Pathfinding en mapas topológicos

Algoritmos de búsqueda en grafos[1]

Un grafo, representa un conjunto de nodos unidos en una red. Si dos nodos están unidos, al viajar de uno a otro se considerara sucesor el nodo al que nos movemos, y predecesor el nodo del que venimos. Además, normalmente existirá un coste vinculado al desplazamiento entre nodos. Un algoritmo de búsqueda tratará de encontrar un camino óptimo entre dos nodos como por ejemplo un camino que minimice el coste de desplazamiento, o el número de pasos a realizar. La principal diferencia entre los algoritmos es la información que guardan acerca del grafo. Algunos de ellos no guardan información alguna, simplemente expanden la búsqueda desde el nodo inicial, hasta que se llega al nodo final, otros guardan el coste de viajar desde el origen hasta ese nodo, o incluso una estimación de lo prometedor que es un nodo para conducir el camino a su objetivo. La expansión de la búsqueda se realiza en forma de árbol. Partiendo del nodo inicial, se extenderá la búsqueda a sus nodos vecinos, de cada uno de estos nodos vecinos, a sus respectivos nodos vecinos, y así hasta que uno de los nodos a los que se expande la búsqueda es el nodo objetivo. Veamos un algoritmo de búsqueda lo suficientemente general para trabajar en la mayoría de los grafos, y que da paso a otros métodos de búsqueda más complejos[2].

Pseudocódigo de búsqueda

```
Búsqueda en grafo ()
Crear dos listas vacías, LAbierta (vacía) y LCerrada (vacía)
Meter el nodo origen O en la lista LAbierta
Repetir
      Si (LAbierta está vacía) entonces
            Devolver ERROR
      Quitar el primer nodo de LAbierta, N, y meterlo en LCerrada
      Si (N es el nodo destino D) entonces
            Devolver N
      Expandir (N) para obtener el conjunto de sucesores
      Para cada (sucesor S de N)
            Si S no está en LAbierta ni en LCerrada entonces
                  Guardar N como predecesor de S
                  Meter S en la lista LAbierta
Hasta llegar al nodo de destino
```

Primero se crean dos listas que almacenarán nodos, la lista LAbierta, que contiene los nodos que se tienen que expandir, y la lista LCerrada que contiene los que ya han sido expandidos. El nodo de origen, O, se mete en la lista LAbierta para ser expandido. En la propagación de la búsqueda se crea un bucle que acabará en dos posibles ocasiones: cuando la lista LAbierta esté vacía o cuando se encuentre el nodo destino D. El bucle consiste en: primero se obtiene el primer nodo de la lista LAbierta. La selección del nodo es arbitraria (en otros algoritmos se cogería un nodo que cumpliera ciertas propiedades). Si esta lista está vacía quiere decir que no quedan más candidatos para la expansión, y aun así no se encontró el destino, el algoritmo ha fallado: ERROR. Por otra parte, si el nodo obtenido de la lista LAbierta es el destino D, el algoritmo habrá acabado. El camino entre el origen y el destino es un conjunto de nodos que se obtendrán guardando cada predecesor del nodo destino. En otro caso, se seguirá expandiendo la

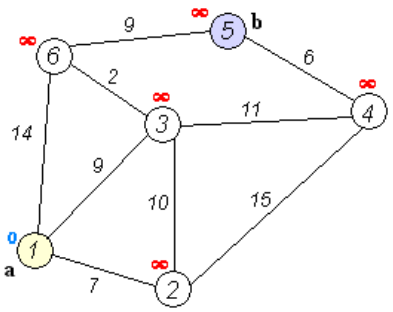
[1] https://es.wikipedia.org/wiki/Algoritmos_de_b%C3%BAsqueda_en_grafos

[2] Para más información sobre tipos de búsquedas se puede ver: Recorrido de árboles https://es.wikipedia.org/wiki/Recorrido_de_%C3%A1rboles

lista, obteniendo una lista de sucesores del nodo actual y guardando en la lista LAbierta aquellos nodos que no estuvieran antes en una lista (aquellos nodos a los que aún no se había llegado). De esta forma, el algoritmo acabará encontrando el destino, o recorriendo todo el grafo.

Algoritmo de Dijkstra[3]

El **algoritmo de Dijkstra**, también llamado **algoritmo de caminos mínimos**, es un algoritmo para la determinación del camino más corto, dado un vértice origen, hacia el resto de los vértices en un grafo que tiene pesos en cada arista.



