Instituto Politécnico Nacional

Escuela Superior de Cómputo

Estructuras de Datos

Proyecto

Generación y Resolución de Laberintos

Equipo:

Díaz Urbina Eduardo

Jhovanni Gaspar Vilchis

Introducción

La generación y resolución de laberintos se puede lograr a través de estructuras simples como **pilas** y **árboles binarios**. El presente proyecto tiene como finalidad hacer uso de estas 2 estructuras para crear laberintos, almacenados en un arreglo unidimensional, en los que dados 2 puntos, por ellos pase solo un camino, que se denomina **laberinto perfecto**, y en los que los puntos de partida y salida estén lo más alejados posibles; y para poder resolver estos laberintos haciendo uso solo de pilas.

También tiene la finalidad de almacenar estos laberintos como mapas en archivos de texto para su uso posterior y que no se tenga ningún límite en el tamaño de los laberintos, salvo por limitaciones de memoria. Para esto se hará uso de **colas** en las que se almacenarán momentáneamente los datos del laberinto.

El programa está dividido en tres funcionalidades: la **generación del laberinto**, la **navegación del laberinto** y la **visualización de la solución del laberinto**.

Índice

Justificación

La generación de laberintos como el descrito en la Introducción sería más difícil sin el uso de estructuras como pilas y/o árboles binarios ya que se tendría que estar verificando a cada momento el estado completo del laberinto para verificar que se cumplan con las condiciones descritas y más aún, encontrar el camino más largo tomaría mucho más tiempo. La resolución de laberintos es similar a la generación de laberintos en el uso de pilas, por lo que también se decidió dar cobertura a esto en el proyecto.

Si se quieren guardar y procesar los mapas de laberintos de tamaños arbitrarios en archivos de texto, se tendría que realocar memoria constantemente conforme se leen estos mapas, por lo que una estructura como las colas es de gran ayuda y evita realocar memoria.

Análisis

Se tendría que guardar el laberinto en un arreglo 3d o 2d dependiendo de sí es un laberinto 3d o 2d. Cada entrada sería un **Muro** o un **Pasillo**. Para la generación del laberinto, se puede empezar con uno cuyas entradas sean todas Muros e ir convirtiendo esos Muros en Pasillos, para que se cumpla la condición de ser perfecto se tendrían que recordar todos los muros que ya no pueden derribarse por conectar caminos que crearían bucles. Sin embargo, almacenar los muros haría que la búsqueda de éstos, conforme se avanza, sea más costosa; se podría usar una estructura adecuada para su almacenamiento pero es más sencillo usar información vertida entrada del laberinto y que esta información sea actualizada solo cuando entradas adyacentes sean alteradas.

La visualización del mapa de un laberinto es muy limitada en una pantalla de caracteres, y más aún si se quieren laberintos de tamaños arbitrarios, por lo que si se desea ver la resolución del laberinto se necesita mostrar solo una parte de éste.

Diseño

Descripción General

Para las estructuras se usó una función macro que crea las funciones necesarias en tiempo de compilación para cada tipo de estructura, localizadas en la carpeta estructuras: **estructuras/stack.c** para pilas, **estructuras/queue.c** para colas y **estructuras/tree.c** para los árboles binarios.

La **representación en pantalla** se hizo usando una pantalla temporal (**array 2d char**) en la que se pinta solo una porción del mapa (16 entradas por línea y 4 líneas, siendo cada entrada de 5x5), donde el centro y el piso a pintar se especifica mediante un **struct punto** (ver más abajo).

Para **guardar** el mapa del laberinto, las tres primeras líneas del archivo de texto se usan para especificar la dimensión del mapa, la cuarta y quinta para almacenar los puntos (en 1d) del camino más largo, para **abrir** se lee de acuerdo a como se guardó y hace uso de una **cola de unsigned char**, que conforme va leyendo los valores, los almacena en la cola y después verifica que coincida la dimensión del mapa con los valores obtenidos y vierte las información en las entradas del laberinto.

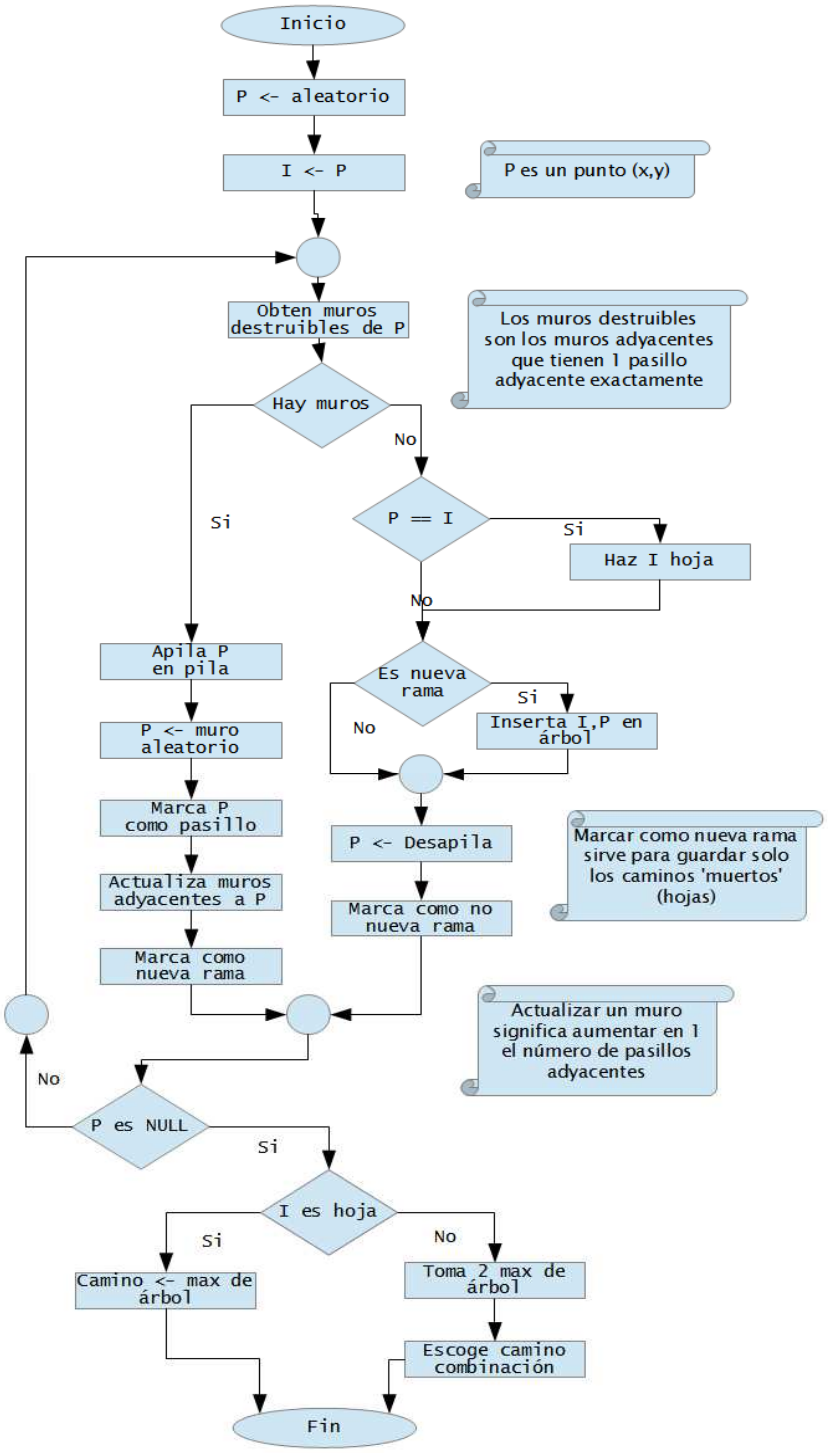
El código para **abrir**, **guardar** y **representar en pantalla** están en el archivo **lectura.c**, junto con algunas funciones para leer de consola.

