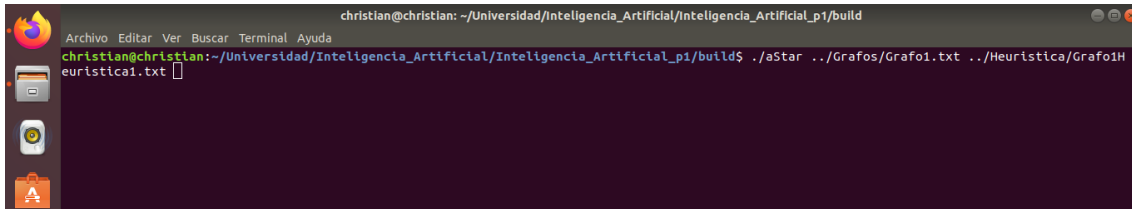
A continuación, se mostrará mediante imágenes el funcionamiento del programa, es decir como deberemos ejecutar el programa a través de la consola, que parámetros hay que suministrarle y finalmente la salida que se obtiene al ejecutar el programa.

Tal y como se muestra para la ejecución del programa, primero nos deberemos situar en el directorio build, ya que como comentamos en el apartado de “Jerarquía de directorios (1)”, es en este directorio donde se encuentran los ficheros pertenecientes para la compilación y ejecución del programa. Una vez situados en el directorio, deberemos escribir en la terminal

```
./aStar <../Grafos/nombre_grafo> <../Heuristica/nombre_heurística>
```

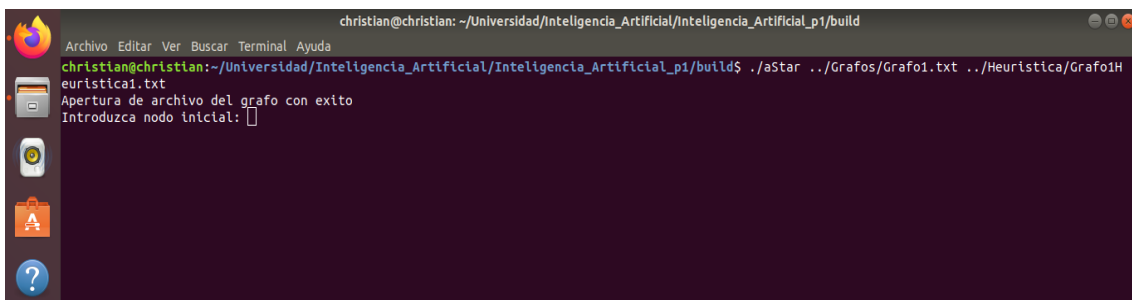
Lo que estamos introduciendo a través de estos comandos son

- **./aStar**: nombre del fichero ejecutable del programa
- **<../Grafos/nombre\_grafo>**: a continuación, escribimos el nombre del fichero que contiene al grafo. Debemos especificar que se encuentra en “../Grafos/\*\*\*” ya que no se encuentran en el mismo directorio.
- **<../Heuristica/nombre\_heurística>** finalmente, introduciremos el nombre del fichero que contiene los valores heurísticos perteneciente al grafo introducido. Al igual que con los grafos, al no encontrarse en el mismo directorio, deberemos indicar la ruta.



```
christian@christian: ~/Universidad/Inteligencia_Artificial/Inteligencia_Artificial_p1/build
christian@christian:~/Universidad/Inteligencia_Artificial/Inteligencia_Artificial_p1/build$ ./aStar ../Grafos/Grafo1.txt ../Heuristica/Grafo1Heuristica1.txt
```

Al introducir el comando y comenzar con la ejecución, nos pedirá que introduzcamos el nodo inicial, pero no el final, ya que este es calculado directamente del programa a través del valor heurístico, siendo este cuando coincida que es 0.



```
christian@christian: ~/Universidad/Inteligencia_Artificial/Inteligencia_Artificial_p1/build
christian@christian:~/Universidad/Inteligencia_Artificial/Inteligencia_Artificial_p1/build$ ./aStar ../Grafos/Grafo1.txt ../Heuristica/Grafo1Heuristica1.txt
Apertura de archivo del grafo con éxito
Introduzca nodo inicial:
```

