

Inteligencia Artificial

Algoritmos de búsqueda

*Búsqueda A**

Encuentra la salida del laberinto con el menor coste

Profesora: Belén Melián Batista

Alumno: Alejandro Melián Lemes

Índice

Lectura del LaberintoPágina 3

Clase Nodo Página 4

2.1 Cálculo de $f(n)$ Página 4

*2.2 Cálculo de $f(n)$
con heurística alternativa* Página 5

Clase Maze Página 6

*3.1 Algoritmo A^** Página 7

3.2 Almacenamiento del Camino Página 9

Tablas de Resultados Página 10

Referencias Página 14

Lectura del Laberinto

El código se ha desarrollado en lenguaje C++.

En primer lugar, para la lectura del laberinto se ha utilizado la estructura de datos de una matriz, que se implementa con un vector. Dichos ficheros con la determinada implementación se han reutilizado de la asignatura “Algoritmos y Estructura de Datos”, siendo los nombres de los archivos “matrix_t.hpp” y “vector_t.hpp”. Además, se ha establecido la matriz sobre la que se trabajará como una matriz de enteros. De esta manera, se hace un uso efectivo de memoria dado que los obstáculos definidos como “1”s nunca generarán un objeto Nodo en memoria.

```
int main(int argc, char *argv[]) {
    std::string infile = argv[1];
    matrix<int> camino;
    int cont_line = 0, num_fil, num_col, lec_fila = 1, lec_col = 1, opcion;
    int lec_entrada, lec_salida;
    bool indicar_teclado = false, euclides = false;
    std::ifstream file(infile);
    std::string lectura;
    Maze laberinto;
    std::cout << "¿Desea indicar la salida y entrada del laberinto por teclado? (1) Sí (2) No" << std::endl;
    std::cin >> opcion;
    if (opcion == 1) {
        indicar_teclado = true;
    }
    if (file.is_open()) {
        while (file >> lectura) {
            if (cont_line == 0) {
                num_fil = std::stoi(lectura);
            } else if (cont_line == 1) {
                num_col = std::stoi(lectura);
            } else if (cont_line == 2) {
                camino.resize(num_fil, num_col);
            }
            if (cont_line > 1) {
                if (lectura[0] - '0' == 3 && indicar_teclado == false) {
                    laberinto.set_entrada(lec_fila, lec_col);
                    camino.at(lec_fila, lec_col) = lectura[0] - '0';
                }
                if (lectura[0] - '0' == 4 && indicar_teclado == false) {
                    laberinto.set_salida(lec_fila, lec_col);
                    camino.at(lec_fila, lec_col) = lectura[0] - '0';
                }
                if (lectura[0] - '0' != 3 && lectura[0] - '0' != 4) {
                    camino.at(lec_fila, lec_col) = lectura[0] - '0';
                }
                if (lectura[0] - '0' == 3 && indicar_teclado == true) {
                    camino.at(lec_fila, lec_col) = 1;
                }
                if (lectura[0] - '0' == 4 && indicar_teclado == true) {
                    camino.at(lec_fila, lec_col) = 1;
                }
            }
        }
    }
}
```

Código de la función ‘main’ del programa donde se realiza la lectura del laberinto.

Además, como se aprecia, es en la función ‘main’ donde se realiza la recogida de opciones. Como seleccionar el tipo de heurística, o si se desea indicar la salida y entrada por teclado.

Clase Nodo

La clase Nodo representa la casilla dentro del laberinto, y es la responsable del cálculo de su $f(n)$, $h(n)$ y $g(n)$. Se podría decir que su identificación distintiva son las coordenadas “i” y “j” en el laberinto.

```
#ifndef NODO_H
#define NODO_H
#include <iostream>
#include <utility>
#include <vector>
#include "maze.hpp"

class Maze;
/// @brief Clase Nodo que refiere a cada casilla dentro del laberinto y que calcula su f(n)
class Nodo {
public:
    Nodo(int coord_i, int coord_j, int is_diag) : coord_i_{coord_i}, coord_j_{coord_j}, is_diag_{is_diag}, funcion_movimiento_{0}, acumulado_propio_{0},
    Nodo() = default;
    void SetPadre(Nodo *nodoprev) { nodopadre_ = nodoprev; }
    Nodo *get_padre() { return nodopadre_; }
    void obtener_fn(const Maze& lab, const int acumulado_previo);
    void obtener_fn_alternativo(const Maze& lab, const int acumulado_previo);
    int get_fn() const { return funcion_movimiento_; }
    int get_gn() { return acumulado_propio_; }
    int get_coord_i() { return coord_i_; }
    int get_coord_j() { return coord_j_; }
    bool operator()(Nodo* other, Nodo* other_node) { return other->get_fn() > other_node->get_fn(); }

private:
    int funcion_movimiento_; // f(n) y valor identificativo del nodo
    Nodo *nodopadre_; //Nodo Padre
    int coord_i_, coord_j_; //Coordenadas del nodo en el laberinto
    int is_diag; //Entero para comprobar si el nodo ha sido accedido diagonalmente (0 no, 1 diagonal)
    int acumulado_propio_; //g(n) coste acumulado del nodo
};

#endif
```

Clase Nodo que representa una casilla dentro del laberinto y se encarga del cálculo de **su** $f(n)$

Además, cada Nodo tiene un atributo puntero “nodopadre_” que es útil a la hora de encontrar el camino al final del algoritmo. Dicho puntero apunta al nodo anterior dentro del camino. Por lo que el nodo raíz (la casilla de entrada) tendrá su atributo apuntando a NULL.