Se escogió almacenar las entradas del laberinto en un **arreglo dinámico 1d** dada la facilidad de obtener y disponer la memoria dinámica en arreglos simples; aunque el acceso a las entradas sea más complicado, se accedería en 2d **(i,j)->(i+j*lx)** donde lx es la longitud total, también lo hace más fácil de extender a 3d usando la fórmula **(i,j,k)->(i+j*lx*ly+k*ly)**, visto esto en comparación con la dificultad de la creación de arreglos dinámicos 3d.

Las entradas del laberinto son variables **unsigned char** donde constantes enteras identifican a la entrada como Pasillo, Muro/Muro con pasillos adyacentes.

Se usó un **struct punto** que almacena coordenadas 3d, esto para facilitar el movimiento a través del arreglo 1d de entradas, y funciones (en realidad macros) que traducen las coordenadas 3d al arreglo 1d y viceversa. En conjunto se crearon constantes enteras para identificar las direcciones, **Norte, Sur, Este, Oeste, Arriba** y **Abajo** y funciones que alteran el punto de acuerdo a cada dirección. Para escoger el camino más largo se usó un **struct camino** que almacena los puntos de inicio, fin y la longitud del camino. Todo esto se encuentra en **mapa.c**.

La **generación del laberinto** se puede apreciar en el siguiente diagrama de flujo, empezando con un mapa lleno de muros, la pila almacena los puntos y el árbol almacena los caminos, para encontrar el camino más largo dependiendo si el punto donde se empezó es una hoja o no.



Generación de Laberinto

Con este algoritmo ya se puede guardar el mapa en un archivo. La **resolución del laberinto** es muy similar y solo difiere en que se obtienen los pasillos en vez de los muros, se descartan posiciones anteriores para evitar regresarse y cuando se halle un pasillo con múltiples caminos, se escoge una dirección, se apilan el pasillo y las direcciones, esto para cuando se halle al final de un camino, se tome uno de la pila y se dirija a otras direcciones, el algoritmo acaba cuando se llegue a la meta final. El código de ambos algoritmos está en **generar.c**.

Manual de Uso

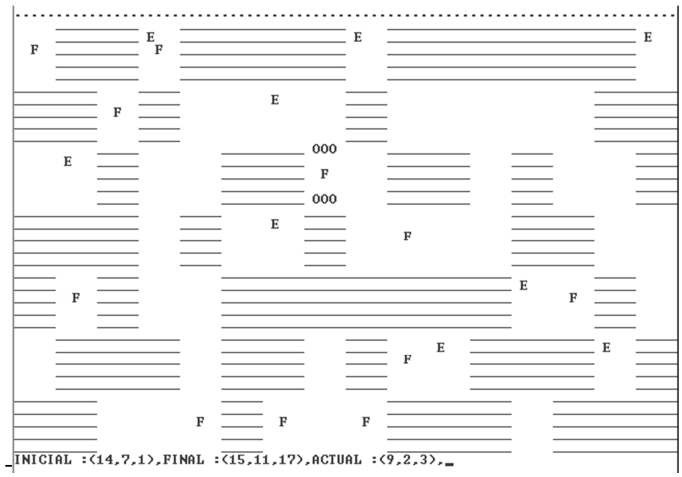
Para su uso, se debe llamar al programa lab.exe desde símbolo de sistema de acuerdo a la funcionalidad deseada:

Para **generar un mapa**, debe llamarse con los argumentos **gen nombre_mapa x y z**, donde **nombre_mapa** es el nombre del archivo donde se guardará el mapa (preferible en archivo txt) y **x**, **y**, **z** son las dimensiones del mapa, lanzará un error en la pantalla si encuentra algún problema con el mapa, cualquier archivo con el mismo nombre lo sobreescribirá. No se recomienda alterar de alguna forma los archivos creados. Ejemplo:

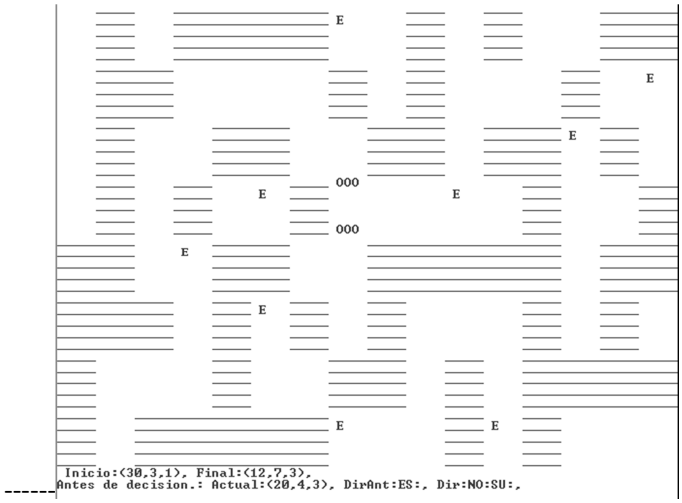
```
C:\Users\Urbo\Desktop\programa>lab gen mapa20_20_20.txt 20 20 2
Creando mapa 'mapa20_20_20.txt'...
Mapa muy chico...

C:\Users\Urbo\Desktop\programa>lab gen mapa20_20_20.txt 20 20 20
Creando mapa 'mapa20_20_20.txt'...
Mapa guardado exitosamente.
```

