Curvas de Peano aleatorias en Arduino

Trabajo para el curso «Robótica Práctica»
del CTIF Capital de la Consejería de Educación de la Comunidad de Madrid
Autor: Pedro Reina <pedro@pedroreina.net>; fecha: D.2.4.2017

Punto de partida

La curva de Peano es un concepto matemático que consiste en una curva (por tanto, de dimensión uno) que rellena completamente un cuadrado (que es de dimensión dos):
https://es.wikipedia.org/wiki/Curva_de_Peano

La construcción clásica de tal curva lleva a darle un aspecto sumamente uniforme. Se me ocurrió la idea de trazar curvas que llenaran un cuadrado pero que no fueran uniformes, usando algo de aleatoriedad. Para llevarlo a la práctica, no necesitaba el paso al límite que exige la construcción matemática, sino simplemente considerar el cuadrado dividido en cuadrados más pequeños e idear la manera de recorrerlos con unas sencillas reglas:

- Se comienza en un punto arbitrario
- Se pasa por todos los cuadrados
- Solo se pasa una vez por cada cuadrado
- Se puede acabar en cualquier punto
- El recorrido sigue un orden aleatorio

Estas reglas también permiten generalizar el problema, y en vez de partir de un cuadrado, se puede partir de un rectángulo, que consideramos descompuesto en una retícula de filas y columnas formadas por cuadrados del mismo tamaño.

Aplicación práctica

No estoy al corriente de ninguna aplicación práctica en robótica de la existencia de curvas de Peano no uniformes, pero como estas curvas garantizan que se pasa por todos los puntos de manera no determinista, no descarto que puedan ser usadas en robots de vigilancia, de fotografía, de limpieza o similares.

Sí han sido usadas ya en la práctica para generar transiciones entre dos imágenes de un vídeo, como se puede ver en el ejemplo
http://pedroreina.net/desarrollos/videopy/peano-50

Uso en el aula

Como profesor de Matemáticas, en bachillerato necesito explicar el proceso de paso al límite; las curvas de Peano son un ejemplo intrigante, aunque excedan el nivel de bachillerato, ya que el concepto de llenar un espacio de dimensión dos con una línea de dimensión uno es aparentemente contradictorio.

Creo que puede resultar motivador para los alumnos mostrar cómo se pueden llevar a la práctica las ideas abstractas de la Matemática y para ello utilizar una placa Arduino con una matriz de 8 × 8 LEDs puede ser atractivo.

Problemas de implementación

Se ha estudiado muchas veces cómo escribir un algoritmo eficiente similar al que exige este problema, ya que es una variante de encontrar un camino

hamiltoniano en un grafo no orientado:
https://es.wikipedia.org/wiki/Camino_hamiltoniano

Como es sabido, el problema del camino hamiltoniano es NP-completo, por lo que ya preveía que el tiempo de resolución del problema podía ser elevado si el número de cuadrados en los que se divide el rectángulo es grande. La cuestión práctica, como siempre, es la implementación que se pueda desarrollar: en qué lenguaje, con qué heurísticos, con cuánta habilidad, etc.

Mi primer intento fue en Python en un ordenador de escritorio con un procesador Intel i5 a 3.5 GHz: tardaba alrededor de quince minutos en resolver un problema de 6 × 4 cuadrados. El tiempo era muy variable, claro está, al utilizar un algoritmo no determinista. El algoritmo utiliza la técnica de «vuelta atrás», lo que se conoce en inglés como *backtracking*, y aprovecha un heurístico *ad hoc*.

