Sirvent Navarro, Álvaro DNI.74531046Y

1. ESTRUCTURA DE DATOS

- Nodo.

En primer lugar, vamos a pasar a hablar sobre el contenido que he incluido en el nodo para la realización correcta del algoritmo.
En mi caso, para llevar a cabo el algoritmo, lo que he hecho ha sido utilizar una tupla que contiene los siguientes valores.

```
37
38  typedef struct nodos miStructDeNodos;
39  typedef tuple<int , miStructDeNodos , vector<pos>> tuplaDeNodos;
40
```

Por lo que, si nos fijamos en lo que se ha incluido en la tupla, podemos observar que nos encontramos en primer lugar, con un entero que será el que me facilite la ordenación en la cola de prioridad, aunque de eso, hablaremos más adelante.
Seguidamente, nos encontramos con un tipo de struct que hemos creado, que contiene tanto la fila y la columna del nodo, como los pasos que hemos tenido que realizar para llegar hasta ese nodo.
Por último, nos encontramos con un vector de otro tipo de struct que me servirá para sacar el camino resultado.

```
17  struct posicion{ //Nos creamos un struct que utilizaremos para la recurs
18      int fila;
19      int columna;
20      int pasos;
21  };
22  typedef struct posicion pos;
23  string camino="";
24  bool arrived = false;
25  bool pasaPor= true;
26
27  int minimo= 10000000;
28
29  vector<pos> noFactibles;
30  vector<pos> explorados;
31  bool opP= false;
32
33  struct nodos{
34      pos nodo;
35      int pasos;
36  };
37
38  typedef struct nodos miStructDeNodos;
```

- **LISTA DE NODOS VIVOS**

En segundo lugar, nos encontramos con la estructura de datos que hemos utilizado para guardar los nodos, en mi caso, me he decidido por utilizar una cola de prioridad, donde hemos incluido la tupla definida en el apartado anterior, viéndose de la siguiente manera.

```
47  //Nos definimos la cola de prioridad de la que vamos a ir sacando nodos para expan
48  priority_queue<tuplaDeNodos, vector<tuplaDeNodos> , LessThanByOpt> miCola;
49  vector<pos> resultado;
50  //int pasos= 0;
```

Por otro lado, para que la cola de prioridad se ordenase por los valores que nosotros queríamos de la tupla, he tenido que añadir un operador dentro de un nuevo struct. Quedándose de la siguiente manera.

```
22  typedef struct posicion pos;
23  string camino="";
24  bool arrived = false;
25  bool pasaPor= true;
26
27  int minimo= 10000000;
28
29  vector<pos> noFactibles;
30  vector<pos> explorados;
31  bool opP= false;
32
33  struct nodos{
34      pos nodo;
35      int pasos;
36  };
37
38  typedef struct nodos miStructDeNodos;
39  typedef tuple<int , miStructDeNodos , vector<pos>> tuplaDeNodos;
40
41  struct LessThanByOpt{
42      bool operator() ( const tuplaDeNodos& t1 , const tuplaDeNodos& t2 ){
43          return get<0>(t1) < get<0>(t2);
44      }
45  };
46
47  //Nos definimos la cola de prioridad de la que vamos a ir sacando nodos para expa
48  priority_queue<tuplaDeNodos, vector<tuplaDeNodos> , LessThanByOpt> miCola;
49  vector<pos> resultado;
```

Por último, en cuanto a la manera que he ordenado los diferentes nodos, he decido emplear la distancia de Chebyshov, no obstante, he implementado otros métodos de ordenación, que son tanto la distancia Euclidea, y el cálculo de la diagonal de un rectángulo, aunque he decidido quedarme con el de Chebyshov dado los resultados que este me ha devuelto en las pruebas.

```
221             }
222             else if(vectorNodos[n2][m2] > actuales && minimo> actuales){
223                 vectorNodos[n2][m2] = actuales;
224                 added++;
225                 miStructDeNodos exp2;
226                 exp2.nodo.fila = n2;
227                 exp2.nodo.columna = m2;
228                 exp2.pasos = actuales;
229                 int res = distanciaCheb(n2, m2 , n-1 ,m-1);
230                 vector<pos> cam;
231                 pos exp3;
232                 exp3.fila = n2 ;
233                 exp3.columna= m2;
234                 cam = posiciones;
235                 cam.push_back(exp3);
236                 miCola.push(tuplaDeNodos( res,exp2, cam));
237                 pos exp;
238                 exp.fila = n2;
239                 exp.columna = m2;
240                 exp.pasos = actuales;
241                 explorados.push_back(exp);
242                 //posiciones.push_back(exp);
243
244             }
245         }
```

```
    int columAct =0 ;
    while(  !miCola.empty()){//Hacemos un bucle que mirará por la fila y la columna actual del nodo que estamos expandiendo
        posicionesIda(n , m , filaAct , columAct , matriz , vectorNodos , camino , actuales );
        miStructDeNodos exp2;
        tuplaDeNodos nd = miCola.top();//MIramos en el nodo que estamos expandiendo todas las posiciones a las que se puede ir, con el fin
        miCola.pop();
        exp2 = get<1>(nd);
        actuales = exp2.pasos;
        filaAct = exp2.nodo.fila;
        columAct = exp2.nodo.columna;
        counter++; //Nodos explorados
    }
    return counter; //DEVOLVEMOS EL NÚMERO DE NODOS EXPLORADOS
```