Para **navegar o inspeccionar un mapa**, debe llamarse con los argumentos **nav nombre_mapa**, se recomienda tener la ventana de símbolo de sistema maximizada. Podrá navegar tecleando las direcciones individuales: **Norte:w, Sur:s, Este:d, Oeste:a, Arriba:e, Abajo:f**, y después dando **ENTER**, el mapa mostrára la posición actual centrada e indicará si hay movimientos hacia Arriba o hacia Abajo o ambos. Si se desea abandonar el programa debe presionar **0** y luego **ENTER**. Ejemplo:

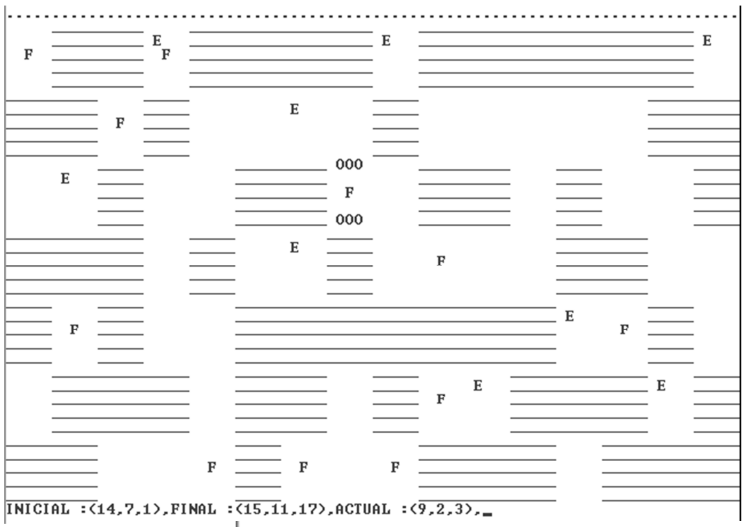


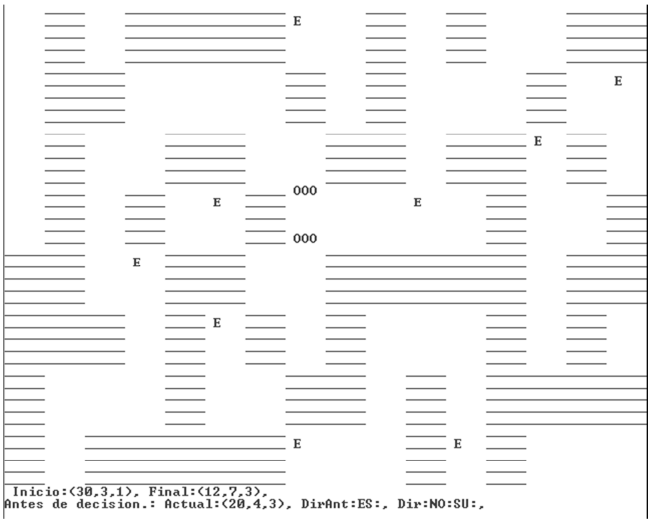
Para **resolver un mapa**, debe llamarse con los argumentos **res nombre_mapa**, para avanzar en cada paso de la resolución, debe presionarse **ENTER**, el programa indicará cuando se llegue a la meta final, después de esto, debe teclearse la palabra **salir** y presionar **ENTER**. Para mapas muy grandes, si en algún momento se desea salir del programa, deben presionarse las teclas **Ctrl+c**. Ejemplo:



Resultados

Capturas de pantalla:






```
C:\Users\Urbo\Desktop\programa>lab gen mapa20_20_20.txt 20 20 2
Creando mapa 'mapa20_20_20.txt'...
Mapa muy chico...

C:\Users\Urbo\Desktop\programà>lab gen mapa20_20_20.txt 20 20 20
Creando mapa 'mapa20_20_20.txt'...
Mapa guardado exitosamente.
```

Ejemplos de mapas generados:

4	30	0100	----------	1101
4	63	0110	1110	0011
4	1011	1101	1011	----------

0101

0000

0101

1100

1011

1110

1010

1011

8

8

8

411

417

01111000

01001011

01011001

11010101

10100111

10111100

10100011

11100010

10000111

10110101

10100110

00110010

01001000

01000011

01011100

00010110

11001000

01010010

00011001

11001010

10101101

11110110

00001011

11101001

10111101

10100110

11100011

00100001

11010001

00001101

11110000

10011111

11000011

01001000

00001010

10011000

01110110

11000011

00001100

11100010

00011110

11010101

10110101

11101101

00000011

10111100

10100111

00101001

11100001

00000101

00001010

00010010

11011100

00000011

10010000

11011111

10001011

11111010

10101011

10101101

10100101

01101001

01011111

01100000

Conclusiones

El uso de las pilas y de los árboles facilitó el algoritmo para general y resolver los laberintos. En cuanto al tamaño de los mapas generados, cuanto más grande sea el mapa, mayor tiempo se tarda en generarlos. También se notó que los mapas generados son muy predecibles, pues se crean caminos muy largos primero antes de hacer las bifurcaciones.

Impresión de Código

queue.c:

```
/*
Necesita la librería: stdlib.h
Para definir una nueva cola, node
debe ser del tipo:
struct node{
  struct node* next;
```

```
  ...
}

Se crean las funciones:
queue_node crear_queue_node();
int       empty_queue_node();
node*     front_node(queue_node);
void      insert_queue_node(node,queue_node);
node*     remove_queue_node(queue_node);
*/
#define DEFINIR_NUEVO_QUEUE( node ) \
typedef struct queue_##node{ \
  node *front; \
  node *back; \
}queue_##node; \
\
queue_##node* crear_queue_##node(){ \
  queue_##node *q = (queue_##node*)malloc(sizeof(queue_##node)); \
\
  if( q == NULL ) return q; \
    q->front = NULL; \
    q->back = NULL; \
  return q; \
} \
\
int empty_queue_##node(queue_##node *q){ \
  return q->front == NULL; \
} \
\
node* front_##node(queue_##node *q){ \
  return q->front; \
} \
\
void insert_queue_##node(node *n,queue_##node *q){ \
  if( n == NULL ) return; \
\
  if(empty_queue_##node(q)) \
    q->front = n; \
  else \
    q->back->next = n; \
  q->back = n; \
  q->back->next = NULL; \
} \
\
node* remove_queue_##node(queue_##node *q){ \
  if(empty_queue_##node(q)) \
    return NULL; \
\
  node *aux = q->front; \
  q->front = q->front->next; \
\
  aux->next = NULL; \
\
  return aux; \
} \
\
void drop_queue_##node(queue_##node **q){ \
  while(!empty_queue_##node(*q)) \
    free(remove_queue_##node(*q)); \
  free(*q);\
} \

int queue_gen = 1;
```