Una vez introducido el grafo inicial, el programa automáticamente calculara el camino mas corto. Una vez el programa se ha terminado, éste muestra por pantalla los datos obtenidos, siendo estos:

- Camino a seguir desde el origen al destino con coste mínimo.
- Coste que supone seguir el camino.
- Nodos generados.
- Nodos inspeccionados.

```

christian@christian: ~/Universidad/Inteligencia_Artificial/Inteligencia_Artificial_p1/build
christian@christian:~/Universidad/Inteligencia_Artificial/Inteligencia_Artificial_p1/build$ ./aStar ../Grafos/Grafo1.txt ../Heuristica/Grafo1Heuristica1.txt
Apertura de archivo del grafo con éxito
Introduzca nodo inicial: 1
1 -> 2 -> 4 -> 12 -> 5 -> 13 -> 8 -> 3
Costo camino: 43
Nodos generados: 39
Nodos inspeccionados: 31
christian@christian:~/Universidad/Inteligencia_Artificial/Inteligencia_Artificial_p1/build$

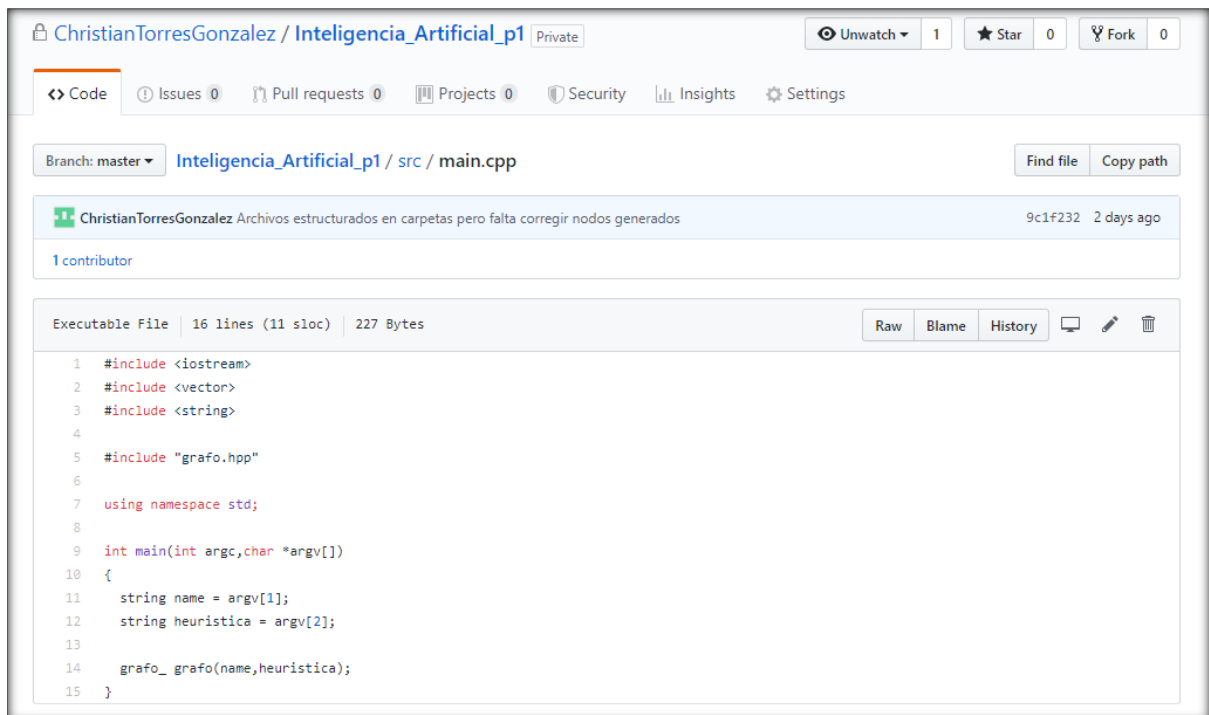
```

Pero esta no es toda la información, ya que el programa además de los nodos, coste y camino también muestra las aristas entre los diferentes nodos con sus respectivos costes, los sucesores de cada nodo y además los valores heurísticos.