Cálculo de $f(n)$

Tal como establece el enunciado, la función de movimiento para cada casilla se realizará mediante el cálculo $f(n) = g(n) + h(n)$.

Se ha implementado el cálculo de la siguiente manera:

```
/// @brief Método que calcula el h(n) {Manhattan} y g(n) de un nodo y obtiene el f(n) asociándolo al atributo del nodo
/// @param lab El propio laberinto que contiene las coordenadas de la entrada y salida cruciales para el cálculo de h(n)
/// @param acumulado_previo El g(n) del nodo padre
void Nodo::obtener_fn(const Maze &lab, const int acumulado_previo) {
    int h_n = 0, g_n = 0;
    if (lab.salida_.first > coord_i_) {
        for (int i = coord_i_; i < lab.salida_.first; i++) {
            h_n += 3;
        }
    } else if (lab.salida_.first < coord_i_) {
        for (int i = coord_i_; i > lab.salida_.first; i--) {
            h_n += 3;
        }
    }
    if (lab.salida_.second > coord_j_) {
        for (int i = coord_j_; i < lab.salida_.second; i++) {
            h_n += 3;
        }
    } else if (lab.salida_.second < coord_j_) {
        for (int i = coord_j_; i > lab.salida_.second; i--) {
            h_n += 3;
        }
    }
    if (is_diag_ == 1) {
        g_n = acumulado_previo + 7;
    } else if (is_diag_ == 0) {
        g_n = acumulado_previo + 5;
    }
    acumulado_propio_ = g_n;
    funcion_movimiento_ = h_n + g_n;
}
```

El método “obtener_fn” recibe como parámetros el laberinto (estructura de datos que se comentará más adelante) y el $g(n)$ (“acumulado previo”) del nodo padre, es decir, de la casilla anterior en el camino.

Para la heurística de Manhattan $\{h(n)\}$ se ha optado por desarrollarla de una manera alternativa, en vez de con la fórmula matemática. Tratando de dos bucles, uno para las coordenadas “i” y otro para las coordenadas “j”. Siendo el bucle desde i la coordenada del nodo en las filas hasta la coordenada de la casilla de salida en las filas, se suma 3 unidades. Ídem para la coordenada “j” en las columnas.

Y para el coste acumulado $g(n)$ simplemente se realiza la suma del acumulado del nodo que precede más 5 si el nodo que está calculando su $g(n)$ fue generado en diagonal o vertical/horizontal.

Cálculo de $f(n)$ con heurística alternativa.

La práctica establece que se debe diseñar una heurística alternativa. Es por ello por lo que se ha propuesto la heurística de Euclides, vista en clase de teoría de la asignatura. Es la distancia en línea recta entre dos puntos en un plano.

Si el nodo se encuentra en el punto $(x1, y1)$ y la salida está en el punto $(x2, y2)$, la distancia euclidiana se calcula con la fórmula:

$$\text{Distancia} = \sqrt{(x2 - x1)^2 + (y2 - y1)^2}$$

```
/// @brief Método que obtiene el f(n) mediante la suma de g(n) y h(n) {Euclides} mediante la relación triangular
/// @param lab El propio laberinto del que se sacan las coordenadas del destino (la salida)
/// @param acumulado_previo El g(n) del nodo padre
void Nodo::obtener_fn alternativo(const Maze& lab, const int acumulado_previo) {
    int h_n = 0, g_n = 0;
    int i1 = coord_i;
    int j1 = coord_j;
    int i2 = lab.salida.first;
    int j2 = lab.salida.second;
    // Cálculo de la distancia euclidiana entre el nodo actual y el nodo objetivo
    h_n = std::sqrt((i2 - i1) * (i2 - i1) + (j2 - j1) * (j2 - j1));
    if (is_diag_ == 1) {
        g_n = acumulado_previo + 7;
    } else if (is_diag_ == 0) {
        g_n = acumulado_previo + 5;
    }
    acumulado_propio_ = g_n;
    funcion_movimiento_ = h_n + g_n;
}
```

Método que implementa la heurística alternativa (Euclides) y que es análoga a la heurística de Manhattan

Como se observa, se igualan los valores $i1$ y $j1$ a las coordenadas del nodo que está calculando su $f(n)$, mientras que $i2$ y $j2$ toman los valores de las coordenadas de la salida. Para posteriormente, aplicar la fórmula euclidiana. El cálculo de $g(n)$ y $f(n)$ permanecen sin cambios.

