ADA

PRACTICA FINAL:

CAMINO DE COSTE MÍNIMO CON RAMIFICACIÓN Y PODA

Martín Aznar García

1. - ESTRUCTURA DE DATOS

1.1.- NODOS

```
// Representación de cada nodo en el laberinto
struct Nodo {
    int costeEntrada; // Valor del nodo (coste de entrada)
    bool valido;      // Indica si se puede pasar por el nodo
    int heuristica;     // Estimación heurística (distancia restante)
    int costeAcumulado; // Mejor coste conocido hasta este nodo
    bool enCaminoOptimo; // Marca si el nodo está en el camino óptimo
};
```

Los nodos se encuentran representados mediante una estructura “Nodo” la cual cuenta con los atributos:

- costeEntrada : Int : Valor de entrada.
- valido : Bool : Indica si el nodo es válido, si su valor es distinto a 0.
- heuristica : Int : Almacen del valor de la cota optimista del nodo.
- costeAcumulado : Int : Coste mínimo para llegar hasta el nodo.
- enCaminoOptimo : Bool : Indica si el nodo se encuentra en el camino más óptimo.

1.2.-NODOS VIVOS

```
// Estructura que guarda el estado para los Nodos utilizados en la rama actual
struct NodoVivo {
    int fila; // Posición actual en el laberinto en el eje X
    int columna; // Posición actual en el laberinto en el eje Y
    vector<int> pasosRealizados; // Pasos dados hasta ahora
    Nodo* nodo; // Nodo actual en el laberinto
    int costeHastaAhora;
    int costeEstimadoTotal; // costeHastaAhora + heurística

    bool operator>(const NodoVivo& otro) const {
        return costeEstimadoTotal > otro.costeEstimadoTotal;
    }
};
```

Los nodos vivos són representados mediante una estructura “NodoVivo” la cual cuenta con los siguientes atributos:

- fila : Int : Posición del Nodo en la coordenada X.
- columna : Int : Posición del Nodo en la coordenada Y.
- nodo : Nodo* :Almacena una referencia del nodo en el que nos encontramos.
- pasosRealizados : vector<Int> : Almacena el recorrido seguido hasta este nodo.
- operador> : Este nos será útil más adelante para determinar el orden en el que recorrer los nodos.
- costeHastaAhora : Int : Coste necesario para llegar al nodo.
- costeEstimadoTotal : Int : Coste total estimado

2.- MECANISMOS DE PODA

2.1.- PODA DE NODOS NO FACTIBLES

```
if (!laberinto[nx][ny].valido) {
    totalNoFactibles++;
    continue;
}
```

Para el descarte de nodos por no ser factibles se dará el caso mediante la comprobación de los nodos adyacentes al nodo actual de manera que si alguno de estos tiene el indicador “valido” a false esto indicaría que se trata de un nodo con valor de entrada a 0 y por tanto no es un nodo posible de acceder.

2.2.- PODA DE NODOS NO PROMETEDORES

```
if (actual.costeHastaAhora >= actual.nodo->costeAcumulado) {
    totalNoPrometedores++;
    continue;
}
actual.nodo->costeAcumulado = actual.costeHastaAhora;
```

Se compara el valor del coste del nodo actual con el que tiene el nodo en caso de que sea mayor o igual a este se descartará debido a que no es el camino más corto para llegar a ese nodo por tanto ya exite una solución mejor optimizada.

```
int nuevoCoste = actual.nodo->costeAcumulado + laberinto[nx][ny].costeEntrada;
int estimacion = nuevoCoste + laberinto[nx][ny].heuristica;

if (estimacion >= mejorCosteEncontrado) {
    totalNoPrometedores++;
    continue;
}

if (laberinto[nx][ny].costeAcumulado <= nuevoCoste) {
    totalNoPrometedores++;
    continue;
}
```

Para el descarte de nodos no prometedores se dará el caso cuando el “nuevoCoste”, siendo éste la suma del coste hasta llegar a el nodo actual y el valor de entrada del nodo siguiente, y la “heurística” del siguiente nodo, correspondiendo esta a la cota optimista del nodo siguiente:

En el caso de que la suma de nuevoCoste y heurística sea mayor o igual al menor coste encontrado o el nuevoCoste sea mayor o igual al coste para llegar a este nodo calculado en otra rama anterior. Esto indicaría que el nodo aunque sí es factible este no sería capaz de encontrar un coste menos para resolver el laberinto.

```
if(!prometedorDescartado) {
    totalPrometedoresDescartados++;
}
```

De otra manera una vez recorridos todos los nodos adyacentes si ninguno de los nodos adyacentes no visitados hasta el momento es válido entonces se descarta este nodo como no prometedor ya que este no tiene salida hacia ningún nodo.