La idea consiste en ir explorando todos los caminos más cortos que parten del vértice origen y que llevan a todos los demás vértices; cuando se obtiene el camino más corto desde el vértice origen hasta el resto de los vértices que componen el grafo, el algoritmo se detiene[4]. Se trata de una especialización de la búsqueda de costo uniforme y, como tal, no funciona en grafos con aristas de coste negativo (al elegir siempre el nodo con distancia menor, pueden quedar excluidos de la búsqueda nodos que en próximas iteraciones bajarían el costo general del camino al pasar por una arista con costo negativo) [Figura 1].

Pseudocódigo

Teniendo un grafo dirigido ponderado de N nodos no aislados, sea x el nodo inicial. Un vector D de tamaño N guardará al final del algoritmo las distancias desde x hasta el resto de los nodos.

```
1. Inicializar todas las distancias en D con un valor infinito
   relativo, ya que son desconocidas al principio, exceptuando la de
   x, que se debe colocar en 0, debido a que la distancia de x a x
   sería 0.
2. Sea a = x (Se toma a como nodo actual.)
3. Se recorren todos los nodos adyacentes de a, excepto los nodos
   marcados. Se les llamará nodos no marcados vi.
4. Para el nodo actual, se calcula la distancia tentativa desde dicho
   nodo hasta sus vecinos con la siguiente fórmula: dt(vi) = Da +
   d(a,vi). Es decir, la distancia tentativa del nodo 'vi' es la
   distancia que actualmente tiene el nodo en el vector D más la
   distancia desde dicho nodo 'a' (el actual) hasta el nodo vi. Si la
   distancia tentativa es menor que la distancia almacenada en el
   vector, se actualiza entonces el vector con esta distancia
   tentativa. Es decir, si dt(vi) < Dvi → Dvi = dt(vi)
5. Se marca como completo el nodo a.
6. Se toma como próximo nodo actual el de menor valor en D (puede
   hacerse almacenando los valores en una cola de prioridad) y se
   regresa al paso 3, mientras existan nodos no marcados.
```

Una vez terminado al algoritmo, el vector D estará completo[5].

---

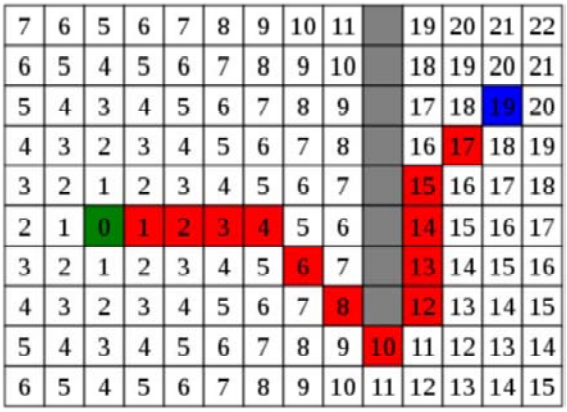
[3] Kikipedia: https://es.wikipedia.org/wiki/Algoritmo_de_Dijkstra; YouTube: https://www.youtube.com/watch?v=dS1Di2ZHl4k; https://www.youtube.com/watch?v=fgdCNuGPJnw

[4] Se puede ver un ejemplo de Algoritmo de Dijkstra en https://es.wikipedia.org/wiki/Anexo:Ejemplo_de_Algoritmo_de_Dijkstra

[5] En https://es.wikipedia.org/wiki/Algoritmo_de_Dijkstra hay descripciones de este algoritmo en pseudocódigo y una implementación en Java

Algoritmo de búsqueda A*[6]

El **algoritmo de búsqueda A*** (pronunciado "A asterisco", "A estrella" o "Astar" en inglés) se clasifica dentro de los algoritmos de búsqueda en grafos[7]. El algoritmo A* encuentra, siempre y cuando se cumplan unas determinadas condiciones, el camino de menor coste entre un nodo origen y uno objetivo [Figura 2[8]].