2.MECANISMOS DE PODA.

PODA DE NODOS NO FACTIBLES

Para podar los nodos no factibles, lo que tenía pensado hacer y que había llegado a implementar, era que se podaran respecto a lo que devolvía el algoritmo voraz en ese punto, claro, esto funciona relativamente bien para laberintos pequeños, pero en el momento en el que se empieza a hacer más grande, me ha sido imposible de mantener esta solución. Es por ello, que para podar los nodos que nos son factibles, utilizo la cota pesimista que he calculado antes de la realización del algoritmo, comparada con la distancia de Chebyshov (Aunque, soy consciente de que esto es lo que se debería utilizar para los casos optimistas) y la suma de los pasos que llevo hasta el momento. Por otro lado, recalcar que aunque he utilizado la distancia de Chebyshov, también he usado la distancia euclidea en las pruebas.

```
void miraPosiciones(int n , int m , int **matriz , int n2 , int m2 , int **vectorNodos , int actuales , vector<pos> posiciones){
    explorando++;
    if( n2<n && n2>=0 && m2 <m && m2>=0 &&minimo>actuales && actuales+ distanciaCheb(n2, m2 , n-1 ,m-1) <best && cotaOpt(n2,m2)<=cotaOpt(n-1,m-1) ){
        if( matriz[n2][m2]==1){
            /*pos exp;
            exp.fila = n2;
```

```
int distanciaEuclidea( int a1 , int a2 , int b1 , int b2){ //Método que me
    return sqrt( pow(a1 - b1,2) + pow(a2 - b2,2));
}

int distanciaCheb(int a1 , int a2, int b1 , int b2){
    return max(abs(a1-b1) , abs(a2-b2));
}
```

PODA DE NODOS NO PROMETEDORES

Para podar los nodos no prometedores, entendiendo por estos aquellos que superan la cota pesimista, pero que son los más alejados de la cota optimista y de la solución que llevamos actualmente.

Para ello, como voy cambiando el valor de “best” que sería lo que me ha servido para el apartado anterior, estaríamos hablando de la misma parte, solo que, para tener también otro mecanismo de poda, lo que he hecho ha sido podar mediante el uso del cálculo de la diagonal, comparando la que tendríamos en el nodo (0,0), con la que nos encontraríamos en el nodo que estamos actualmente.

La diagonal la he calculado de la siguiente manera.

```
//MÉTODO PARA CALCULAR LA COTA OPTIMISTA************************************************
int cotaOpt(int n , int m){
    return sqrt( (n)*(n) + (m)*(m)); // Devolvemos el cálculo de la diagonal del rectángulo como cota optimista
}

```

COTAS PESIMISTAS Y OPTIMISTAS.

NODO INICIAL

Pesimistas: Para el cálculo de la cota pesimista, lo que he hecho ha sido usar el algoritmo voraz que ya tenía implementado de prácticas anteriores, y que como sabemos, no es seguro que este nos devuelva una solución, es por ello, que lo que he hecho ha sido que en el caso de que no se me devuelva una solución, ponerlo como el número total de 1s que podríamos tener en la matriz, es decir, n(fila)*m(columna).

```
    best = vorazGreedy(n , m , matriz , exp2 , cam,copia );
    bool sol = true;
    //Miramos si ha conseguido llegar hasta el final
    if(arrived == false &&pasaPor == false ){
        best = n*m; // Ponemos como cota pesimista como si la matriz fueran todo 1s
        sol = false;
    }else if(arrived == false && pasaPor ==true){
        best =n*m;//Ponemos como cota pesimista como si la matríz fueran todo 1s
        sol= false;
    }
```

Optimista: Para el cálculo de la cota optimista, en este caso, me coincide que la que tengo para el primer nodo, es la misma que nos vamos a encontrar para los nodos posteriores, es decir, la diagonal entre los puntos 0,0 y el final.