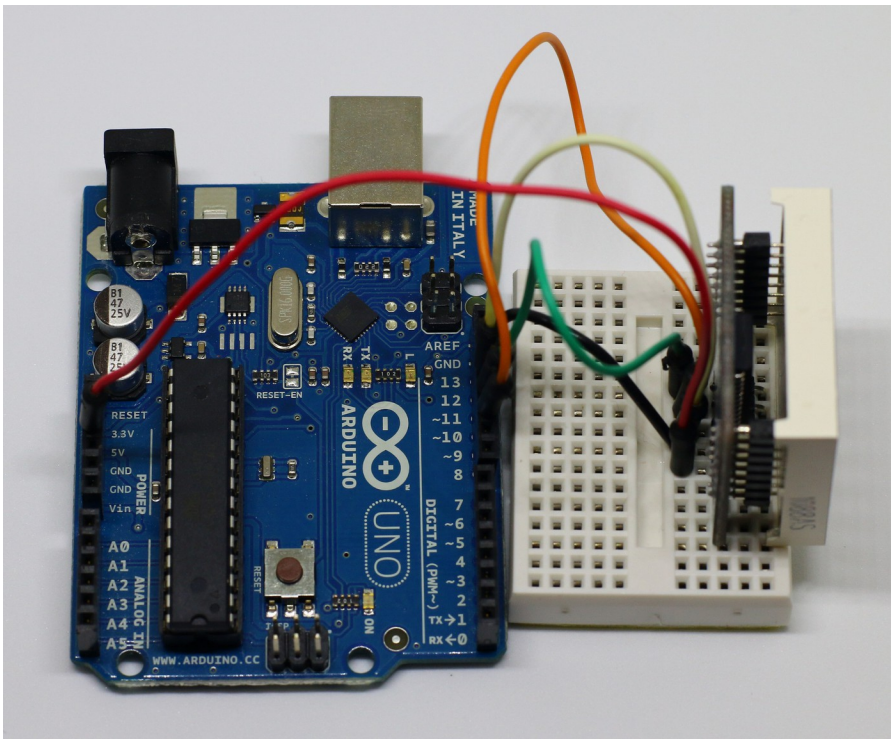
El segundo intento consistió en programar el mismo algoritmo en C intentando aprovechar al máximo los ciclos de la CPU. La mejora de rendimiento fue espectacular: los problemas de 6 × 4 cuadrados se resuelven instantáneamente en el ordenador usado y los problemas de 9 × 6 requieren pocos segundos. A partir de ahí, los tiempos de cálculo son muy variables, pero no he conseguido resolver en un tiempo razonable problemas de 10 × 10, por ejemplo.

Por último, se presentaba la oportunidad de llevar el algoritmo a una placa Arduino, aprovechando que esta se puede programar en C estándar. El tamaño máximo de problema que he podido conseguir resolver en un tiempo razonable con efectos de demostración es 6 × 6. Se pueden resolver problemas de 7 × 7, pero se tarda demasiado y la paciencia de los espectadores no podría soportarlo. No he intentado resolver problemas de mayor tamaño.

Construcción real de la demostración

La práctica que presento como trabajo para el curso tiene un apartado de hardware y otro de software.

He utilizado una placa Arduino UNO original, una matriz de LEDs rojos 8 × 8 dirigida por un circuito integrado MAX7219, una pequeña placa de prototipado y cinco cables de conexión:



Para el software, he utilizado una librería para gestionar el MAX7219, que he descargado de
https://github.com/shaai/Arduino_LED_matrix_sketch

Explicación del código

Es necesario un archivo de cabecera para poder definir el tipo **celda**, que he usado para adaptar el código entre un ordenador de sobremesa y una placa Arduino. Si el *typedef* se incluye directamente en el archivo `.ino`, el IDE de Arduino da errores de compilación (es un fallo del IDE).

La demostración realiza un ciclo de cuatro fases:

- Iluminar la matriz con una curva de Peano aleatoria de 4 × 4
- Apagar la matriz con una curva de Peano aleatoria de 4 × 4
- Iluminar la matriz con una curva de Peano aleatoria de 6 × 6
- Apagar la matriz con una curva de Peano aleatoria de 6 × 6

Vídeo de demostración

He preparado un vídeo de demostración que se puede descargar hasta el 30 de junio de 2017 de la dirección http://pedroreina.net/rp/peano.mov

Código

Los archivos originales del proyecto se pueden descargar hasta el 30 de junio de 2017 de la dirección http://pedroreina.net/rp/

Se reproducen a continuación:

celda.h

```
/*-------------------------------------------------------------------------
 * Tipos de usuario
 *-------------------------------------------------------------------------*/

typedef byte celda;
```