Pseudocódico del Algoritmo A*

```
1. LAbierta es una lista de nodos ordenada de forma ascendente en
   función del factor F de los nodos.
2. Añadimos el nodo origen a LAbierta.
3. Cogemos el primer elemento de LAbierta, lo sacamos y lo insertamos
   en la lista LCerrada.
4. Seleccionamos los nodos adyacentes al nodo extraído. Para cada uno
   de estos nodos:
   A) Si es el nodo destino, hemos terminado. Para obtener el camino
   recorremos inversamente la lista hasta llegar al origen.
   B) Si el nodo no es accesible lo ignoramos.
   C) Si el nodo ya está en la lista LCerrada, lo ignoramos.
   D) Si el nodo está en la LAbierta, comprobamos si su nueva G es
   mejor que la actual, en cuyo caso recalculamos los factores y
   ponemos como padre del nodo al nodo extraído. En caso de que no sea
   mejor, lo ignoramos.
   E) Al resto de los nodos adyacentes les asignamos como padre el
   nodo extraído y recalculamos los factores. Después los añadimos a
   la lista abierta.
5. Ordenamos la lista abierta.
6. Volvemos al paso 2.
```

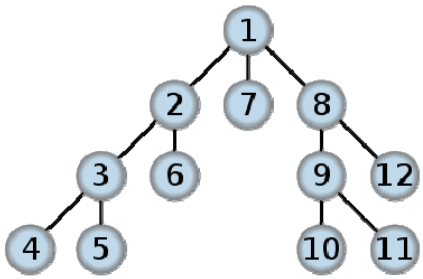
[6] https://es.wikipedia.org/wiki/Algoritmo_de_b%C3%BAsqueda_A*

[7] En https://www.lanshor.com/pathfinding-a-estrella/ hay una demostración de pathfinding mediante A* con mapas

[8] Tomada de CC BY-SA 3.0, https://commons.wikimedia.org/w/index.php?curid=589791]

Búsqueda en profundidad[9]

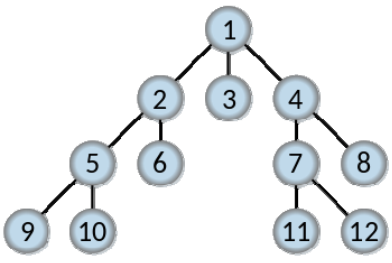
Una **Búsqueda en profundidad** (en inglés DFS o *Depth First Search*) es un algoritmo de búsqueda no informada[10] utilizado para recorrer todos los nodos de un grafo de manera ordenada, pero no uniforme. Su funcionamiento consiste en ir expandiendo todos y cada uno de los nodos que va localizando, de forma recurrente, en un camino concreto. Cuando ya no quedan más nodos que visitar en dicho camino, regresa (Back tracking), de modo que repite el mismo proceso con cada uno de los hermanos del nodo ya procesado [Figura 3].




```
  { /* dev objetivo*/
     Pila lista_nodos;
     Boolean acabar = false;
     lista_nodos.apilar (A.nodo_raiz);
     while ((!lista_nodos.esVacia()) AND (!acabar)) do
     {
        nodo_actual = lista_nodos.cima();
        lista_nodos.desapila();
        if (esObjetivo (nodo_actual) ) then
           {
                acabar = true;
           }
        else
           {
                 nodo_actual.expandir();  //se  añaden  los  hijos  del
nodo actual
                lista_nodos.apilarMuchos
(nodos_expandidos_de_nodo_actual);
           }
     }
     return nodo_actual;
  }
```

Búsqueda en anchura[11]

Una b**úsqueda en anchura** (en inglés *BFS - Breadth First Search*) es un algoritmo de búsqueda no informada[10] utilizado para recorrer o buscar elementos en un grafo. Intuitivamente, se comienza en la raíz y se exploran todos los vecinos de este nodo. A continuación para cada uno de los vecinos se exploran sus respectivos vecinos adyacentes, y así hasta que se recorra todo el árbol. [Figura 4[12]]



---

[9] Kikipedia: https://es.wikipedia.org/wiki/B%C3%BAsqueda_en_profundidad; YouTube: https://www.youtube.com/watch?v=iaBEKo5sM7w

[10] Los métodos de búsqueda no informados o ciegos son estrategias de búsqueda en las cuales se evalúa el siguiente estado sin conocer a priori si este es mejor o peor que el anterior.

[11] https://es.wikipedia.org/wiki/B%C3%BAsqueda_en_anchura

[12] Tomada de Alexander Drichel - Trabajo propio, CC BY 3.0, https://commons.wikimedia.org/w/index.php?curid=3786735

```
BFS(grafo G, nodo_fuente s)
  {
     // recorremos todos los vértices del grafo inicializándolos a
NO_VISITADO,
     // distancia INFINITA y padre de cada nodo NULL
     for u ∈ V[G] do
     {
        estado[u] = NO_VISITADO;
        distancia[u] = INFINITO; /* distancia infinita si el nodo no
es alcanzable */
        padre[u] = NULL;
     }
     estado[s] = VISITADO;
     distancia[s] = 0;
     padre[s] = NULL;
     CrearCola(Q); /* nos aseguramos que la cola está vacía */
     Encolar(Q, s);
     while !vacia(Q) do
     {
        // extraemos el nodo u de la cola Q y exploramos todos sus
nodos adyacentes
        u = extraer(Q);
        for  v ∈ adyacencia[u]  do
        {
           if estado[v] == NO_VISITADO then
           {
                estado[v] = VISITADO;
                distancia[v] = distancia[u] + 1;
                padre[v] = u;
                Encolar(Q, v);
           }
        }
     }
  }
```