stack.c:

```
/*
Necesita la librería: stdlib.h
Para definir una nueva pila, node
debe ser del tipo:
struct node{
  struct node* next;
  ...
}

Se crean las funciones:
stack_node crear_stack_node();
int       empty_stack_node();
node*     top_node(stack_node);
void      push(node,stack_node);
node*     pop(stack_node);
*/
#define DEFINIR_NUEVO_STACK( node ) \
typedef struct stack_##node{ \
  node *top; \
}stack_##node; \
\
stack_##node* crear_stack_##node(){ \
  stack_##node *s = (stack_##node*)malloc(sizeof(stack_##node)); \
  if( s != NULL ) \
    s->top = NULL; \
  return s; \
} \
\
int empty_stack_##node(stack_##node *s){ \
  return s->top == NULL; \
} \
\
node* top_##node(stack_##node *s){ \
  return s->top; \
} \
\
void push_##node(node *n,stack_##node *s){ \
  n->next = s->top; \
  s->top = n; \
} \
\
node* pop_##node(stack_##node *s){ \
```

```
  node *aux = s->top; \
\
  if(aux != NULL){ \
    s->top = s->top->next; \
    aux->next = NULL; \
  } \
\
  return aux; \
} \
;
```

```
\
void drop_stack_##node(stack_##node **s){ \
  while(!empty_stack_##node(*s)) \
    free(pop_##node(*s)); \
  free(*s); \
} \

int stack_gen = 1
```

tree.c:

```
/*
Necesita la librería: stdlib.h
Para definir un nuevo arbol, node
debe ser del tipo:
struct node{
  struct node* l;
  struct node* r;
  struct node* f;
  ...
}

Se crean las funciones:
stack_node crear_stack_node();
int       empty_stack_node();
node*     top_node(stack_node);
void      push(node,stack_node);
node*     pop(stack_node);
*/
#define DEFINIR_NUEVO_TREE( node ) \
typedef node tree_##node; \
\
tree_##node* crear_tree_##node(){ return NULL; } \
\
int size_tree_##node( tree_##node *t ){ \
  if( t == NULL )   return 0; \
  return size_tree_##node( t->l ) + 1 + \
        size_tree_##node( t->r ); \
} \
\
node* insert_tree_##node( node *n , tree_##node **t){ \
  if( *t == NULL ){ \
    *t = n; \
    n->l = NULL; \
    n->r = NULL; \
    return n; \
  } \
  if( max_##node( *t , n ) == n ){ \
    (*t)->r = insert_tree_##node( n , &((*t)->r) ); \
    (*t)->r->f = (*t)->r; \
  }else{ \
    (*t)->l = insert_tree_##node( n , &((*t)->l) ); \
    (*t)->l->f = (*t)->l; \
  } \
\
  return *t; \
} \
\
node* remove_min_tree_##node( tree_##node **t ){ \
  node *min; \
  if( *t == NULL ) return NULL; \
  if( (*t)->l == NULL ){ \
    min = *t; \
    if( (*t)->f != NULL ) (*t)->f->l = NULL; \
    *t = (*t)->r; \
    min->r = NULL; \
    return min; \
  } \
  return remove_min_tree_##node( &((*t)->l) ); \
} \
\
node* remove_max_tree_##node( tree_##node **t ){ \
  node *max; \
  if( *t == NULL ) return NULL; \
  if( (*t)->r == NULL ){ \
    max = *t; \
    if( (*t)->f != NULL ) (*t)->f->r = NULL; \
    *t = (*t)->l; \
    max->l = NULL; \
    return max; \
  } \
  return remove_max_tree_##node( &((*t)->r) ); \
} \
\
void drop_tree_##node(tree_##node **t){ \
  if( *t != NULL ){ \
    drop_tree_##node( &((*t)->l) ); \
    drop_tree_##node( &((*t)->r) ); \
    free( *t ); \
  } \
} \

int tree_gen = 1;
```