peano.ino

```
/*-------------------------------------------------------------------------
 * Fichero:  peano.ino
 * Objetivo: Demostración de curvas de Peano aleatorias
 * Autor:    Pedro Reina <pedro@pedroreina.net>
 * Fecha:    D.12.2.2017
 * Licencia: Dominio público
 *           https://creativecommons.org/publicdomain/zero/1.0/
 *-------------------------------------------------------------------------*/

/*-------------------------------------------------------------------------
 * Ficheros de cabecera
 *-------------------------------------------------------------------------*/

// La librería de acceso al MAX7219
// https://github.com/shaai/Arduino_LED_matrix_sketch
#include "LedControlMS.h"

// Definición del tipo celda
#include "celda.h"

/*-------------------------------------------------------------------------
 * Definición de constantes que puede retocar el usuario
 *-------------------------------------------------------------------------*/

// La intensidad con que iluminamos los LED (de 0 a 15)
#define INTENSIDAD 1

// El número de nuestra matriz LED
#define NUM_MATRIZ 0
```

```
// Tiempo de espera en milisegundos para mostrar otro paso
#define RETARDO 150

/*------------------------------------------------------------------------
 * Definición de constantes que usa el programa internamente
 *------------------------------------------------------------------------*/

// Valores lógicos
#define SI 1
#define NO 0

// En cuantas direcciones nos podemos mover
#define MAX_DIR 4

// Los posibles estados de una celda
#define TIERRA ((celda)0)
#define AGUA   ((celda)1)
#define AIRE   ((celda)2)

/*------------------------------------------------------------------------
 * Definición de macros
 *------------------------------------------------------------------------*/

// Índice unidimensional de una celda bidimensional
#define Indice(f,c) (MaxCol*(f)+(c))

// Dice el estado de una celda
#define DiEstado(f,c) (Estado[Indice(f,c)])

// Cambia el estado de una celda
#define CambiaEstado(f,c,e) (Estado[Indice(f,c)]=(e))

// Dice cuántas vecinas tiene una celda
#define DiVecinas(f,c) (NumVecinas[Indice(f,c)])

// Anota cuántas vecinas tiene una celda
#define AnotaNumVecinas(f,c,n) (NumVecinas[Indice(f,c)]=(n))

/*------------------------------------------------------------------------
 * Variables globales
 *------------------------------------------------------------------------*/

// Las conexiones que uso yo para conectar Arduino con el controlador
// http://www.prometec.net/8x8-max7219/
LedControl ControladorMatriz=LedControl(12,11,10,1);

// Los datos para la generación de la curva
celda MaxFil, MaxCol, FilIni, ColIni;

// El total de puntos del rectángulo
celda Total;

// El estado de una celda: AIRE, AGUA o TIERRA
celda *Estado;

// Fila y columna de los puntos de la solución
celda *SolFil, *SolCol;

// Posición del último punto encontrado de la posible solución
int PosicionSolucion;

// Cuántos saltos se han dado desde un punto
celda *NumeroSaltosDesde;

// Qué saltos se han dado desde un punto
celda *SaltoDesde;

// Direcciones para moverse de un punto a otro
int Direccion[MAX_DIR][2] = {{1,0}, {-1,0}, {0,1}, {0,-1}};
int Permutacion[MAX_DIR] = {0, 1, 2, 3};

// Número de celdas vecinas a una dada
celda *NumVecinas;

// Lista de vecinas de cada celda
celda *ListaVecinas;
```

```
// Lista de celdas inundadas comprobando la conexión
celda *ListaInundadas;

/*-------------------------------------------------------------------------
 * Bloque de inicio
 *-------------------------------------------------------------------------*/

void setup()
  {
  // Activar la matriz de LEDs
  ControladorMatriz.shutdown(NUM_MATRIZ,false);

  // Ajustar el brillo
  ControladorMatriz.setIntensity(NUM_MATRIZ,INTENSIDAD);

  // Borrar todo
  ControladorMatriz.clearDisplay(NUM_MATRIZ);

  // Arrancamos el generador de números aleatorios