1	3,111	9
10	6,10	10
11	6,10	9
12	6,60	3
13	6,10	7
14	3,60	9
15	3,10	1
16	3,10	5
17	6,60	3
18	6,10	9
19	6,70	13
20	6,60	7
21	6,100	11
22	7,60	13
23	7,10	8
24	7,10	5
25	6,70	8
26	6,10	11
27	6,100	7
28	9,60	7
29	9,10	7
30	10,60	11
31	11,10	9
32	12,60	7
33	12,10	1
34	13,10	5
35	13,60	11
36	14,60	7
37	15,70	5
38	15,10	7
39		
40	31	32, 33
41	31	32, 33, 40
42	31	32, 33, 40, 11
43	40	32, 33, 40, 11
44	40	32, 11, 13
45	40	40, 32, 7, 9, 10
46	71	40, 33, 10
47	81	71, 11, 10
48	81	40, 11
49	100	81
50	110	71
51	110	40, 9
52	110	71, 9
53	140	81
54	150	71, 9

### 3. Documentación de main.cpp

Nuestro fichero main.cpp no es muy extenso ya que simplemente contiene la lectura de los nombres de los archivos y la llamada a la clase grafo, comentada más adelante.



```

1  #include <iostream>
2  #include <vector>
3  #include <string>
4
5  #include "grafo.hpp"
6
7  using namespace std;
8
9  int main(int argc, char *argv[])
10 {
11     string name = argv[1];
12     string heuristica = argv[2];
13
14     grafo_ grafo(name, heuristica);
15 }

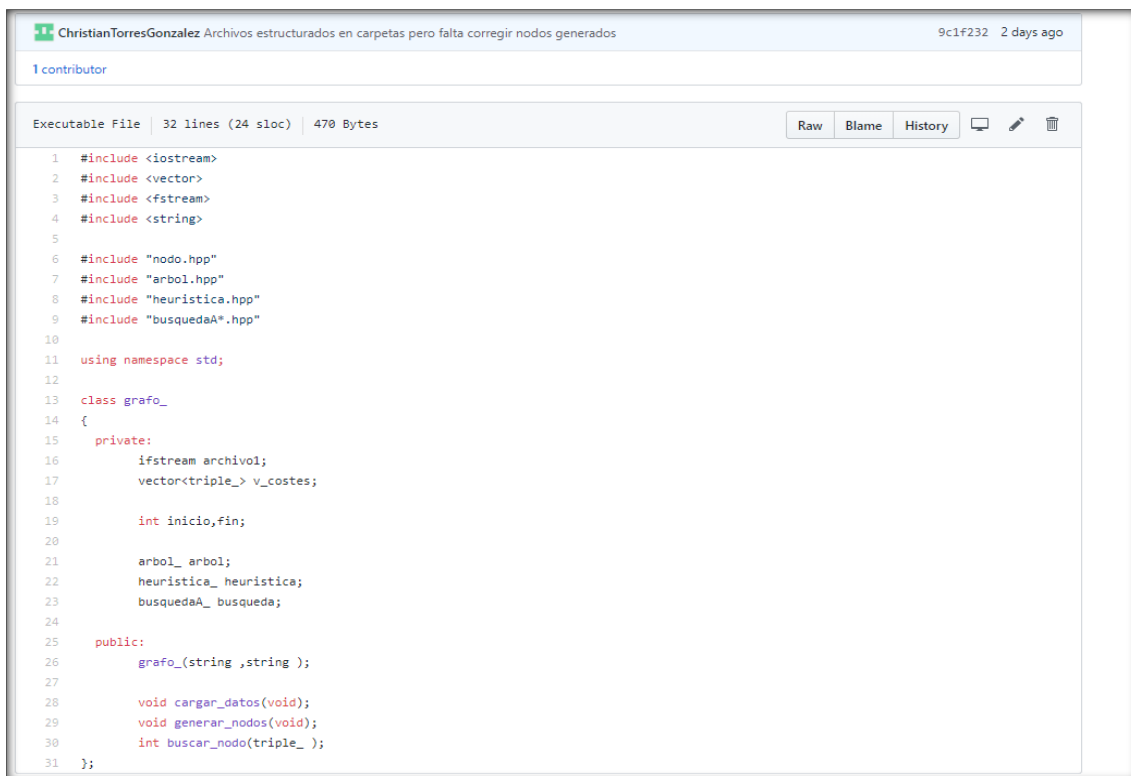
```

Tal y como se aprecia en la imagen, nuestro fichero “main.cpp”, se encarga de la lectura de los nombres de los archivos del grafo y de la heurística y se los pasa a la clase grafo que será la encargada de gestionar la búsqueda.