Comparación y selección de algoritmo de búsqueda

Tabla 1: Comparación y selección de algoritmo de búsqueda

Algoritmo	**Dijkstra**	**A***	**Búsqueda en profundidad**	**Búsqueda en anchura**
Comple-jidad				
Ventajas				
Inconve-nientes				
Aplicabi-lidad en pathfin-ding				
¿Qué algorimo habéis selecciona do y por qué?				

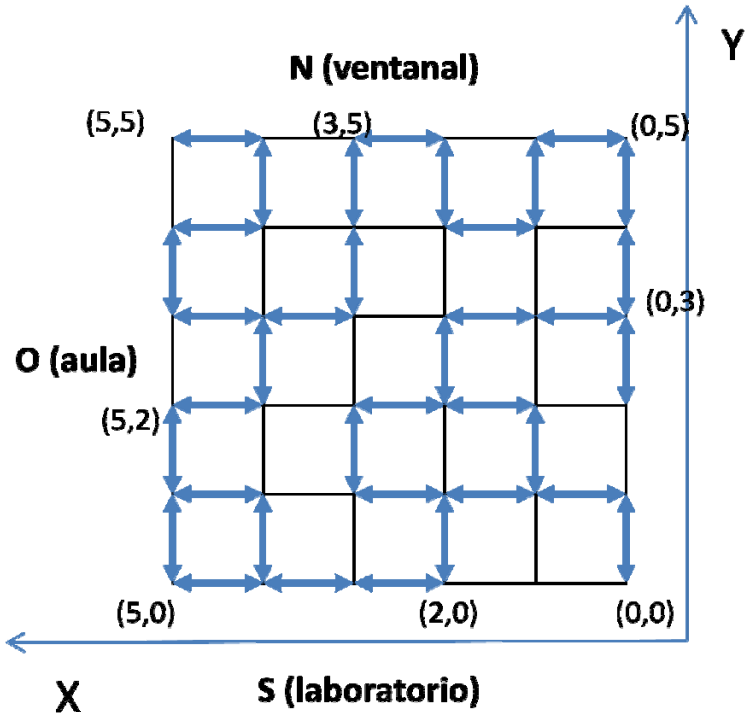
Uso de algoritmos de búsqueda en la Práctica 5

Mapa

El mapa está contenido en un fichero XML con la siguiente estructura:

```
<Laberinto nombre="Laberinto" dim_x="n" dim_y="m">
      // en nuestro caso n=6, m=6
  <Nodo x="a" y="b">
  <Camino dir="norte" />
  <Camino dir="sur" />
  <Camino dir="este" />
  <Camino dir="oeste" />
  </Nodo>…
 </Laberinto>
```

El mapa sólo contiene los caminos existentes. Como los caminos son bidireccionales, están presentes en los nodos de ambos extremos.



Estructuras en C para trabajar con mapas

Los laberintos se componen de nodos:

```
struct nodo{
      int x;
      int y;
      nodo *Norte;
      nodo *Este;
      nodo *Oeste;
      nodo *Sur;
};
```

El camino recorrido por el robot es una lista enlazada

```
struct camino{
      nodo        *nodo_actual;
      camino      *siguiente;
      camino      *anterior;
};
```

Class Laberinto

La clase laberinto contiene las siguientes definiciones y métodos.

```
class Laberinto {
    private:
        string nombre;
        int dim_x;
        int dim_y;
    // struct nodo **matriz;
    // La posición actual del robot se representa usando un nodo
        struct robot{
            nodo *nodo_actual;
            char orientacion;
        };
        struct robot iCreate;
        struct camino *path;
        struct camino *first; // primer elemento de la lista
```

Métodos de uso interno para cargar el XML y rellenar la matriz de nodos

```
        void cargarLaberinto(const char* pFilename);
        void procesarNodos( TiXmlHandle * hParent);
        void procesarCaminos( TiXmlHandle *hNodo, int pos_x,
            int pos_y);
```