Siguientes nodos

Para los siguientes nodos, lo que he hecho ha sido ir cambiando solamente mi cota pesimista, de manera que cada vez que se encuentre una solución mejor que la que tenemos en la cota pesimista, esta se cambie por la nueva solución, de manera que, cada vez los nodos que vamos a ir metiendo en la cola, se disminuirá.

En cuanto a la optimista, la he dejado igual que para el primer nodo.

*Recalcar que mi cota pesimista, es un entero con el nombre de “best”.

```
            actuales++; //Sumamos +1 al actual
            if( n2== n-1 && m2 == m-1) { //CUANDO LLEGA AL FINAL
                pos exp;
                exp.fila = n2;
                exp.columna = m2;
                vector<pos> cam;
                cam = posiciones;
                cam.push_back(exp);
                vectorNodos[n2][m2] = actuales;
                minimo = actuales;
                resultado= cam; //Pasamos el camino
                best = actuales; //CAMBIAMOS EL MEJOR ENCONTRADO (*****COTA PESIMISTA)
                nodosFull++;


            }
```

MEDIOS EMPLADOS PARA ACELERAR LA BÚSQUEDA

En cuanto a los medios que he empleado para acelerar la búsqueda, realmente no he implementado casi ninguno nuevo, salvo por las diferentes cotas que he usado, y el intento de minimizar al máximo posible los nodos expandidos mediante el uso de la diagonal y la distancia de Chebyshov.

```
175 //MÉTODO PARA CALCULAR LA COTA OPTIMISTA*****************************************************
176 int cotaOpt(int n , int m){
177     return sqrt( (n)*(n) + (m)*(m)); // Devolvemos el cálculo de la diagonal del rectángulo como cota optimista
178 }
179
180 int distanciaEuclidea( int a1 , int a2 , int b1 , int b2){ //Método que me sirve para sacar la distancia existente entre un nodo y el final, y así sa
181     return sqrt( pow(a1 - b1,2) + pow(a2 - b2,2));
182 }
183
184 int distanciaCheb(int a1 , int a2, int b1 , int b2){
185     return max(abs(a1-b1) , abs(a2-b2));
186 }
187
188 bool llegado = false;
189 int added = 0 ;
190 int explorando=0;
191 int nodosExpandidos = 0;
192 int noF=0;
193 int noPrometedores= 0 ;
194 int nodosFull= 0;
195 int noMejor= 0;
196 int siProm = 0;
197
198 void miraPosiciones(int n , int m , int **matriz , int n2 , int m2 , int **vectorNodos , int actuales , vector<pos> posiciones){
199     explorando++;
200     if( n2<n && n2>=0 && m2 <m && m2>=0 &&minimo>actuales && actuales+ distanciaCheb(n2, m2 , n-1 ,m-1) <best && cotaOpt(n2,m2)<=cotaOpt(n-1,m-1) ){
201         if( matriz[n2][m2]==1){
202             /*pos exp;
203             exp.fila = n2;
204             exp.columna = m2;
205             posiciones.push_back( exp); *///Metemos la posiciones actual
206             actuales++; //Sumamos +1 al actual
207             if( n2== n-1 && m2 == m-1) { //CUANDO LLEGA AL FINAL
208                 pos exp;
```

ESTUDIO DE COMPARATIVAS DE DISTINTAS ESTRATEGIAS DE BÚSQUEDA

*LAS PRUEBAS PARA LAS DIFERENTES COMPARATIVAS, LAS HE REALIZADO CON EL ARCHIVO 29,8maze, dado como una de las pruebas de las prácticas.