3.- COTA PESIMISTA Y OPTIMIZACIÓN

3.1.- COTA PESIMISTA INICIAL(INICIALIZACIÓN)

```
cotaPesimistaInicial = 0;
for (int i = 0; i < ancho; i++) {
    for (int j = 0; j < alto; j++) {
        int valor;
        fichero >> valor;

        Nodo nodo;
        nodo.costeEntrada = valor;
        nodo.valido = (valor != 0);
        nodo.enCaminoOptimo = false;
        nodo.costeAcumulado = COSTE_MAXIMO;
        nodo.heuristica = max(ancho - 1 - i, alto - 1 - j);

        if (nodo.valido)
            cotaPesimistaInicial += valor;

        laberinto[i][j] = nodo;
    }
}
fichero.close();
```

La cota pesimista inicial es calculada al inicio del programa, en la lectura de datos del fichero de manera que cada vez que un nodo sea válido, que su valor sea distinto a 0, se incrementará. De este modo la cota pesimista inicial representará el número máximo de nodos válidos en el laberinto.

3.2.- COTA PESIMISTA DEL RESTO DE NODOS

```
if (f == ancho - 1 && c == alto - 1) {
    totalHojas++;
    if (actual.nodo->costeAcumulado + actual.nodo->costeEntrada <= mejorCosteEncontrado) {
        mejorCosteEncontrado = actual.nodo->costeAcumulado + actual.nodo->costeEntrada;
        rutaOptima = actual.pasosRealizados;
        actualizacionesSolucion++;
    }
    if (mejorCosteEncontrado < cotaPesimistaInicial) {
        cotaPesimistaInicial = mejorCosteEncontrado;
        actualizacionesCotaPesimista++;
    }
    continue;
}
```

La cota pesimista representará el mayor coste posible para recorrer el laberinto, está coincidirá con el mejor coste encontrado, ya que ambos se modifican con el mismo valor al llegar al final del laberinto, siempre y cuando el coste del camino sea menor al ya establecido.

3.3- COTA SUPERIOR

```
cotaPesimistaInicial = 0;
for (int i = 0; i < ancho; i++) {
    for (int j = 0; j < alto; j++) {
        int valor;
        fichero >> valor;

        Nodo nodo;
        nodo.costeEntrada = valor;
        nodo.valido = (valor != 0);
        nodo.enCaminoOptimo = false;
        nodo.costeAcumulado = COSTE_MAXIMO;
        nodo.heuristica = max(ancho - 1 - i, alto - 1 - j);

        if (nodo.valido)
            cotaPesimistaInicial += valor;

        laberinto[i][j] = nodo;
    }
}
fichero.close();
```

La cota superior se almacena en el atributo “heurística” del nodo, calculado mediante la distancia Manhattan:

“*max(alto - 1 - fila , ancho - 1 - columna)*”

La cota es calculada en el momento de introducir los datos al vector<Nodo> laberinto. Representando la distancia diagonal desde la posición hasta el final.

4-. OTROS MEDIOS EMPLEADOS PARA ACELERAR LA BÚSQUEDA

```
priority_queue<NodoVivo, vector<NodoVivo>, greater<NodoVivo>> pq;
```

Para poder acelerar el la búsqueda del mejor camino posible utilizo un priority_queue el cual se establece con la etiqueta greater<NodoVivo> de esta manera se llama al operador> determinando el orden de priority_queue de manera que se ordene de menor a mayor por coste del nodo. Así al acceder al top siempre se encontrará el nodo de menor coste.

```
// Movimientos en 8 direcciones (N, NE, E, SE, S, SW, W, NW)
const int dx[8] = {-1, -1, 0, 1, 1, 1, 0, -1};
const int dy[8] = {0, 1, 1, 1, 0, -1, -1, -1};
const int codigosDireccion[8] = {4, 3, 5, 2, 6, 8, 1, 7}; // Como teclado numérico
```

Determinar el orden de recorrido de nodos adyacentes empezando por la esquina inferior derecha, seguido por derecha, abajo, esquina superior derecha, esquina inferior izquierda, arriba y esquina superior izquierda.

5.- ESTUDIO COMPARATIVO DE DISTINTAS ESTRATEGIAS DE BÚSQUEDA

NO IMPLEMENTADO

6.- TIEMPOS DE EJECUCIÓN

Fichero de test	Tiempo(ms)
100-bb.maze	1.491
200-bb.maze	4.4696
300-bb.maze	41.0223
500-bb.maze	269.9159
700-bb.maze	102.2894
900-bb.maze	223.3134
k01-bb.maze	193.8651
k02-bb.maze	?
k03-bb.maze	927.7118
k05-bb.maze	3161.3112
k10-bb.maze	?

Representación	Coste Temporal	Coste Espacial	Legibilidad
struct	Muy bajo	Muy bajo	Alta
tuple	Bajo	Muy bajo	Baja
map/unordered_map	Medio/Alto (O(log n)/O(1))	Alto (hash table o árbol)	Alta
vector<vector<>>	Alto	Alto	Media

Decante por el lado de struct como forma de almaenar los atributos de un nodo.

Unorder_map : solo almacenamiento de nodos posibles, no almacenamiento de valores a 0.

vacetor