## 4. Documentación de grafo.hpp y grafo.cpp

Como ya adelantamos en la documentación del main.cpp, la clase grafo será la encargada de almacenar el árbol generado por los sucesores de cada nodo, un objeto de tipo heurística, el cual gestiona los valores heurísticos para cada nodo y finalmente un objeto de tipo búsqueda, este ultimo objeto es el encargado de generar la búsqueda entre el nodo inicial y final.



```

1  #include <iostream>
2  #include <vector>
3  #include <fstream>
4  #include <string>
5
6  #include "nodo.hpp"
7  #include "arbol.hpp"
8  #include "heuristica.hpp"
9  #include "busquedaA.hpp"
10
11 using namespace std;
12
13 class grafo_
14 {
15     private:
16         ifstream archivo1;
17         vector<triple_> v_costes;
18
19         int inicio,fin;
20
21         arbol_ arbol;
22         heuristica_ heuristica;
23         busquedaA_ busqueda;
24
25     public:
26         grafo_(string ,string );
27
28         void cargar_datos(void);
29         void generar_nodos(void);
30         int buscar_nodo(triple_ );
31 };

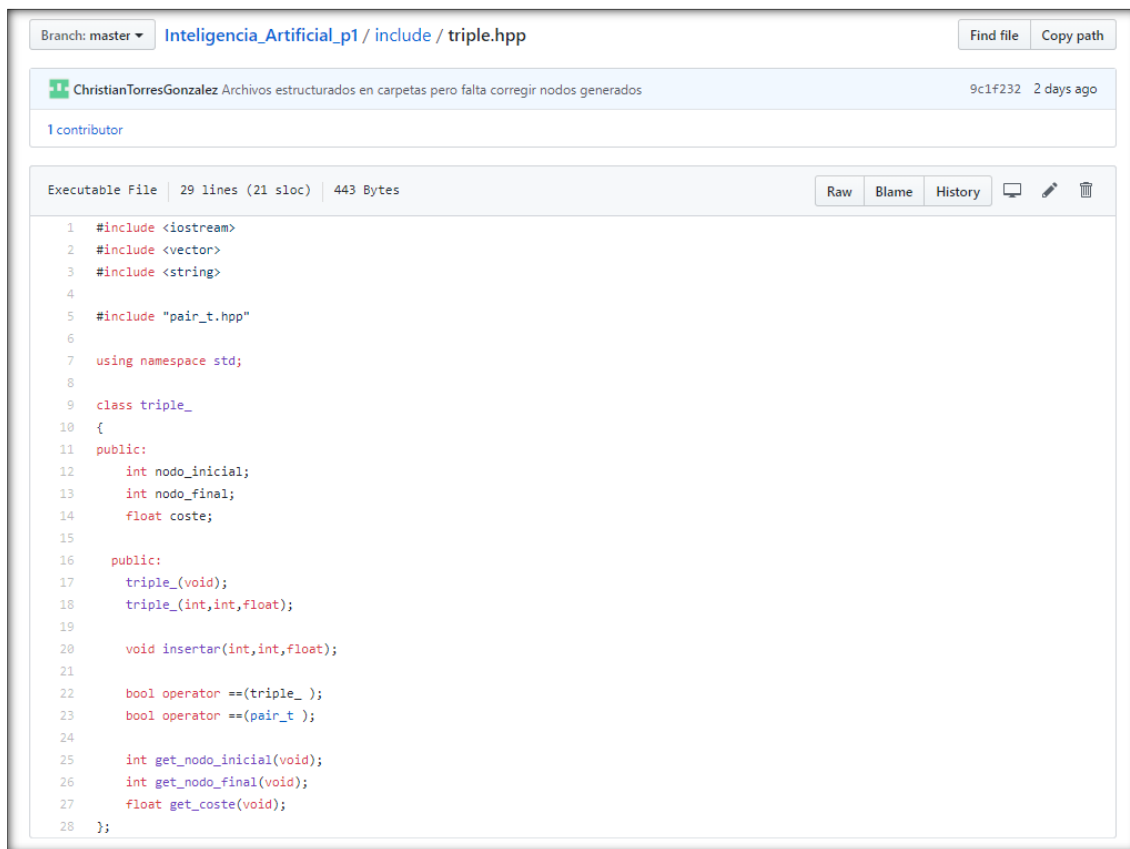
```