Clase Maze

La clase Maze implementa la búsqueda A*.

```
/// @brief Clase Maze que implementa el laberinto, almacenando la matriz con el laberinto, las coordenads de entrada
/// y salida, e implementa el algoritmo A* mediante su método "encontrar_camino"
class Maze {
public:
    Maze() = default;
    Maze( matrix_t<int> &laberinto) : laberinto {laberinto} {}
    void set_matrix(const matrix_t<int> &matriz) { laberinto_ = matriz; }
    void set_entrada(const int i, const int j) {
        entrada_.first = i;
        entrada_.second = j;
    }
    void set_salida(const int i, const int j) {
        salida_.first = i;
        salida_.second = j;
    }
    matrix_t<int>& get_laberinto() { return laberinto_; }
    void encontrar_camino(const bool& euclides);
    void vuelta_atras(const std::vector<Nodo*> nodos_cerrados);
    void encontrar_nodo_abierto(std::priority_queue<Nodo*, std::vector<Nodo*>, Nodo*> &nodos_abiertos_, Nodo* swapnode);
    bool encontrar_nodo_cerrado(std::vector<Nodo*> &nodos_cerrados, Nodo* find_nodo);
    bool abiertos_repetido(std::priority_queue<Nodo*, std::vector<Nodo*>, Nodo*> &nodos_abiertos_, Nodo* busq_nodo);
    void imprimir_nodos_abiertos(std::priority_queue<Nodo*, std::vector<Nodo*>, Nodo*> &nodos_abiertos_);
    void escritura_a_fichero(const std::vector<Nodo*> nodos_visitados, const std::vector<Nodo*> nodos_generados);
    //Atributos en público para poder ser directamente accedidos por los nodos
    std::pair<int, int> entrada_; // Coordenadas de la entrada (first = i_fila, second = j_col)
    std::pair<int, int> salida_; // Coordenadas de la salida (first = i_fila, second = j_col)

private:
    matrix_t<int> laberinto_; //Matriz con el laberinto
    std::queue<Nodo*> camino;
    int coste_total = 0;
};

#endif
```

Clase Maze que recoge las características del laberinto y que implementa la búsqueda del camino

La clase Maze, almacena la matriz de enteros que contiene el laberinto, a la hora de buscar en la lista de nodos cerrados el camino como una cola de punteros Nodo, el coste final del camino ($g(n)$ del nodo que representa la casilla de salida) y las coordenadas de entrada y la salida mediante un pair de enteros donde el primer elemento representa la coordenada 'i' y el segundo elemento la coordenada 'j'.

Mediante sus métodos implementa el algoritmo A*, como buscar si un nodo ya se encuentra en la lista de nodos cerrados ("encontrar_nodo_cerrado"), o en la de abiertos ("abiertos_repetido"), actualizar el $g(n)$ si fuera posible si el nodo ya se encuentra en la lista de nodos abiertos ("encontrar_nodo_abierto"). Introducir el camino encontrada en la cola de punteros que almacena el camino mediante (vuelta_atras), y por último escribir el resultado en un fichero de texto (Nº de nodos generados, inspeccionados, camino y coste del camino)

Algoritmo A*

Como mencionamos antes, es la clase Maze la que implementa el algoritmo A* y por tanto la búsqueda del camino.

Se define “encontrar_camino” de la siguiente manera:

```
/// @brief Método que implementa el algoritmo A*
/// @param euclides Booleano que identifica si se ha seleccionado la heurística alternativa (true)
void Maze::encontrar_camino(const bool& euclides) {
    int movimiento_i, movimiento_j; // Coordenadas desde las que se estudian los posibles movimientos
    int g_n_salida = 99999; //g(n) de referencia, hasta que se encuentre la salida tendrá un valor arbitrario
    std::vector<Nodo*> nodos_generados; //Vector que almacenará los nodos generados para escribirlos en el fichero
    std::priority_queue<Nodo*, std::vector<Nodo*>, Nodo> nodos_abiertos; //Lista de nodos abiertos
    std::vector<Nodo*> nodos_cerrados; //Lista de nodos cerrados
    Nodo* nodo_partida = new Nodo(entrada.first, entrada.second, 2);
    if (euclides == true) {
        nodo_partida->obtener_fn_alternativo(*this, nodo_partida->get_gn());
    } else {
        nodo_partida->obtener_fn(*this, nodo_partida->get_gn());
    }
    nodos_abiertos.push(nodo_partida);
    nodos_generados.emplace_back(nodo_partida);
    while (!nodos_abiertos.empty() && nodos_abiertos.top()->get_fn() < g_n_salida) { // Mientras la lista de nodos abiertas no esté vacía
        Nodo* iterator_nodo = nodos_abiertos.top();
        if (iterator_nodo->get_coord_i() == salida.first && iterator_nodo->get_coord_j() == salida.second) {
            g_n_salida = iterator_nodo->get_gn();
        }
        //Cambio de Marcha
        movimiento_i = iterator_nodo->get_coord_i();
        movimiento_j = iterator_nodo->get_coord_j();
        for (int mov_horiz = -1; mov_horiz <= 1; mov_horiz++) { // Bucle de movimiento (izquierda, derecha)
            for (int mov_vert = -1; mov_vert <= 1; mov_vert++) { // Bucle de movimiento (arriba, abajo)
                if (movimiento_i + mov_vert <= laberinto.get_m() && // Condicional para no salir de los lim. del laberinto
                    movimiento_i + mov_vert > 0 && movimiento_j + mov_horiz <= laberinto.get_n() && movimiento_j + mov_horiz > 0) {
                    if (laberinto.at(movimiento_i + mov_vert, movimiento_j + mov_horiz) == 0) {
                        laberinto.at(movimiento_i + mov_vert, movimiento_j + mov_horiz) = 4; // Si se encuentra camino
                        if ((mov_horiz == 1 && mov_vert == 1) ||
                            (mov_horiz == 1 && mov_vert == -1) ||
                            (mov_horiz == -1 && mov_vert == 1) || (mov_horiz == -1 && mov_vert == -1)) { //Para averiguar si es diagonal
                            Nodo* newnodo = new Nodo(movimiento_i + mov_vert, movimiento_j + mov_horiz, 1);
                            if (euclides == true) {
                                newnodo->obtener_fn_alternativo(*this, iterator_nodo->get_gn());
                            } else {
                                newnodo->obtener_fn(*this, iterator_nodo->get_gn());
                            }
                        }
                    }
                }
            }
        }
    }
}
```