  randomSeed(analogRead(0));

  // Cuántas celdas tenemos en el caso mayor
  Total = 36;

  // Reservamos la memoria necesaria
  PreparaVariables();
  }

/*-------------------------------------------------------------------------
 * Bucle principal
 *-------------------------------------------------------------------------*/

void loop()
  {
  Peano(4,true);
  Peano(4,false);
  Peano(6,true);
  Peano(6,false);
  }

/*-------------------------------------------------------------------------
 * Definición de funciones
 *-------------------------------------------------------------------------*/

//-------------------------------------------------------------------------
// Crea y representa una curva de Peano con la dimensión e iluminación dadas
void Peano(int Tamano, int Flag)
  {
  // Los valores iniciales
  MaxFil = Tamano;
  MaxCol = Tamano;
  FilIni = random(0,Tamano);
  ColIni = random(0,Tamano);

  // Cuántas celdas tendrá la solución
  Total = Tamano*Tamano;

  // Precalculamos todas las celdas vecinas
  CalculaVecinas();

  // Limpiamos los valores de ejecuciones anteriores
  LimpiaTerreno();

  // Buscamos una solución
  BuscaSolucion();

  // La mostramos
  MuestraSolucion(Tamano, Flag);

  // Esperamos un poquito para ver el resultado
  delay(4*RETARDO);
  }

//-------------------------------------------------------------------------
// Reserva memoria dinámicamente para algunas variables
void PreparaVariables(void)
  {
  celda i;
```

```
  SolFil = (celda *)malloc(Total*sizeof(celda));
  SolCol = (celda *)malloc(Total*sizeof(celda));
  Estado = (celda *)malloc(Total*sizeof(celda));
  NumVecinas = (celda *)malloc(Total*sizeof(celda));
  NumeroSaltosDesde = (celda *)malloc(Total*sizeof(celda));
  ListaInundadas = (celda *)malloc(Total*sizeof(celda));
  SaltoDesde = (celda *)malloc(MAX_DIR*Total*sizeof(celda));
  ListaVecinas = (celda *)malloc(MAX_DIR*Total*sizeof(celda));
  }

//---------------------------------------------------------------------------
// Prepara el terreno para volver a generar otra curva
void LimpiaTerreno(void)
  {
  celda i;

  for ( i=0; i<Total; i++ )
    {
    Estado[i]=AIRE;
    NumeroSaltosDesde[i]=0;
    }
  }

//---------------------------------------------------------------------------
// Calcula cuántas vecinas tiene cada celda y cuáles son
void CalculaVecinas(void)
  {
  celda i, j;

  // Las esquinas tienen dos vecinas
  AnotaNumVecinas(0,0,2);
  AnotaVecina(0,0,0,1,0);
  AnotaVecina(0,0,1,0,1);

  AnotaNumVecinas(0,MaxCol-1,2);
  AnotaVecina(0,MaxCol-1,0,MaxCol-2,0);
  AnotaVecina(0,MaxCol-1,1,MaxCol-1,1);

  AnotaNumVecinas(MaxFil-1,0,2);
  AnotaVecina(MaxFil-1,0,MaxFil-1,1,0);
  AnotaVecina(MaxFil-1,0,MaxFil-2,0,1);

  AnotaNumVecinas(MaxFil-1,MaxCol-1,2);
  AnotaVecina(MaxFil-1,MaxCol-1,MaxFil-2,MaxCol-1,0);
  AnotaVecina(MaxFil-1,MaxCol-1,MaxFil-1,MaxCol-2,1);

  // Los bordes tienen tres vecinas
  for ( i=1; i<MaxCol-1; i++ )
    {
    AnotaNumVecinas(0,i,3);
    AnotaVecina(0,i,0,i-1,0);
    AnotaVecina(0,i,0,i+1,1);
    AnotaVecina(0,i,1,i,2);

    AnotaNumVecinas(MaxFil-1,i,3);
    AnotaVecina(MaxFil-1,i,MaxFil-1,i-1,0);
    AnotaVecina(MaxFil-1,i,MaxFil-1,i+1,1);
    AnotaVecina(MaxFil-1,i,MaxFil-2,i,2);
    }
  for ( i=1; i<MaxFil-1; i++ )
    {
    AnotaNumVecinas(i,0,3);
    AnotaVecina(i,0,i-1,0,0);
    AnotaVecina(i,0,i+1,0,1);
    AnotaVecina(i,0,i,1,2);

    AnotaNumVecinas(i,MaxCol-1,3);
    AnotaVecina(i,MaxCol-1,i-1,MaxCol-1,0);
    AnotaVecina(i,MaxCol-1,i+1,MaxCol-1,1);
    AnotaVecina(i,MaxCol-1,i,MaxCol-2,2);
    }

  // El resto tiene cuatro vecinas
  for ( i=1; i<MaxFil-1; i++ )
    {
    for ( j=1; j<MaxCol-1; j++ )
      {
```

```
      AnotaNumVecinas(i,j,4);
      AnotaVecina(i,j,i-1,j,0);
      AnotaVecina(i,j,i+1,j,1);
      AnotaVecina(i,j,i,j-1,2);
      AnotaVecina(i,j,i,j+1,3);
      }
    }
  }