En cuanto a los métodos contenidos en esta clase, encontramos el método encargado de leer las transiciones para cada nodo del fichero de entrada “**cargar\_datos()**”, la función encargada de generar los nodos del árbol “**generar\_nodos()**” a partir de los sucesores y costes leídos en la función de “**cargar\_datos()**” y finalmente la función “**buscar\_nodo()**” que nos devuelve el nodo al que pertenece una respectiva transición con su correspondiente coste.

Para generar los nodos, la función “**cargar\_datos()**” hace uso de la clase triple. Esta clase es una estructura que nos permite almacenar el nodo en el que nos encontramos, el nodo hijo al que llegamos desde ese nodo y su respectivo coste.

## 4.1 Documentación triple.hpp y triple.cpp

Como ya hemos comentado, utilizaremos esta clase para almacenar nodo actual, nodo hijo y coste de la arista.



```

1 #include <iostream>
2 #include <vector>
3 #include <string>
4
5 #include "pair_t.hpp"
6
7 using namespace std;
8
9 class triple_
10 {
11 public:
12     int nodo_inicial;
13     int nodo_final;
14     float coste;
15
16 public:
17     triple_(void);
18     triple_(int,int,float);
19
20     void insertar(int,int,float);
21
22     bool operator ==(triple_ );
23     bool operator ==(pair_t );
24
25     int get_nodo_inicial(void);
26     int get_nodo_final(void);
27     float get_coste(void);
28 };

```

Al ser una clase que solo utilizaremos para almacenar este triple de datos, las únicas funciones que contiene serán la de insertar los datos y las funciones **get\_()** para leer los datos.

## 5. Documentación `heuristica.hpp` y `heuristica.cpp`

Una vez la clase grafo ha generado los nodos y sus sucesores con sus respectivos costes, se invoca al objeto `heuristica` para cargar los valores heurísticos para cada nodo.

```

1  #pragma once
2
3  #include <iostream>
4  #include <fstream>
5  #include <vector>
6
7  using namespace std;
8
9  class heuristica_
10 {
11     private:
12         ifstream archivo1;
13
14         vector<float> v_heuristico;
15
16     public:
17         heuristica_(string);
18
19         void cargar_heuristica(void);
20         void imprimir_heuristica(void);
21
22         int get_heuristica_size(void);
23         float get_heuristica_valor(int );
24
25 };
  
```

Dado que es una clase que solo se utiliza para almacenar los valores heurísticos, sus únicos métodos serán `cargar_heuristica()`, utilizado para leer dichos valores heurísticos del fichero de entrada. Método `imprimir_heuristico()` utilizado para la impresión de dichos valores. Y finalmente los métodos para acceder a dichos valores.

## 6. Documentación árbol.hpp y árbol.cpp

Como ya he comentado anteriormente, uno de los objetos que contiene la clase grafo, es el árbol que va a contener el nodo inicial, el nodo final y además el conjunto de nodos que tiene el grafo con su respectiva información.

```

1  #pragma once
2
3  #include <iostream>
4  #include <vector>
5
6  #include "nodo.hpp"
7
8  using namespace std;
9
10 class arbol_
11 {
12     public:
13         nodo_ raiz;
14         nodo_ fin;
15         vector<nodo_> conjunto_nodos;
16
17     public:
18         arbol_(void);
19
20         void insertar_sucesor(int ,pair_t );
21         void insertar(nodo_ );
22
23         void get_raiz(int );
24         void get_fin(int );
25         int get_size(void);
26         int get_val(int a);
27         float get_coste_hijo(int );
28         nodo_ get_nodo(int );
29
30         void set_raiz(int );
31         void set_fin(int );
32
33         void imprimir_conjunto(void);
34
35         nodo_ operator [](int );
36 };

```


Ese conjunto de nodos va a ser una estructura de vector de tipo nodo, lo que quiere decir que cada posición del vector será un objeto de tipo nodo.