Primera parte de la implementación del Algoritmo A*

El método “encontrar_camino” recibe como parámetro un booleano en el que se establece ‘true’ si se desea usar la heurística de Euclides o false en otro caso.

En primera instancia, se declaran las estructuras de datos que se utilizarán para almacenar la lista de nodos cerrados y abiertos. De manera que:

- La lista de nodos cerrados se implementa como un vector de punteros nodos.
- La lista de nodos abiertos se implementa como una cola de prioridad, de manera que el primer elemento de la cola siempre sea el de menor $f(n)$ tal como establece el algoritmo.

Se parte de la casilla de entrada, por lo que se crea un nodo con las coordenadas de ésta y un 2 en el atributo para identificar si es diagonal o no, y es que el nodo de partida no es ni diagonal ni vertical/horizontal por lo que su $g(n)$ será de 0. Se calcula su $f(n)$ llamando a su miembro obtener_fn (según sea Euclides o Manhattan) y se inserta en la lista de nodos abiertos.

```
int movimiento_i, movimiento_j; // Coordenadas desde las que se estudian los posibles movimientos
int g_n_salida = 99999; //g(n) de referencia, hasta que se encuentre la salida tendrá un valor arbitrario
std::vector<Nodo*> nodos_generados; //Vector que almacenará los nodos generados para escribirlos en el fichero
std::priority_queue<Nodo*, std::vector<Nodo*>, Nodo> nodos_abiertos; //Lista de nodos abiertos
std::vector<Nodo*> nodos_cerrados; //Lista de nodos cerrados
Nodo* nodo_partida = new Nodo(entrada.first, entrada.second, 2);
if (euclides == true) {
    nodo_partida->obtener_fn_alternativo(*this, nodo_partida->get_gn());
} else {
    nodo_partida->obtener_fn(*this, nodo_partida->get_gn());
}
nodos_abiertos.push(nodo_partida);
nodos_generados.emplace_back(nodo_partida);
```

El bucle while que implementa el algoritmo se ejecutará hasta que o bien la lista de nodos abiertos esté vacía o el nodo que se inspecciona tiene mayor $f(n)$ que $g(n)$ el nodo de salida, lo que significará que no existe ningún otro camino óptimo.

Así pues, se establecen las coordenadas de movimiento al nodo que se está inspeccionando y se recorren las 8 direcciones posibles a partir de éstas. Produciéndose en estos recorridos la generación de nodos adyacentes a la casilla Nodo que se está estudiando, sin embargo, hay que controlar si esta generación se produce diagonalmente o vertical/horizontal, por lo que se establece un if/else en el que lo único que difiere es el valor del atributo si es diagonal en la inicialización del objeto puntero Nodo.

```
while (!nodos_abiertos.empty() && nodos_abiertos.top()->get_fn() < g_n_salida) { // Mientras la lista de nodos abiertos no esté vacía
    Nodo* iterator_nodo = nodos_abiertos.top(); //Nodo que se está inspeccionando
    if (iterator_nodo->get_coord_i() == salida_.first && iterator_nodo->get_coord_j() == salida_.second) {
        g_n_salida = iterator_nodo->get_gn();
    }
    //Cambio de Marcha
    movimiento_i = iterator_nodo->get_coord_i();
    movimiento_j = iterator_nodo->get_coord_j();
    for (int mov_horiz = -1; mov_horiz <= 1; mov_horiz++) { // Bucle de movimiento (izquierda, derecha)
        for (int mov_vert = -1; mov_vert <= 1; mov_vert++) { // Bucle de movimiento (arriba, abajo)
            if (movimiento_i + mov_vert <= laberinto.get_m() && // Condicional para no salir de los lím. del laberinto
                movimiento_i + mov_vert > 0 && movimiento_j + mov_horiz <= laberinto.get_n() && movimiento_j + mov_horiz > 0) {
                if (laberinto.at(movimiento_i + mov_vert, movimiento_j + mov_horiz) == 0) {
                    laberinto.at(movimiento_i + mov_vert, movimiento_j + mov_horiz) == 4) { // Si se encuentra camino
                        if ((mov_horiz == 1 && mov_vert == 1) ||
                            (mov_horiz == 1 && mov_vert == -1) ||
                            (mov_horiz == -1 && mov_vert == 1) || (mov_horiz == -1 && mov_vert == -1)) { //Para averiguar si es diagonal
                            Nodo* newnodo = new Nodo(movimiento_i + mov_vert, movimiento_j + mov_horiz, 1); //Se genera el nodo
                            if (euclides == true) {
                                newnodo->obtener_fn_alternativo(*this, iterator_nodo->get_gn()); //f(n) con Euclides
                            } else {
                                newnodo->obtener_fn(*this, iterator_nodo->get_gn()); //f(n) con Manhattan
                            }
                            newnodo->SetPadre(iterator_nodo); //Se establece el puntero del nodo padre al nodo que se está inspeccionando
                            if (!encontrar_nodo_cerrado(nodos_cerrados, newnodo) && !abiertos_repetido(nodos_abiertos, newnodo)) { //Si no se encuentra en ninguna
                                nodos_abiertos.push(newnodo);
                                nodos_generados.emplace_back(newnodo);
                            }
                            if (abiertos_repetido(nodos_abiertos, newnodo)) { //Si se encuentra en la lista de nodos abiertos
                                encontrar_nodo_abierto(nodos_abiertos, newnodo); //Se intenta actualizar su g(n)
                            }
                        }
                    }
                }
            }
        }
    }
}
```