mapa.c:

```
typedef unsigned char uchar;
typedef uchar dir;

#define malloc2( size , var ) (var*)malloc((size)*sizeof(var))
#define malloc1( var ) malloc2( 1 , var )

/*Estan definidos así para mayor facilidad*/
#define AR       (1 << 5)
#define NO       (1 << 4)
#define OE       (1 << 3)
#define AB       (1 << 2)
#define SU       (1 << 1)
#define ES       (1 << 0)
#define SIN_DIR   ( 0  )

#define MURO  ( 0  )
#define PASO  (1 << 3)
#define META  (PASO+1)

dir d_arr[ 6 ] = { AR , NO , ES , AB , SU , OE };

/*coord es un array 1d que sirve para guardar un array 3d*/
typedef struct mapa{
  uchar *coord;
  int meta_i, meta_f;
  int lx, ly, lz, lt;
}mapa;

mapa *m = NULL;

typedef struct punto{
  int x, y, z;
  dir dc, dv;
  int dist;
  struct punto *next;
}punto;

typedef struct camino{
  int d, p_i, p_f;
  struct camino *l, *r, *f;
}camino;

/*Crea el mapa con memoria almacenada
y coord todas como MURO*/
void crearMapa( int , int , int );

/*Libera la memoria del mapa y retorna NULL*/
void borrarMapa();

void obtenMetas();

punto* crearPunto( int x , int y , int z , int d );

camino* max_camino( camino *a , camino *b );

camino* crearCamino( int d , punto *i , punto *f );

void crearMapa( int x , int y , int z ){
  m = malloc1( mapa );
  if( m == NULL )   error( "El mapa no se pudo crear" );
    m->lx = x;  m->ly = y;   m->lz = z;
    m->lt = x*y*z;
  m->coord = malloc2( x*y*z , uchar );
  if( m->coord == NULL ){ borrarMapa(); error("Mapa sin crear"); }
  int i;
  for(i = 0; i < m->lt; i++)
    m->coord[ i ] = MURO;
}

void borrarMapa(){
  if( m == NULL )   return;
  free( m->coord );   free( m );   m = NULL;
}

void obtenMetas(){
  int i = 0, j = (m->lx)*(m->ly)*(m->lz);
  for(; i < j; i++)
    if( m->coord[ i ] == META ){ m->meta_i = i++; break; }
  for(; i < j; i++)
    if( m->coord[ i ] == META ){ m->meta_f = i; break; }
}

punto* crearPunto( int x , int y , int z , int dist ){
  punto *p = malloc1( punto );
  if( p == NULL )   return NULL;
    p->x = x;   p->y = y;   p->z = z;   p->dc = (p->dv = SIN_DIR);
    p->next = NULL;   p->dist = dist;
  return p;
}

DEFINIR_NUEVO_STACK( punto )
DEFINIR_NUEVO_QUEUE( punto )
DEFINIR_NUEVO_TREE( camino )


/*Acceso al indice 1d con coord 3d*/
#define COORD( x , y , z ) \
  m->coord[ ((x)+(y)*(m->lx)+(z)*(m->ly)*(m->lx)) ]

/*Es coord 3d valida*/
#define ES_VAL( x , y , z ) \
(( 0 <= (x) && (x) < m->lx ) && \
 ( 0 <= (y) && (y) < m->ly ) && \
 ( 0 <= (z) && (z) < m->lz ))

#define ES_PASO( x , y , z ) \
  (ES_VAL( x , y , z ) && \
   ((COORD( x , y , z ) & PASO) != 0))

#define ES_MURO( x , y , z ) \
```

```
  (ES_VAL( x , y , z ) && \
   ((COORD( x , y , z) & PASO) == 0))

#define HAY_PASO_AR( p ) ES_PASO( p->x , p->y , p->z-1 )
#define HAY_PASO_AB( p ) ES_PASO( p->x , p->y , p->z+1 )
#define HAY_PASO_NO( p ) ES_PASO( p->x , p->y-1 , p->z )
#define HAY_PASO_SU( p ) ES_PASO( p->x , p->y+1 , p->z )
#define HAY_PASO_OE( p ) ES_PASO( p->x-1 , p->y , p->z )
#define HAY_PASO_ES( p ) ES_PASO( p->x+1 , p->y , p->z )

#define HAY_MURO_AR( p ) ES_MURO( p->x , p->y , p->z-1 )
#define HAY_MURO_AB( p ) ES_MURO( p->x , p->y , p->z+1 )
#define HAY_MURO_NO( p ) ES_MURO( p->x , p->y-1 , p->z )
#define HAY_MURO_SU( p ) ES_MURO( p->x , p->y+1 , p->z )
#define HAY_MURO_OE( p ) ES_MURO( p->x-1 , p->y , p->z )
#define HAY_MURO_ES( p ) ES_MURO( p->x+1 , p->y , p->z )

#define HAY_PASO_A( d , p ) \
(d == AR)?(HAY_PASO_AR(p))?1:0: \
(d == AB)?(HAY_PASO_AB(p))?1:0: \
(d == NO)?(HAY_PASO_NO(p))?1:0: \
(d == SU)?(HAY_PASO_SU(p))?1:0: \
(d == OE)?(HAY_PASO_OE(p))?1:0: \
(d == ES)?(HAY_PASO_ES(p))?1:0:0 \

/*Se mueve a d*/
#define IR_A( d , p ) do{ \
  switch( d ){ \
    case AR: p->z--; break; case AB: p->z++; break; \
    case NO: p->y--; break; case SU: p->y++; break; \
    case OE: p->x--; break; case ES: p->x++; break; \
  } \
}while( 0 ) \

/*Resuelve indice 1d de coord 3d*/
#define INDP( p ) ((p->x)+((p->y)+(p->z)*(m->ly))*(m->lx))
/*Acceso a indice 1d con punto 3d p*/
#define COORD_P( p ) COORD( p->x , p->y , p->z )

#define ES_META_I( p ) (INDP( p ) == m->meta_i)
#define ES_META_F( p ) (INDP( p ) == m->meta_f)

#define IGUAL_P( p , q ) \
  (p->x == q->x && p->y == q->y && p->z == q->z)

/*Resuelve los indices 3d del indice 1d*/
#define IND_X( ind ) ((ind)%(m->lx))
#define IND_Y( ind ) ((((ind)-IND_X(ind))/(m->lx))%(m->ly))
#define IND_Z( ind ) (((((ind)-IND_X(ind))/(m->lx))-IND_Y(ind))/(m->ly))

#define ES_P_PASO( p ) ES_PASO( p->x , p->y , p->z )
#define ES_P_MURO( p ) ES_MURO( p->x , p->y , p->z )

#define ES_P_VAL( p ) ES_VAL( p->x , p->y , p->z )

/*Direccion contraria: AR-AB,NO-SU,ES,OE*/
#define DIR_CONTR( d ) (d > AB)?(d >> 3):(d << 3)
```