Dado que solo es una clase utilizada para almacenar la información del grafo, sus únicas funciones serán de acceso a los datos de ahí que sus funciones solo sean `get_()` y `set_()` a parte de la de imprimir utilizada para mostrar dichos valores.

## 6.1 Documentación nodo.hpp y nodo.cpp

Como ya hemos adelantado y al igual que ocurre con el resto de las clases introducidas anteriormente, la clase nodo vuelve a ser una clase simplemente utilizada para almacenar toda la información referida a un nodo:

- Nodo actual
- Variable para comprobar si ha sido generado
- Variable para comprobar si ha sido inspeccionado
- Un vector usado para almacenar los nodos hijos con sus respectivos costes.


ChristianTorresGonzalez Archivos estructurados en carpetas pero falta corregir nodos generados
9c1f232 2 days ago

1 contributor

Executable File | 39 lines (28 sloc) | 618 Bytes
Raw | Blame | History

```

1  #pragma once
2
3  #include <iostream>
4  #include <vector>
5
6  #include "triple.hpp"
7
8  using namespace std;
9
10 class nodo_
11 {
12     private:
13         short int nodo;
14         bool generado;
15         bool inspeccionado;
16
17         vector<pair_t> sucesores;
18
19     public:
20         nodo_(void);
21         nodo_(int );
22
23         void insert(int );
24
25         void insertar_nodo_hijo(pair_t );
26         int get_nodo(void);
27         int get_size_sucesores(void);
28         int get_nodo_hijo(int );
29         bool get_inspeccionado(void);
30         float get_coste_hijo(int );
31         pair_t get_sucesores(int );
32
33         void set_inspeccionado(bool);
34         bool is_inspeccionado(void);
35
36         bool operator ==(nodo_ );
37         bool operator ==(triple_ );
38 };

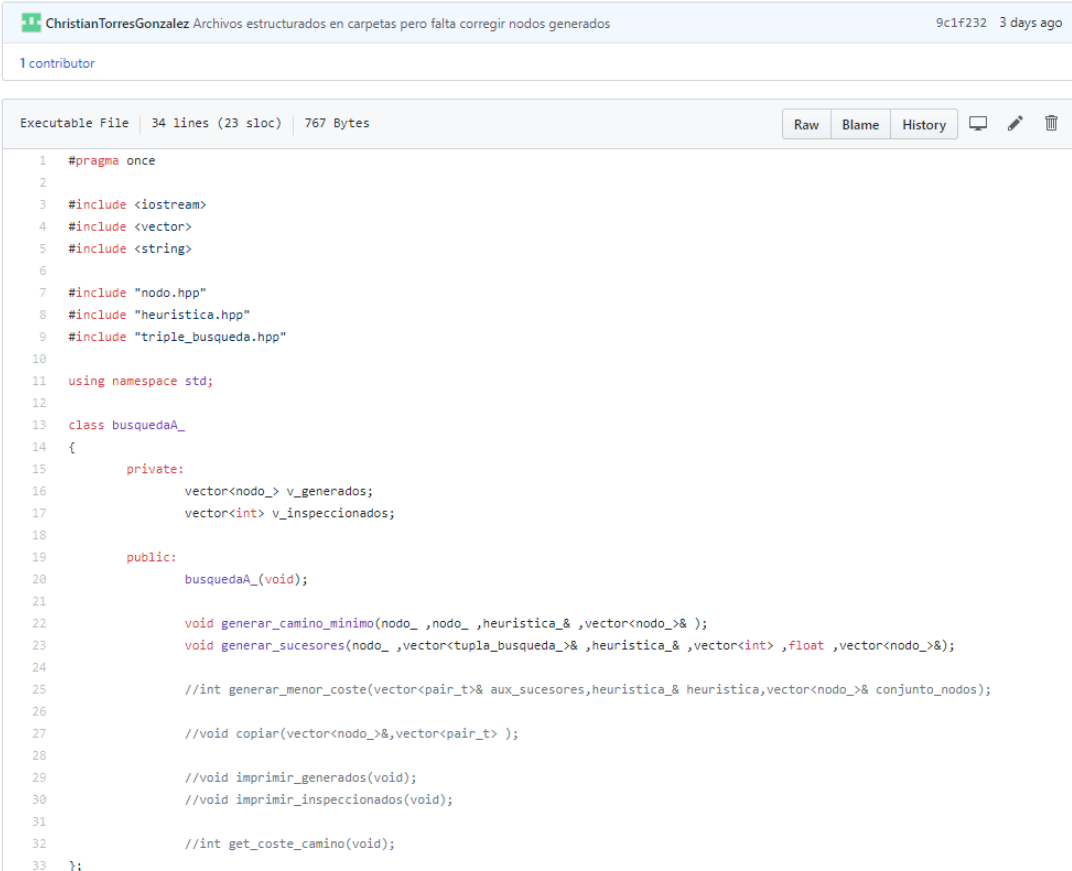
```