Método para situar el robot en un nodo concreto del laberinto

```
        void situarRobot(int pos_x, int pos_y, char orientacion);
```

Actualiza el recorrido del robot en el laberinto

```
        void modificarCamino(robot *aux);
        bool visitado(nodo *proximo, camino *first_visited);
```

Añade un nodo nuevo a un camino

```
        void modificarCamino(camino** cam, nodo **aux);
    public:
      struct nodo **matriz; // Cambiado para que el ejemplo funcione
      Laberinto(void);
      Laberinto(const char* pFilename);
      Laberinto(const char* pFilename, int x, int y,
            char orientacion);
      ~Laberinto(void);
```

Función a completar en la práctica

```
        camino*  encontrarCamino(int x_orig, int y_orig, int x_dest,
            int y_dest);
```

Para obtener la posición y orientación del robot

```
        void getPosRobot(int *pos_x, int *pos_y, char * orientacion);
```

Métodos para consultar si el robot puede avanzar en una dirección

```
        bool esSurLibre();
        bool esNorteLibre();
        bool esEsteLibre();
        bool esOesteLibre();
```

Métodos para cambiar la orientación del robot en el laberinto

```
        void orientarRobotNorte();
        void orientarRobotSur();
        void orientarRobotEste();
        void orientarRobotOeste();
```

Métodos para mover el robot en el laberinto de un nodo a otro nodo adyacente

```
        int avanzaNorte();
        int avanzaSur();
        int avanzaEste();
        int avanzaOeste();
```

Métodos para visualizar la información del laberinto

```
        void imprimirLaberinto();
        void imprimirCamino(camino *first_element);
        void imprimirCaminoRobot();
```

Se puede ver un ejemplo de cómo funcionan las clases y métodos añadidos para representar mapas en un programa en el ejemplo contenido en el fichero “src_mapping.zip”. El algoritmo principal del ejemplo se encuentra en el fichero *Principal.cpp* y usa varias funciones encapsuladas en la clase *Laberinto.h/cpp*. El programa muestra una representación del laberinto del laboratorio y una representación gráfica del robot (>>>) se mueve a través de él mediante los comandos: *a(vanza), n(orte), e(este), w(oeste), s(ur), Imprimir y salir*.

Programación requerida para la práctica 5

Para cambiar el comportamiento del robot y ajustarlo al laberinto hay que modificar los ficheros:

- Laberinto.h y Laberinto.cpp,
- ControlRobot.h y ControlRobot.cpp
- Principal.cpp

Modificar Principal.cpp para que resuelva los problemas con los mapas.

```
int main(int argc, char * argv[])
{
    Laberinto *maze = new Laberinto("../xml/Laberinto.xml");

    // Generamos una solucion entre dos puntos origen [0,0] y destino [5,5]
    camino *solucion = maze->encontrarCamino(0,0,5,5);

    ControlRobot robot;

    robot.cargarMapa(maze);
    robot.recorrerCamino(solucion);

    robot.inicializacion();

    while(!robot.condicionSalida()){ // <-- Mientras no hayamos completado el "camino"
        robot.leerSensores();       // <-- Detectamos cruces
        robot.logicaEstados();      // <-- Decidimos que hacer en cada cruce
        robot.moverActuadores();    // <-- Nos orientamos en el laberinto
        robot.imprimirInfo();
    }

    robot.finalizacion();

    return 0;
}
```

Laberinto.h/Laberinto.cpp

En las clases **Laberinto.h/Laberinto.cpp** hay que completa el método:

```
camino*  encontrarCamino (int x_orig, int y_orig, int x_dest,
            int y_dest);
```

Para ello se puede usar el método:

```
void modificarCamino(camino** cam, nodo **aux);
```