generar.c:

```
/*********************************************************
*****
Funciones para la generacion y resolucion de laberintos
*********************************************************
****/

camino* max_camino( camino *a , camino *b ){
  if( a->d <= b->d )   return b;
  return a;
}

camino* crearCamino( int d , punto *i , punto *f ){
  camino *c = malloc1( camino );
  if( c == NULL )   error( "No se pudo crear camino" );
    c->d = d;   c->p_i = INDP( i );   c->p_f = INDP( f );
    c->l = NULL;   c->r = NULL;   c->f = NULL;
  return c;
}

/*Crea punto aleatorio*/
#define P_ALEAT() \
crearPunto( rand()%(m->lx) , rand()%(m->ly) , rand()%(m->lz) , 1 )

#define BORRAR_DIR( elim , d ) (d - (elim))

#define CONV_A_PASO( p ) COORD_P( p ) = PASO

#define ES_DESTRUIBLE( x , y , z ) \
  (ES_MURO( x , y , z ) && \
   (COORD( x , y , z ) <= MURO+1))

#define ES_DESTR_AR( p ) ES_DESTRUIBLE( p->x , p->y , p->z-1 )
#define ES_DESTR_AB( p ) ES_DESTRUIBLE( p->x , p->y , p->z+1 )
#define ES_DESTR_NO( p ) ES_DESTRUIBLE( p->x , p->y-1 , p->z )
#define ES_DESTR_SU( p ) ES_DESTRUIBLE( p->x , p->y+1 , p->z )
#define ES_DESTR_OE( p ) ES_DESTRUIBLE( p->x-1 , p->y , p->z )
#define ES_DESTR_ES( p ) ES_DESTRUIBLE( p->x+1 , p->y , p->z )

#define ACT_MURO( x , y , z ) COORD( x , y , z ) += 1

#define ACT_MUROS_ADY( p ) do{ \
  if( HAY_MURO_AR( p ) ) ACT_MURO( p->x , p->y , p->z-1 ); \
  if( HAY_MURO_AB( p ) ) ACT_MURO( p->x , p->y , p->z+1 ); \
  if( HAY_MURO_NO( p ) ) ACT_MURO( p->x , p->y-1 , p->z ); \
  if( HAY_MURO_SU( p ) ) ACT_MURO( p->x , p->y+1 , p->z ); \
  if( HAY_MURO_OE( p ) ) ACT_MURO( p->x-1 , p->y , p->z ); \
  if( HAY_MURO_ES( p ) ) ACT_MURO( p->x+1 , p->y , p->z ); \
}while( 0 )
```

```
#define OBTEN_DIRS_DESTR( p ) do{ \
  p->dv = SIN_DIR;   p->dc = SIN_DIR; \
  if( ES_DESTR_AR( p ) ){ p->dv += AR; p->dc++; } \
  if( ES_DESTR_AB( p ) ){ p->dv += AB; p->dc++; } \
  if( ES_DESTR_NO( p ) ){ p->dv += NO; p->dc++; } \
  if( ES_DESTR_SU( p ) ){ p->dv += SU; p->dc++; } \
  if( ES_DESTR_OE( p ) ){ p->dv += OE; p->dc++; } \
  if( ES_DESTR_ES( p ) ){ p->dv += ES; p->dc++; } \
}while( 0 )

#define OBTEN_DIRS_ACCES( p ) do{ \
  p->dv = SIN_DIR;   p->dc = SIN_DIR; \
  if( HAY_PASO_AR( p ) ){ p->dv += AR; p->dc++; } \
  if( HAY_PASO_AB( p ) ){ p->dv += AB; p->dc++; } \
  if( HAY_PASO_NO( p ) ){ p->dv += NO; p->dc++; } \
  if( HAY_PASO_SU( p ) ){ p->dv += SU; p->dc++; } \
  if( HAY_PASO_OE( p ) ){ p->dv += OE; p->dc++; } \
  if( HAY_PASO_ES( p ) ){ p->dv += ES; p->dc++; } \
}while( 0 )

#define ESCOGER_DIR( p ) \
(HAY_PASO_AR(p) && (p->dv & AR))?AR: \
(HAY_PASO_AB(p) && (p->dv & AB))?AB: \
(HAY_PASO_NO(p) && (p->dv & NO))?NO: \
(HAY_PASO_SU(p) && (p->dv & SU))?SU: \
(HAY_PASO_OE(p) && (p->dv & OE))?OE: \
(HAY_PASO_ES(p) && (p->dv & ES))?ES:SIN_DIR

dir dirAleatoria( dir *d );

void generarLaberinto();

void resolverLaberinto();

dir dirAleatoria( dir *d ){
  dir op1[ 4 ], op2[ 2 ];
  int num_op1 = 0, num_op2 = 0, i;

  for(i = 0; i < 6; i++){
    if( (*d) & d_arr[ i ] ){
      if( d_arr[ i ] == AR || d_arr[ i ] == AB )
        op2[ num_op2++ ] = d_arr[ i ];
      else{
        op1[ num_op1++ ] = d_arr[ i ];
      }
    }
  }

  if( num_op1 == 0 && num_op2 == 0 )   return SIN_DIR;
  if( num_op1 == 0 ){
    if( num_op2 == 1 )   return op2[ 0 ];
    return op2[ rand()%num_op2 ];
  }
  if( num_op1 == 1 )   return op1[ 0 ];
  return op1[ rand()%num_op1 ];
}
```

```
#define debug0( str ) do{ \
  pintarMapa( p ); \
  imprimirPantalla(); \
  printf( "%s:" , str ); \
  debug( DEBUG_PUNTO , p , "Punto" ); \
  debug( DEBUG_DIR , &(p->dv) , "Dir" ); \
  leerDeConsola(); }while( 0 )
#define debug1( s ) if( 1 ) debug0( s )

void generarLaberinto(){
  stack_punto *pila = crear_stack_punto();
  tree_camino *arb = crear_tree_camino();

  dir d = SIN_DIR;
  int esNuevaRama = 1, esHoja = 0;

  punto *p = P_ALEAT();
  punto *i = p;
    CONV_A_PASO( p );   ACT_MUROS_ADY( p );
  int jh = 1;

  do{
    OBTEN_DIRS_DESTR( p );
    if( p->dv != SIN_DIR ){
      if( p->dc != 1 )   push_punto( p , pila );
      d = p->dv;
      p = crearPunto( p->x , p->y , p->z , p->dist );
        IR_A( dirAleatoria( &d ) , p );
        CONV_A_PASO( p );   ACT_MUROS_ADY( p );
      esNuevaRama = 1;
      p->dist++;
    }else{
      if( p == i )   esHoja = 1;
      if( esNuevaRama ){
        insert_tree_camino( crearCamino( p->dist , i , p ) , &arb );
        if( size_tree_camino( arb ) > 4 ){
          free( remove_min_tree_camino( &arb ) );
          free( remove_min_tree_camino( &arb ) );
        }
      }else   free( p );
      p = pop_punto( pila );
      esNuevaRama = 0;
    }
    if( (jh++)%1000000 == 0 ) printf("%d,",jh);
  }while( p != NULL );

  /*Escogemos el camino más largo de acuerdo al árbol*/
  camino *cam, *cam2;
  if( esHoja ){
    cam = remove_max_tree_camino( &arb );
    m->meta_i = cam->p_i;
    m->meta_f = cam->p_f;
    free( cam );
  }else{
    cam = remove_max_tree_camino( &arb );
    cam2 = remove_max_tree_camino( &arb );
```

```
    m->meta_i = cam->p_f;
    m->meta_f = cam2->p_f;
    free( cam );   free( cam2 );
  }

  drop_tree_camino( &arb );
  drop_stack_punto( &pila );
}

#define debug2( str ) do{ \
  pintarMapa( p ); \
  imprimirPantalla(); \
  print( T_PUNTO , r , " Inicio" ); \
  print( T_PUNTO , q , " Final" ); \
  printf( "\n%s:" , str ); \
  print( T_PUNTO , p , " Actual" ); \
  print( T_DIR , &(d) , " DirAnt" ); \
  print( T_DIR , &(p->dv) , " Dir" ); \
  leerDeConsola(); }while( 0 )
#define debug3( s ) if( 1 ) debug2( s )

void resolverLaberinto(){
  stack_punto *pila = crear_stack_punto();
  punto *p = crearPunto( IND_X( m->meta_i ) ,
                  IND_Y( m->meta_i ) ,
                  IND_Z( m->meta_i ) , 1 );
  punto *q = crearPunto( IND_X( m->meta_f ) ,
                  IND_Y( m->meta_f ) ,
                  IND_Z( m->meta_f ) , 1 );
  punto *r = crearPunto( IND_X( m->meta_i ) ,
                  IND_Y( m->meta_i ) ,
                  IND_Z( m->meta_i ) , 1 );
  dir d = SIN_DIR;
  int esDePila = 0;
  debug3("Inicio.");
  do{
    if( !esDePila ) OBTEN_DIRS_ACCES( p );
    debug3("Antes de borrar.");
    if( d != SIN_DIR ){ p->dv = BORRAR_DIR( d , p->dv ); p->dc--; }
    debug3("Antes de decision.");
    if( p->dc == 1 ){
      IR_A( p->dv , p );
      d = DIR_CONTR( p->dv );
      esDePila = 0;debug3("Solo hay un camino.");
    }else if( p->dc > 1 ){
      d = ESCOGER_DIR( p );
      p->dc--;
      p->dv = BORRAR_DIR( d , p->dv );
      debug3("Hay mas de un camino.");
      push_punto( p , pila );
      p = crearPunto( p->x , p->y , p->z , 1 );
      IR_A( d , p );
      d = DIR_CONTR( d );
      esDePila = 0;
    }else if( p->dc == 0 ){
      free( p );
      p = pop_punto( pila );
      d = SIN_DIR;
      debug3("Saco de pila.");
      esDePila = 1;
    }
    debug3("Despues de decision.");
  }while( !ES_META_F( p ) );

  while( strncmp( leerDeConsola() , "salir" , 5 ) != 0 ){
    debug3("SE LLEGO A LA META FINAL...!!!!!!!!!!!!!!!");
  }

  if( p != NULL )   free( p );
  if( q != NULL )   free( q );
  if( r != NULL )   free( r );
  drop_stack_punto( &pila );
}
```