Imagen del código que muestra el bucle While , el estudio de las direcciones y el como se realiza la generación de un Nodo.

Como se aprecia en la imagen anterior, la generación de un nodo se implementa como:

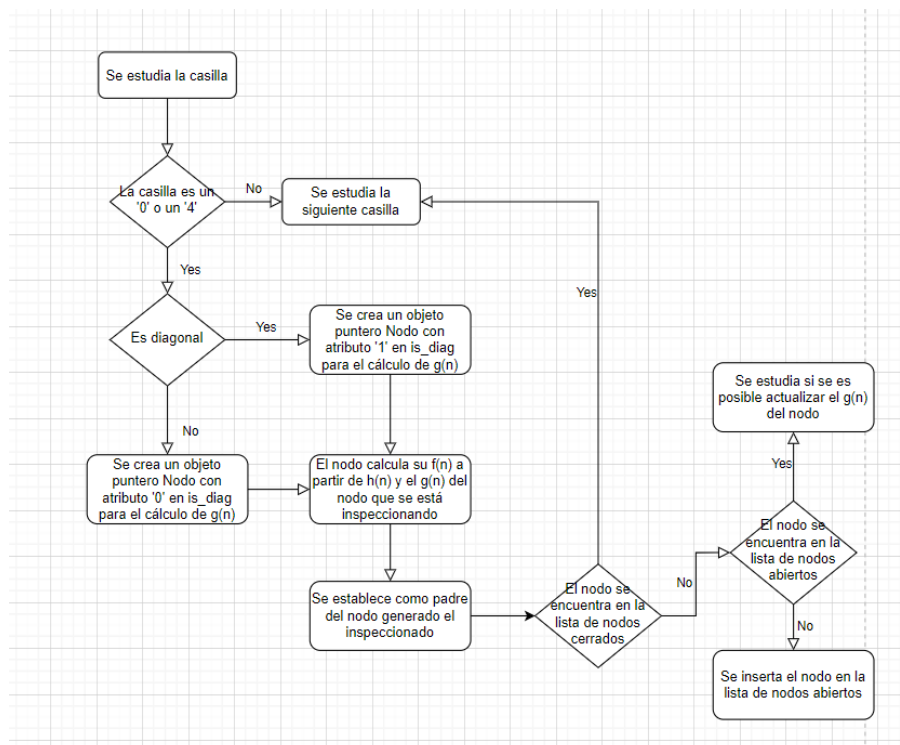


Diagrama de flujo de la generación de un Nodo en el código.

Tras acabar las generaciones, se introduce el nodo inspeccionado en el vector de cerrados

Almacenamiento del camino.

Tras finalizar el algoritmo necesitamos identificar la casilla Nodo de salida dentro de la lista de nodos cerrados y recorrer los padres de cada Nodo. Esto se implementa en el método de Maze “vuelta_atras”.

```
/// @brief Método que busca el camino generado estudiando los padres desde la salida hasta la entrada
/// @param nodos_cerrados La lista de nodos cerrados donde se encuentra la solución
void Maze::vuelta_atras(const std::vector<Nodo*> nodos_cerrados) {
    Nodo* nodo_salida = nullptr;
    for (int i = 0; i < nodos_cerrados.size(); i++) {
        if (nodos_cerrados[i]->get_coord_i() == salida_.first && nodos_cerrados[i]->get_coord_j() == salida_.second) {
            nodo_salida = nodos_cerrados[i];
            break;
        }
    }
    if (nodo_salida == nullptr) {
        std::cout << "No se encontró el nodo de salida en los nodos cerrados." << std::endl;
        return;
    }
    // Recorre los padres desde el nodo de salida hasta el nodo de entrada
    Nodo* nodo_actual = nodo_salida;
    coste_total = nodo_actual->get_gn();
    while (nodo_actual != nullptr) {
        int i = nodo_actual->get_coord_i();
        int j = nodo_actual->get_coord_j();
        laberinto_.at(i, j) = 5;
        camino.push(nodo_actual);
        // Si el padre es nullptr, hemos llegado al nodo de entrada
        if (nodo_actual->get_padre() == nullptr) {
            std::cout << "Llegamos al nodo de entrada." << std::endl;
            break;
        }
        // Avanza al nodo padre
        nodo_actual = nodo_actual->get_padre();
    }
    laberinto_.at(entrada_.first, entrada_.second) = 3;
    laberinto_.at(salida_.first, salida_.second) = 4;
}
```

Código que implementa el recorrido de los padres y la inserción de estos en la cola que almacena el camino.