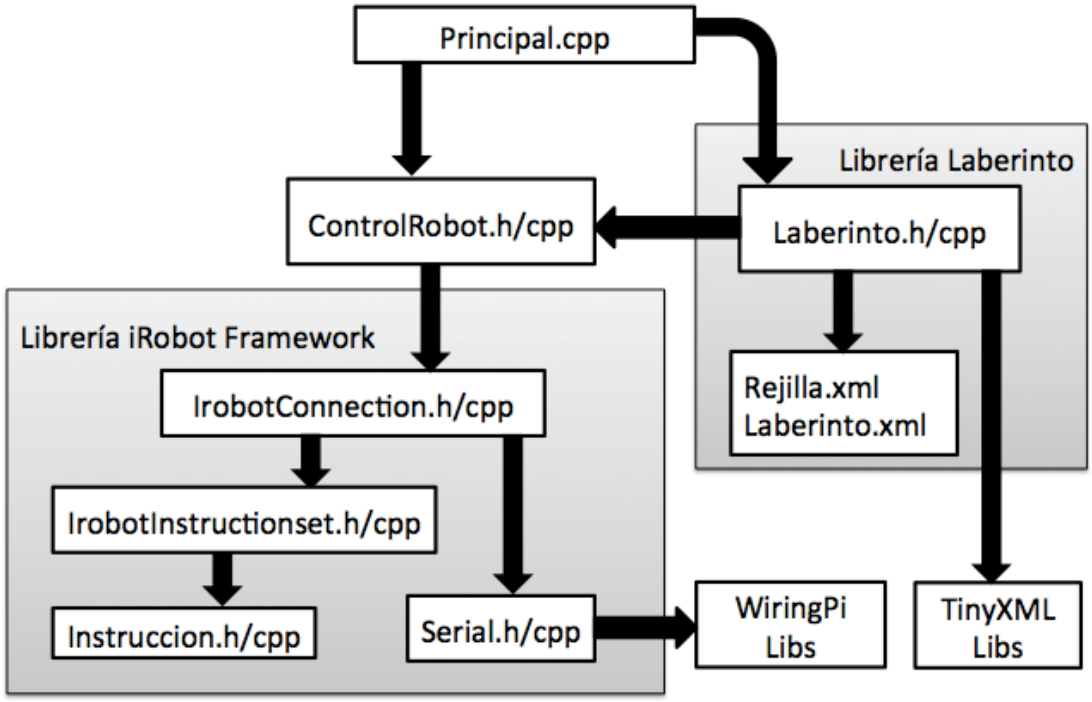
En las clases **ControlRobot.h/ControlRobot.cpp** tenemos que crear los métodos:

```
void cargarMapa(Laberinto * lab);
void recorrerCamino(camino * cam);
```

```
/**
*   Método que obtiene el camino entre un nodo de entrada [x_orig, y_orig] y un nodo de
*   salida [x_dest, y_dest].
*       int x_orig: Posición origen del robot en la Matriz del laberinto
*       int y_orig: Posición origen del robot en la Matriz del laberinto
*       int x_dest: Posición destino del robot en la Matriz del laberinto
*       int y_dest: Posición destino del robot en la Matriz del laberinto
*       return camino*: Apuntador a una estructura tipo camino con la solucion
*/
camino*  Laberinto::encontrarCamino(int x_orig, int y_orig, int x_dest, int y_dest){
    struct camino* solucion  = (struct camino *) malloc (sizeof(camino));

    // ... A COMPLETAR EN LA PRÁCTICA

    return solucion;
}

/**
*   Función que añade un nodo nuevo a un camino
*       camino **cam: Puntero doble a la estructura camino que se va a modificar
*       nodo **aux: Puntero doble al nodo que se quiere añadir en el camino
*/
void Laberinto::modificarCamino(camino** cam, nodo **aux){

    struct camino * siguiente = (struct camino *) malloc (sizeof(camino));

    siguiente->nodo_actual = *aux;
    siguiente->anterior = *cam;
    siguiente->siguiente = NULL;

    (*cam)->siguiente = siguiente;
    *cam = siguiente;
}
```

programar en C el algoritmo seleccionado haciendo uso de las estructuras presentadas

Apéndice 1. Clases y Librerías



Nombre archivo	Función
Principal	Contiene el main con el bucle principal y las llamadas para cargar un laberinto en memoria y comunicarlo con el control del robot. Por defecto tiene un ejemplo que permite mover una representación del robot sobre un mapa virtual en pantalla. Modificaciones: Adaptarlo a los requisitos de la práctica
Laberinto	Carga los mapas en xml y proporciona métodos para crear un camino en un laberinto y recorrerlo. Modificaciones: Completar el método *encontrarCamino* para buscar un camino entre dos nodos del laberinto
ControlRobot	Clase que contiene los métodos que forman parte del bucle principal del robot. Modificaciones: Añadir nuevos métodos para poder usar el laberinto y los caminos en el bucle principal Actualizar la programación de los métodos para que recorran un camino
IRobotConnection	Clase usada para interactuar con el iRobot Create
IRobotInstructionSet	Clase con todas las definiciones de instrucciones permitidas por el iRobot Create
Instrucción	Clase con la definición genérica de instrucción
Serial	Crea una conexión usando la línea serie, envía y recibe datos de ella.