lectura.c:

```
typedef struct ent{
  uchar dato;
  struct ent *next;
}ent;

DEFINIR_NUEVO_QUEUE( ent )

ent* nuevo_ent( int dato ){
  ent *e = (ent*)malloc(sizeof(ent));
    e->dato = dato;
  return e;
}

/*Sale del programa si no puede leerlo*/
void leerMapa( char* nombre );

/*Guarda el mapa en un archivo de texto*/
void guardarMapa( char* nombre );


/*Lee de la consola sin contar espacios*/
char* leerDeConsola();

/*Mueve el punto de acuerdo al caracter c,
no hace algo si es movimiento prohibido*/
void mover( char c , punto *p );
void iniciarPantalla();

/*Pinta iniciando en x,y con ancho w y largo h*/
void pintarEnPantalla( char** obj , int x , int y , int w , int h );
void pintarMuro( int x , int y );
void pintarPasillo( int x , int y );
```

```
void pintarElevadorA( int x , int y );
void pintarElevadorB( int x , int y );
void pintarElevadorC( int x , int y );
void pintarPunto( int x , int y );
void pintarMeta( int x , int y );
void pintarMapa( punto *p );
void imprimirPantalla();

void leerMapa( char* nombre ){
  FILE* fp;
  queue_ent *q;
  int tam = 0, x, y, z, m_i, m_f;
  char s[ 10 ];
  uchar c;

  fp = fopen( nombre , "r" );
  if( fp == NULL )   error( "No se pudo abrir mapa." );

  fgets( s , sizeof( s ) , fp );   x = atoi( s );
  fgets( s , sizeof( s ) , fp );   y = atoi( s );
  fgets( s , sizeof( s ) , fp );   z = atoi( s );
  fgets( s , sizeof( s ) , fp );   m_i = atoi( s );
  fgets( s , sizeof( s ) , fp );   m_f = atoi( s );

  if( x <= 1 || y <= 1 || z <= 1 )
    error( "Mapa muy chico" );
  if( 1024 < x || 1024 < y || 1024 < z )
    error( "Mapa muy grande" );

  q = crear_queue_ent();
  while( !feof( fp ) ){
    c = fgetc ( fp );
    if( c == '1' ){
      insert_queue_ent( nuevo_ent( PASO ) , q );
      tam++;
    }else if( c == '0' ){
      insert_queue_ent( nuevo_ent( MURO ) , q );
      tam++;
    }
  }
  fclose( fp );

  if( tam != x*y*z ){
    drop_queue_ent( &q );
    error( "Las dimensiones del mapa no concuerdan." );
  }

  crearMapa( x , y , z );
  m->meta_i = m_i;
  m->meta_f = m_f;
  tam = 0;
  while( !empty_queue_ent( q ) ){
    m->coord[ tam++ ] = front_ent( q )->dato;
    free( remove_queue_ent( q ) );
  }
  obtenMetas();
  drop_queue_ent( &q );
}

void guardarMapa( char* nombre ){
  FILE *fp = fopen( nombre , "w" );
  if( fp == NULL )   error( "No se pudo guardar el mapa." );

  fprintf( fp , "%d\n" , m->lx );
  fprintf( fp , "%d\n" , m->ly );
  fprintf( fp , "%d\n" , m->lz );
  fprintf( fp , "%d\n" , m->meta_i );
  fprintf( fp , "%d\n" , m->meta_f );

  int i, j, k, x = m->lx, y = m->ly, z = m->lz;
  for(k = 0; k < z; k++){
    for(j = 0; j < y; j++){
      for(i = 0; i < x; i++){
        if( ES_PASO( i , j , k ) )
          fprintf(fp,"%d",1);
        else
          fprintf(fp,"%d",0);
      }
      fprintf( fp , "\n" );
    }
    fprintf( fp , "-----------------------------------\n" );
  }
  fclose( fp );
}
/*Pantalla del laberinto*/
#define LARGO 80
#define ANCHO 40

char pantalla[ ANCHO ][ LARGO + 1 ];

void iniciarPantalla(){
  int i , j;
  for( i = 0 ; i < ANCHO ; i++ )
    for( j = 0 ; j < LARGO + 1 ; j++ )
      if( j == LARGO )
        pantalla[ i ][ j ] = '\0';
      else
        pantalla[ i ][ j ] = '.';
}

/*Pinta iniciando en x,y con ancho w y largo h*/
void pintarEnPantalla( char** obj ,
                       int x , int y ,
                       int w , int h ){
  int i , j;
  for( i = 0 ; i < w ; i++ )
    for( j = 0 ; j < h ; j++ )
      pantalla[ y+j ][ x+i ] = obj[ j ][ i ];
}

void pintarMuro( int x , int y ){
  char *muro[ 5 ] = { "_____" ,
                      "_____" ,
```

```
                "_____" ,
                "_____" ,
                "_____" };
  pintarEnPantalla( muro , 5*x , 5*y , 5 , 5 );
}

void pintarPasillo( int x , int y ){
  char *paso[ 5 ] = { "     " ,
                "    " ,
                "    " ,
                "    " ,
                "    " };
  pintarEnPantalla( paso , 5*x , 5*y , 5 , 5 );
}

void pintarElevadorA( int x , int y ){
  char *elev[ 1 ] = { "E" };
  pintarEnPantalla( elev , 5*x + 1 , 5*y + 1 , 1 , 1 );
}

void pintarElevadorB( int x , int y ){
  char *elev[ 1 ] = { "F" };
  pintarEnPantalla( elev , 5*x + 2 , 5*y + 2 , 1 , 1 );
}

void pintarElevadorC( int x , int y ){
  char *elev[ 1 ] = { "X" };
  pintarEnPantalla( elev , 5*x + 1 , 5*y + 1 , 1 , 1 );
}

void pintarPunto( int x , int y ){
  char *punt[ 1 ] = { "O" };
  pintarEnPantalla( punt , 5*x + 1 , 5*y , 1 , 1 );
  pintarEnPantalla( punt , 5*x + 2 , 5*y , 1 , 1 );
  pintarEnPantalla( punt , 5*x + 3 , 5*y , 1 , 1 );
  pintarEnPantalla( punt , 5*x + 1 , 5*y + 4 , 1 , 1 );
  pintarEnPantalla( punt , 5*x + 2 , 5*y + 4 , 1 , 1 );
  pintarEnPantalla( punt , 5*x + 3 , 5*y + 4 , 1 , 1 );
}

void pintarMeta( int x , int y ){
  char *punt[ 1 ] = { "M" };
  pintarEnPantalla( punt , 5*x + 2 , 5*y + 2 , 1 , 1 );
}

void pintarMapa( punto *p ){
  punto *aux = crearPunto( p->x , p->y , p->z , 0 );

  iniciarPantalla();
  int i, j;
  for(j = 0; j < 8; j++){
    for(i = 0; i < 16; i++){
      aux->x = p->x + i - 7;
      aux->y = p->y + j - 3;
      if( ES_P_PASO( aux ) )   pintarPasillo( i , j );
      if( aux->x == p->x && aux->y == p->y )
          pintarPunto( i , j );
      if( ES_P_PASO( aux ) && HAY_PASO_AR( aux ) &&
                          HAY_PASO_AB( aux ) )
          pintarElevadorC( i , j );
      if( ES_P_PASO( aux ) && HAY_PASO_AR( aux ) )
          pintarElevadorA( i , j );
      if( ES_P_PASO( aux ) && HAY_PASO_AB( aux ) )
          pintarElevadorB( i , j );
      if( ES_P_MURO( aux ) )
          pintarMuro( i , j );
      if( ES_META_I( aux ) || ES_META_F( aux ) )
        pintarMeta( i , j );
    }
  }
  free( aux );
}

void imprimirPantalla(){
  int i;
  system( "CLS" );
  for(i = 0; i < ANCHO; i++)
    printf("%s",pantalla[ i ]);
}

char* leerDeConsola(){
  static char str[ 1024 ];

  fgets( str , sizeof( str ) , stdin );
  if(str[strlen(str) - 1] == '\n'){
    str[strlen(str) - 1] = '\0';
  }else{
    char dummy[2];
    do{ fgets( dummy , sizeof(dummy) , stdin );
    }while( dummy[strlen(dummy) - 1] != '\n' );
  }

  int i = 0, j = 0;
  while( *(str + j) != '\0'){
    if( *(str + j) != ' ' ){
      *(str + i) = *(str + j);   i++;
    }
    j++;
  }
  *(str + i) = '\0';
  return str;
}

void mover( char c , punto *p ){
  dir d = SIN_DIR;
  switch( c ){
    case 'w':   d = NO;   break;
    case 's':   d = SU;   break;
    case 'a':   d = OE;   break;
    case 'd':   d = ES;   break;
    case 'e':   d = AR;   break;
    case 'f':   d = AB;   break;
  }
  if( HAY_PASO_A( d , p ) )   IR_A( d , p );
```