Primero se encuentra el Nodo con las coordenadas de la salida del laberinto dentro del vector de los nodos cerrados. Si dicho Nodo tiene su puntero de padre en NULL significa que el laberinto no tiene salida.

El coste total del camino se iguala al $g(n)$ del nodo salida, y entonces se recorren los padres iterativamente de cada nodo. En cada iteración se establece el valor de las casillas en la coordenada del nodo que se está recorriendo a un ‘5’ que representará el camino en la matriz.

Cuando se identifique el nodo con el puntero padre a NULL significará que hemos llegado a la entrada del laberinto.

Tablas de resultados

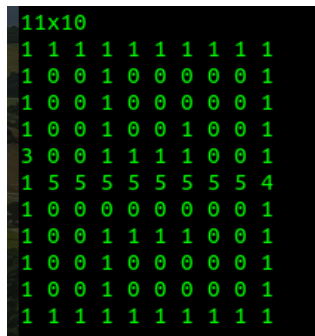
A continuación, se representarán mediante una tabla los resultados de la ejecución del programa con distintas instancias de laberintos, además sobre un mismo laberinto distinta entrada/salida. Hay que tener en cuenta que se trata la primera columna con un índice '1', ídem para la primera fila.

Búsqueda A*. Función heurística $h(\cdot)$ Manhattan

Instancia	n	m	S	E	Camino	Coste	Nodos Generados	Nodos Inspeccionados
M1	11	10	(6,10)	(5,1)	10	47	33	23

Nodos Generados: (5, 1), (4, 2), (5, 2), (6, 2), (7, 2), (5, 3), (6, 3), (7, 3), (4, 3), (6, 4), (7, 4), (6, 5), (7, 5), (6, 6), (7, 6), (3, 2), (3, 3), (8, 2), (8, 3), (6, 7), (7, 7), (5, 8), (6, 8), (7, 8), (5, 9), (6, 9), (7, 9), (2, 2), (2, 3), (6, 10), (8, 8), (9, 2), (9, 3)

*Camino: (6, 10), (6, 9), (6, 8), (6, 7), (6, 6), (6, 5), (6, 4), (6, 3), (6, 2), (5, 1)



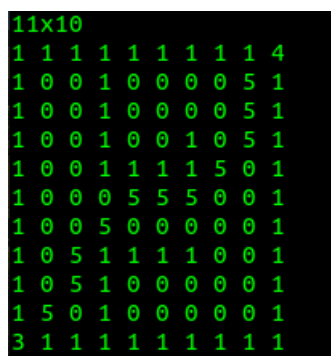
Resultado gráfico de la Instancia M1 (1ª Ejecución)

Búsqueda A*. Función heurística $h(\cdot)$ Manhattan

Instancia	n	m	S	E	Camino	Coste	Nodos Generados	Nodos Inspeccionados
M1	11	10	(1,10)	(11,1)	13	74	44	39

Nodos Generados: (11, 1), (10, 2), (9, 2), (9, 3), (10, 3), (8, 2), (8, 3), (7, 2), (7, 3), (7, 4), (6, 3), (6, 4), (6, 5), (7, 5), (6, 6), (7, 6), (6, 2), (5, 3), (6, 7), (7, 7), (5, 2), (4, 2), (4, 3), (5, 8), (6, 8), (7, 8), (4, 8), (4, 9), (5, 9), (6, 9), (8, 8), (3, 2), (3, 3), (7, 9), (3, 8), (3, 9), (3, 7), (8, 9), (2, 2), (2, 3), (2, 8), (2, 9), (2, 7), (1, 10)

Camino: (1, 10), (2, 9), (3, 9), (4, 9), (5, 8), (6, 7), (6, 6), (6, 5), (7, 4), (8, 3), (9, 3), (10, 2), (11, 1)



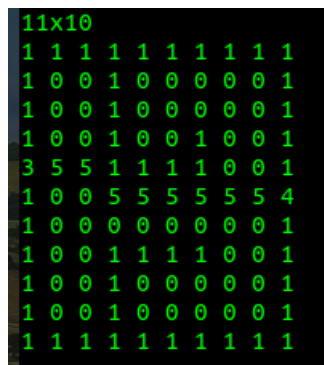
Resultado gráfico de la Instancia M1 (2ª Ejecución)

Búsqueda A*. Función heurística $h(\cdot)$ Euclides

Instancia	n	m	S	E	Camino	Coste	Nodos Generados	Nodos Inspeccionados
M1	11	10	(6,10)	(5,1)	10	47	44	36