Dado que solo es una clase utilizada para almacenar la información del nodo, sus únicas funciones serán de acceso a los datos de ahí que sus funciones solo sean `get_()` y `set_()`.

## 7. Documentación búsqueda.hpp y búsqueda.cpp

Para la clase `búsquedaA_`, dado que es la clase en la que se centra la mayor parte de la práctica, dedicaremos un poco mas para explicar un poco mas detalladamente como se calcula el camino mínimo.

Para empezar, comentemos los atributos principales de la clase y a continuación comentaremos sus respectivos métodos.



The screenshot shows a code editor with the following content:

```

1  #pragma once
2
3  #include <iostream>
4  #include <vector>
5  #include <string>
6
7  #include "nodo.hpp"
8  #include "heuristica.hpp"
9  #include "triple_busqueda.hpp"
10
11 using namespace std;
12
13 class búsquedaA_
14 {
15     private:
16         vector<nodo_> v_generados;
17         vector<int> v_inspeccionados;
18
19     public:
20         búsquedaA_(void);
21
22         void generar_camino_minimo(nodo_ ,nodo_ ,heuristica_& ,vector<nodo_>& );
23         void generar_sucesores(nodo_ ,vector<tuple<busqueda_>& ,heuristica_& ,vector<int> ,float ,vector<nodo_>& );
24
25         //int generar_menor_coste(vector<pair_t>& aux_sucesores,heuristica_& heuristica,vector<nodo_>& conjunto_nodos);
26
27         //void copiar(vector<nodo_>&,vector<pair_t> );
28
29         //void imprimir_generados(void);
30         //void imprimir_inspeccionados(void);
31
32         //int get_coste_camino(void);
33 };
  
```

Como se puede apreciar, en la clase `búsquedaA_`, solo vamos a almacenar la lista con los nodos generados e inspeccionados ya que, para calcular el camino mínimo, los datos necesarios se los pasaremos directamente como argumentos a las funciones.

Una vez comentado los atributos, pasaremos a lo importante que son las funciones que nos calculan el camino mínimo.

Como se observa en la imagen, para calcular el camino utilizamos dos funciones, la principal que es la llamada **generar\_camino\_minimo()**, que es la utilizada, tal y como su nombre indica para calcular el camino.

```

14     void busquedaA::generar_camino_minimo(nodo_ raiz,nodo_ fin,heuristica_& heuristica,vector<nodo_>& conjunto_nodos)
15     {
16         vector<tupla_busqueda_> cola;
17         tupla_busqueda_ inicio(raiz.get_nodo(),0,0);
18         tupla_busqueda_ recorrido;
19
20         cola.push_back(inicio);
21         //v_inspeccionados.push_back(cola.front().get_nodo_inicial());
22
23         while (1)
24         {
25             tupla_busqueda_ actual = cola.front();
26             actual.set_camino(actual.get_nodo_inicial());
27             cola.erase(cola.begin());
28
29             //if (conjunto_nodos[actual.get_nodo_inicial() - 1].get_inspeccionado() == false)
30                 v_inspeccionados.push_back(actual.get_nodo_inicial());
31
32             conjunto_nodos[actual.get_nodo_inicial() - 1].set_inspeccionado(1);
33
34             if (actual.get_nodo_inicial() == fin.get_nodo())
35             {
36                 recorrido = actual;
37                 break;
38             }
39
40             generar_sucesores(conjunto_nodos[actual.get_nodo_inicial()-1],cola,heuristica,actual.get_camino(),actual.get_coste_
41         }
42
43         for (int i = 0;i < recorrido.get_camino_size();i++)
44             cout << recorrido.get_nodo_camino(i) << endl;
45         cout << "Coste camino: " << recorrido.get_coste_camino() << endl;
46         cout << "Nodos generados: " << v_generados.size() << endl ;
47         cout << "Nodos inspeccionados: " << v_inspeccionados.size() << endl;
48     }