}

lab.c:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <ctype.h>
#include <time.h>

#define T_PUNTO 3450
#define T_DIR   3451

void error( char* );

void print( int , void* , char* );

#include "estructuras/stack.c"
#include "estructuras/queue.c"
#include "estructuras/tree.c"
#include "mapa.c"
#include "lectura.c"
#include "generar.c"

void navegar( char *nombre ){
  leerMapa( nombre );
  punto *jug = crearPunto( IND_X( m->meta_i ) ,
                   IND_Y( m->meta_i ) ,
                   IND_Z( m->meta_i ) , 1 );
  punto *q = crearPunto( IND_X( m->meta_f ) ,
                  IND_Y( m->meta_f ) ,
                  IND_Z( m->meta_f ) , 1 );
  punto *r = crearPunto( IND_X( m->meta_i ) ,
                  IND_Y( m->meta_i ) ,
                  IND_Z( m->meta_i ) , 1 );
  char *s = NULL;
  while( 1 ){
    pintarMapa( jug );
    imprimirPantalla();
    if( ES_META_F( jug ) )   printf("META FINAL!!!!!\n");
    if( ES_META_I( jug ) )   printf("META INICIAL.\n");
    print( T_PUNTO , r , "INICIAL " );
    print( T_PUNTO , q , "FINAL " );
    print( T_PUNTO , jug , "ACTUAL " );
    s = leerDeConsola();
    if( s[ 0 ] == '0' ){ break; }
    mover( s[ 0 ] , jug );
  }
  free( jug );
  free( r );
  free( q );
  borrarMapa();
}

int main( int argc , char** argv ){

  if( argc == 1 )   error( "No hay nada que hacer" );


  if( strncmp( argv[ 1 ] , "gen" , strlen( argv[ 0 ] ) ) == 0 ){
    if( argc <= 5 )   error( "Informacion insuficiente" );
    printf( "Creando mapa '%s'...\n" , argv[ 2 ] );
    int x = atoi( argv[ 3 ] );
    int y = atoi( argv[ 4 ] );
    int z = atoi( argv[ 5 ] );
    if( x < 4 || y < 4 || z <4 )   error( "Mapa muy chico..." );
    crearMapa( x , y , z );
    generarLaberinto();
    guardarMapa( argv[ 2 ] );
    puts( "Mapa guardado exitosamente." );
    return 0;
  }

  if( strncmp( argv[ 1 ] , "nav" , strlen( argv[ 0 ] ) ) == 0 ){
    if( argc == 2 )   error( "No se conoce nombre del mapa" );
    navegar( argv[ 2 ] );
    return 0;
  }

  if( strncmp( argv[ 1 ] , "res" , strlen( argv[ 0 ] ) ) == 0 ){
    if( argc == 2 )   error( "No se conoce nombre del mapa" );
    leerMapa( argv[ 2 ] );   resolverLaberinto();
    return 0;
  }

  return 0;
}

void error( char* txt ){
  puts( txt );
  exit( EXIT_FAILURE );
}

void print( int tipo , void *var , char *s ){
  printf("%s:", s );
  punto *a;   dir *b;
  dir arr[ 6 ] = {AR,AB,OE,NO,SU,ES};
  char *arr_s[ 6 ] = {"AR","AB","OE","NO","SU","ES"};
  int i;
  switch( tipo ){
    case T_PUNTO:
      a = (punto*)var;
      printf("(%d,%d,%d),", a->x , a->y , a->z );
      break;
    case T_DIR:
      b = (dir*)var;
      for(i = 0; i < 6; i++)
        if( (*b) & arr[ i ] ) printf("%s:", arr_s[ i ] );
```

```
        printf(",");
        break;
    }
}
```