Nodos generados: (5, 1), (4, 2), (5, 2), (6, 2), (4, 3), (5, 3), (6, 3), (3, 2), (3, 3), (7, 2), (7,3), (6, 4), (7, 4), (2, 2), (2, 3), (8, 2), (8, 3), (6, 5), (7, 5), (9, 2), (9, 3), (6, 6), (7, 6), (10, 2), (10, 3), (6, 7), (7, 7), (5, 8), (6, 8), (7, 8), (8, 8), (5, 9), (6, 9), (7, 9), (4, 8), (4, 9), (8, 9), (9, 7), (9, 8), (9, 9), (6, 10), (3, 7), (3, 8), (3, 9)

Camino: (6, 10) , (6, 9) , (6, 8) , (6, 7) , (6, 6) , (6, 5) , (6, 4) , (5, 3) , (5, 2) , (5, 1)



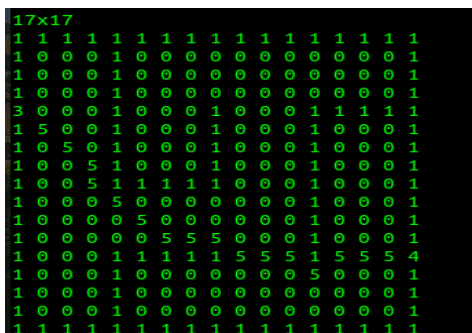
Resultado gráfico de la Instancia M1 (3 ejecución)

Búsqueda A*. Función heurística $h(\cdot)$ Manhattan

Instancia	n	m	S	E	Camino	Coste	Nodos Generados	Nodos Inspeccionados
M2	17	17	(13,17)	(5,1)	18	103	113	91

Nodos generados: (5, 1), (4, 2), (5, 2), (6, 2), (7, 2), (5, 3), (6, 3), (7, 3), (4, 3), (8, 2), (8, 3), (6, 4), (7, 4), (8, 4), (5, 4), (9, 3), (9, 4), (4, 4), (9, 2), (10, 3), (10, 4), (10, 5), (11, 4), (11, 5), (10, 6), (11, 6), (10, 2), (11, 3), (12, 5), (12, 6), (10, 7), (11, 7), (12, 7), (3, 2), (3, 3), (11, 2), (12, 4), (11, 8), (12, 8), (10, 8), (3, 4), (12, 3), (11, 9), (12, 9), (13, 4), (12, 2), (13, 3), (10, 9), (11, 10), (12, 10), (13, 10), (13, 2), (14, 9), (14, 10), (12, 11), (13, 11), (14, 11), (14, 3), (14, 4), (10, 10), (14, 2), (11, 11), (2, 2), (2, 3), (2, 4), (9, 10), (12, 12), (13, 12), (14, 12), (10, 11), (9, 11), (11, 12), (14, 13), (10, 12), (9, 12), (15, 10), (15, 11), (15, 12), (8, 10), (8, 11), (15, 9), (15, 3), (15, 4), (15, 13), (15, 2), (8, 12), (13, 14), (14, 14), (15, 14), (12, 14), (12, 15), (13, 15), (14, 15), (14, 8), (15, 8), (15, 15), (16, 11), (16, 12), (16, 13), (12, 16), (13, 16), (14, 16), (16, 10), (15, 16), (7, 10), (7, 11), (7, 12), (16, 14), (16, 3), (16, 4), (16, 9), (13, 17), (16, 2),

Camino: (13, 17) , (13, 16) , (13, 15) , (13, 14) , (14, 13) , (13, 12) , (13, 11) , (13, 10) , (12, 9) , (12, 8) , (12, 7) , (11, 6) , (10, 5) , (9, 4) , (8, 4) , (7, 3) , (6, 2) , (5, 1)



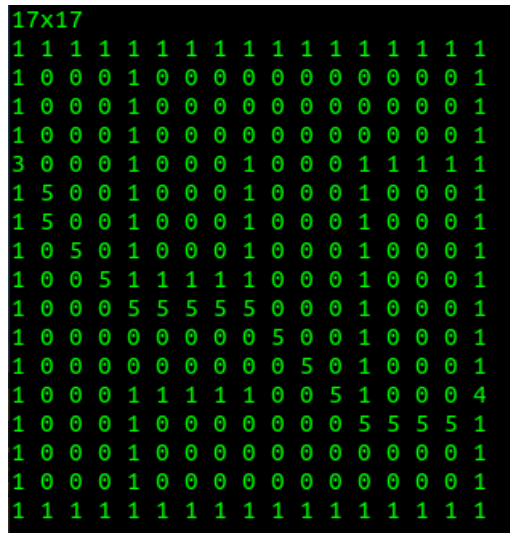
Representación gráfica Instancia M2 (1ª Ejecución)

Búsqueda A*. Función heurística $h(\cdot)$ Euclides

Instancia	n	m	S	E	Camino	Coste	Nodos Generados	Nodos Inspeccionados
M2	17	17	(13,17)	(5,1)	18	103	153	137