MEDIANTE EL USO DE LA DISTANCIA EUCLIDEA.

```
alu@VDI-Ubuntu-EPS-2016:~/Escritorio/ADA-PRÁCTICA FINAL /entrega-FINAL$ ./maze-b
b -f 29.8maze
Shortest path length= 5866
Explored nodes= 75788365 (Added= 75315215; nonpromising= 473138; nonfactible= 12
1)
Expanded nodes= 75315215
Completed nodes= 2 (Best solution updated= 2)
Promising but dicarded nodes= 429344
Best solution updated from a pessimistic bound= 0
CPU elapsed time= 38323.8 ms.
alu@VDI-Ubuntu-EPS-2016:~/Escritorio/ADA-PRÁCTICA FINAL /entrega-FINAL$
```

```
aPosiciones(int n , int m , int **matriz , int n2 , int m2 , int **vectorNodos , int actuales , vector<pos> po
orando++;
n2<n && n2>=0 && m2 <m && m2>=0 &&minimo>actuales && actuales+ distanciaEuclidea(n2, m2 , n-1 ,m-1) <best && c
if( matriz[n2][m2]==1){
    /*pos exp;
    exp.fila = n2;
    exp.columna = m2;
    posiciones.push_back( exp); *///Metemos la posiciones actual
```

MEDIANTE EL USO DE LA DISTANCIA DE CHEBYSHOV.

```
alu@VDI-Ubuntu-EPS-2016:~/Escritorio/ADA-PRÁCTICA FINAL /entrega-FINAL$ ./maze-b
b -f 29.8maze
Shortest path length= 5866
Explored nodes= 75788365 (Added= 75315215; nonpromising= 473138; nonfactible= 12
)
Expanded nodes= 75315215
Completed nodes= 2 (Best solution updated= 2)
Promising but dicarded nodes= 429344
Best solution updated from a pessimistic bound= 0
CPU elapsed time= 37972.4 ms.
alu@VDI-Ubuntu-EPS-2016:~/Escritorio/ADA-PRÁCTICA FINAL /entrega-FINAL$
```

```
raPosiciones(int n , int m , int **matriz , int n2 , int m2 , int **vectorNodos , int actuales , vector<pos> posiciones){
lorando++;
 n2<n && n2>=0 && m2 <m && m2>=0 &&minimo>actuales && actuales+ distanciaCheb(n2, m2 , n-1 ,m-1) <best && cotaOpt(n2,m2)<=cotaOpt(n
 if( matriz[n2][m2]==1){
     /*pos exp;
     exp.fila = n2;
     exp.columna = m2;
     posiciones.push_back( exp); *///Metemos la posiciones actual
     actuales++; //Sumamos +1 al actual
     if( n2== n-1 && m2 == m-1) { //CUANDO LLEGA AL FINAL
         pos exp;
         exp.fila = n2;
         exp.columna = m2;
         vector<pos> cam;
         cam = posiciones;
```

MEDIANTE EL USO DE LA DIAGONAL (SOLAMENTE)

```
alu@VDI-Ubuntu-EPS-2016:~/Escritorio/ADA-PRÁCTICA FINAL /entrega-FINAL$ ./maze-b
b -f 29.8maze
Shortest path length= 5866
Explored nodes= 75788365 (Added= 75315215; nonpromising= 473138; nonfactible= 12
)
Expanded nodes= 75315215
Completed nodes= 2 (Best solution updated= 2)
Promising but dicarded nodes= 429344
Best solution updated from a pessimistic bound= 0
CPU elapsed time= 38573.5 ms.
alu@VDI-Ubuntu-EPS-2016:~/Escritorio/ADA-PRÁCTICA FINAL /entrega-FINAL$
```

```
es(int n , int m , int **matriz , int n2 , int m2 , int **vectorNodos , int actuales , vector<pos> posic

2>=0 && m2 <m && m2>=0 &&minimo>actuales && actuales+ cotaOpt(n2, m2 ) <best && cotaOpt(n2,m2)<=cotaOpt(
z[n2][m2]==1){
 exp;
ila = n2;
olumna = m2;
iones.push_back( exp); *///Metemos la posiciones actual
```

Soluciones y tiempos de ejecución:

00.8maze= 0ms.

01.8maze = 0.014ms.

02.8maze= 0.043ms.

03.8maze = 0.101ms.

04.8maze= 0ms.

05.8maze= 0.082ms.

06.8maze= 0.067ms.

07.8maze = 0.114ms.

08.8maze= 0.13ms.

09.8maze = 0.141ms.

10.8maze= 0.867ms.

20.8maze = 3.141ms.

21.8maze= 0.213ms.

22.8maze = 0.37ms.

23.8maze = 0.35ms.

24.8maze = 747.641ms.

25.8maze= 159.937ms.

26.8maze = 310.901ms.

27.8maze = 454,825ms.

28.8maze= 2073.71ms.

29.8maze = 40686.7ms.

30.8maze = 15162.2ms.