```

El funcionamiento se basa en:

Una cola en la cual se van a ir almacenando en orden creciente de coste creciente los nodos con menor valor, de tal manera que cada vez que vamos a seleccionar un nodo candidato, siempre escogemos el nodo inicial de la cola. Dado que nuestra cola es un vector, en cada posición almacenaremos, el nodo en el que nos encontramos, el coste que acarrea llegar a ese nodo, el camino que hay que seguir a ese nodo. De esta manera, al almacenar el camino para cada nodo, cuando la búsqueda se encuentra que tiene que retroceder ya que hay un camino por otra rama, simplemente se cambiará el nodo inicial de la cola. Una vez



seleccionado el siguiente nodo candidato, procedemos a generar sus hijos mediante la función `generar_sucesores()`

```

50 void busquedaA::generar_sucesores(nodo_ nodo,vector< tupla_busqueda_>& cola,heuristica_& heuristica,vector<int> camino,float c_cami
51 {
52     for (int i = 0; i < nodo.get_size_sucesores(); i++)
53     {
54
55         if (conjunto_nodos[nodo.get_sucesores(i).get_hijo() - 1].get_inspeccionado() == false)
56         {
57             tupla_busqueda_ suceso(nodo.get_nodo_hijo(i),c_camino + nodo.get_coste_hijo(i) + heuristica.get_heuristica
58             v_generados.push_back(conjunto_nodos[sucesor.get_nodo_inicial()-1]);
59
60             if (cola.empty())
61                 cola.push_back(sucesor);
62             else
63             {
64                 int j = 0;
65                 while ((cola[j].get_coste_actual() <= sucesor.get_coste_actual()) && (j < cola.size()))
66                 {
67                     j++;
68                 }
69
70                 cola.insert(cola.begin() + j,sucesor);
71             }
72         }
73     }
74 }
75
76
77
78
79
80
81

```

Por último, solo nos quedaría comentar la función encargada de generar los sucesores para un nodo. Esta simplemente le pasaremos el nodo candidato y generaremos sus sucesores, con la única implicación de que a medida que vamos generando cada sucesor, lo vamos introduciendo directamente ordenado en función de su coste.

Una vez generado los sucesores, se volvería al ciclo de seleccionar el nodo inicial de la cola y seguir generando sus sucesores mientras no ocurran dos casos:

- Que el nodo seleccionado sea el nodo final, por lo que termina
- Que la cola se quede vacía ya que el nodo está aislado.

## 8. Tabla de resultados

Instancia	N	M	Vo	Vd	Camino	Distancia	N. generados	N. inspeccionados
Grafo1-1	15	38	1	8	1-2-4-12-5-13-8	43	39	31
Grafo1-2	15	38	1	8	1-2-4-12-5-13-8	43	39	31
Grafo2-1	15	38	2	14	2-4-12-5-13-8-14	41	33	28
Grafo2-2	15	38	2	14	2-4-12-5-13-8-14	41	39	32
Grafo3-1	20	50	1	15	1-2-4-8-16-15	40	33	27
Grafo3-2	20	50	1	15	1-2-4-8-16-15	40	33	27
Grafo4-1	20	50	10	18	10-5-3-2-4-18	31	20	14
Grafo4-2	20	50	10	18	10-5-3-2-4-18	31	20	15
Grafo5-1	20	44	1	2	1-20-15-3-14-2	540	15	10