Nodos generados: (5, 1), (4, 2), (5, 2), (6, 2), (4, 3), (5, 3), (6, 3), (7, 2), (7, 3), (3, 2), (3, 3), (4, 4), (5, 4), (6, 4), (7, 4), (3, 4), (8, 2), (8, 3), (8, 4), (2, 2), (2, 3), (2, 4), (9, 2), (9, 3), (9, 4), (10, 2), (10, 3), (10, 4), (10, 5), (11, 2), (11, 3), (11, 4), (11, 5), (10, 6), (11, 6), (12, 2), (12, 3), (12, 4), (10, 7), (11, 7), (12, 5), (12, 6), (12, 7), (13, 2), (13, 3), (13, 4), (10, 8), (11, 8), (12, 8), (10, 9), (11, 9), (14, 2), (14, 3), (14, 4), (12, 9), (9, 10), (10, 10), (11, 10), (15, 2), (15, 3), (15, 4), (12, 10), (13, 10), (9, 11), (10, 11), (11, 11), (16, 2), (16, 3), (12, 11), (16, 4), (8, 10), (8, 11), (13, 11), (9, 12), (10, 12), (11, 12), (14, 9), (14, 10), (14, 11), (12, 12), (8, 12), (13, 12), (7, 10), (7, 11), (7, 12), (14, 12), (15, 9), (15, 10), (15, 11), (15, 12), (6, 10), (6, 11), (14, 13), (14, 8), (15, 8), (6, 12), (15, 13), (16, 9), (16, 10), (16, 11), (16, 12), (16, 13), (5, 10), (5, 11), (16, 8), (13, 14), (14, 14), (15, 14), (14, 7), (15, 7), (5, 12), (16, 14), (16, 7), (13, 15), (14, 15), (15, 15), (4, 9), (4, 10), (4, 11), (12, 14), (12, 15), (16, 15), (14, 6), (15, 6), (4, 12), (13, 16), (14, 16), (15, 16), (16, 6), (4, 13), (16, 16), (12, 16), (3, 9), (3, 10), (3, 11), (11, 14), (11, 15), (3, 12), (11, 16), (13, 17), (3, 8), (4, 8), (5, 8), (3, 13), (3, 14), (4, 14), (10, 14), (10, 15), (10, 16), (2, 9), (2, 10), (2, 11), (2, 12),

Camino: (13, 17), (14, 16), (14, 15), (14, 14), (14, 13), (13, 12), (12, 11), (11, 10), (10, 9), (10, 8), (10, 7), (10, 6), (10, 5), (9, 4), (8, 3), (7, 2), (6, 2), (5, 1)



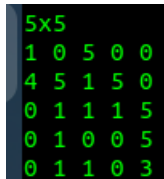
Representación Gráfica de la Instancia M2 (2º ejecución)

Búsqueda A*. Función heurística $h(\cdot)$ Manhattan

Instancia	n	m	S	E	Camino	Coste	Nodos Generados	Nodos Inspeccionados
M3	5	5	(1,2)	(5,5)	7	36	15	14

Nodos generados: (5, 5), (4, 4), (5, 4), (4, 5), (4, 3), (3, 5), (2, 4), (2, 5), (1, 3), (1, 4), (1, 5), (1, 2), (2, 2), (2, 1), (3, 1)

Camino: (2, 1), (2, 2), (1, 3), (2, 4), (3, 5), (4, 5), (5, 5)



5x5
1 0 5 0 0
4 5 1 5 0
0 1 1 1 5
0 1 0 0 5
0 1 1 0 3

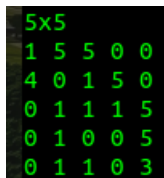
Representación gráfica de la instancia M3 (1ª ejecución)

Búsqueda A*. Función heurística $h(\cdot)$ Euclides

Instancia	n	m	S	E	Camino	Coste	Nodos Generados	Nodos Inspeccionados
M3	5	5	(1,2)	(5,5)	7	36	15	14

Nodos generados: (5, 5), (4, 4), (5, 4), (4, 5), (3, 5), (4, 3), (2, 4), (2, 5), (1, 4), (1, 5), (1, 3), (1, 2), (2, 2), (2, 1), (3, 1)

Camino: (2, 1), (1, 2), (1, 3), (2, 4), (3, 5), (4, 5), (5, 5)



5x5
1 5 5 0 0
4 0 1 5 0
0 1 1 1 5
0 1 0 0 5
0 1 1 0 3

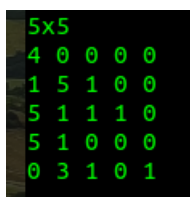
Representación gráfica de la instancia M3 (2ª ejecución)

Búsqueda A*. Función heurística $h(\cdot)$ Euclides

Instancia	n	m	S	E	Camino	Coste	Nodos Generados	Nodos Inspeccionados
M3	5	5	(1,1)	(5,2)	5	26	15	12

Nodos generados: (5, 2), (4, 1), (5, 1), (4, 3), (3, 1), (4, 4), (5, 4), (2, 2), (3, 5), (4, 5), (1, 1), (1, 2), (1, 3), (2, 4), (2, 5)

Camino: (1, 1), (2, 2), (3, 1), (4, 1), (5, 2)



5x5
4 0 0 0 0
1 5 1 0 0
5 1 1 1 0
5 1 0 0 0
0 3 1 0 1

Representación gráfica de la instancia M3 (3ª ejecución)

Referencias

- [1] [Wikipedia. Documentación sobre A*](#)
- [2] [Búsqueda de Rutas Implementando A* en un Entorno Virtual 2D](#)
- [3] [Enunciado de la práctica](#)
- [4] [Herramienta para la creación del diagrama de flujo.](#)