//---------------------------------------------------------------------------
// Anota una vecina de una celda
void AnotaVecina(celda FilOrigen, celda ColOrigen, celda FilVecina,
                 celda ColVecina, celda Numero)
  {
  int Origen, Vecina, Base;

  // Convertimos en cada celda las dos dimensiones en una
  Origen = Indice(FilOrigen,ColOrigen);
  Vecina = Indice(FilVecina,ColVecina);

  // La dirección base de esta vecina
  Base = MAX_DIR * Origen + Numero;

  // La anotamos
  ListaVecinas[Base]=Vecina;
  }

//---------------------------------------------------------------------------
// Busca una solución que tenga los puntos requeridos
void BuscaSolucion(void)
  {
  celda Punto, Fil, Col, FilSig, ColSig, Dir;
  int CurvaCompleta, SaltoEncontrado;

  // Comenzamos en el punto que nos digan
  PosicionSolucion = -1;
  Fil = FilIni;
  Col = ColIni;
  AlargaSolucion(Fil, Col);

  // Tenemos que buscar la solución hasta que la encontremos
  CurvaCompleta = NO;
  while ( !CurvaCompleta )
    {
    // Revolvemos las direcciones para mejorar el efecto aleatorio
    RevuelveDirecciones();

    // Buscamos el siguiente punto
    SaltoEncontrado = NO;
    for ( Dir=0; Dir<MAX_DIR && !SaltoEncontrado; Dir++ )
      {
      // El posible siguiente punto
      FilSig = Fil+Direccion[Permutacion[Dir]][0];
      ColSig = Col+Direccion[Permutacion[Dir]][1];

      // Si se puede dar el salto...
      if ( SaltoAceptable(Fil, Col, FilSig, ColSig) )
        {
        // ... lo damos
        AnotaSalto(Fil, Col, FilSig, ColSig);
        Fil = FilSig;
        Col = ColSig;
        AlargaSolucion(Fil, Col);
        SaltoEncontrado = SI;
        // Comprobamos si hemos terminado
        if ( PosicionSolucion == Total-1 )
          { CurvaCompleta = SI; }
        }
      } // Fin for Dir

    // Si no hemos podido avanzar en ninguna dirección
    if ( !SaltoEncontrado )
      {
      // Quitamos el último punto de la solución
      Fil = SolFil[PosicionSolucion];
      Col = SolCol[PosicionSolucion];
      CambiaEstado(Fil,Col,AIRE);
      PosicionSolucion--;
```

```
      // Eliminamos los saltos que hemos dado desde él
      Punto = Indice(Fil, Col);
      NumeroSaltosDesde[Punto] = 0;

      // Algunos problemas no tienen solución, como 3 3 1 0
      // Nos damos cuenta porque llegamos a eliminar el primer punto que nos
      // dieron y la posición de la solución está en -1
      if ( PosicionSolucion == -1 )
        {
        // En ese caso, nos vamos a la celda (0,0), que siempre da solución
        Fil = 0;
        Col = 0;
        AlargaSolucion(Fil, Col);
        }

      // Normalmente no pasa
      else
        {
        // Seguimos probando con el punto anterior
        Fil = SolFil[PosicionSolucion];
        Col = SolCol[PosicionSolucion];
        }
      }
    } // Fin while !CurvaCompleta
  }

//---------------------------------------------------------------------------
// Muestra la solución encontrada
void MuestraSolucion(int Tamano, int Flag)
  {
  int i;
  celda Fil, Col;

  for (i=0; i<=PosicionSolucion; i++)
    {
    Fil = SolFil[i];
    Col = SolCol[i];
    CambiaPuntoMatriz(Tamano, Fil, Col, Flag);
    delay(RETARDO);
    }
  }

//---------------------------------------------------------------------------
// Decide si se puede pasar de un punto a otro
int SaltoAceptable(celda F1, celda C1, celda F2, celda C2)
  {
  int Respuesta = SI;

  // El punto de llegada debe estar dentro del rectángulo
  if ( F2<0 || F2>=MaxFil || C2<0 || C2>=MaxCol )
    { Respuesta = NO; }

  // No debemos haber pasado aún por el punto de llegada
  else if ( DiEstado(F2,C2) == TIERRA )
    { Respuesta = NO; }

  // No debemos haber probado ya este salto
  else if ( SaltoYaProbado(F1,C1,F2,C2) )
    { Respuesta = NO; }

  // Tendríamos que dejar una zona conexa para poder terminar
  else if ( (PosicionSolucion < Total-2) && (!RestoConexo(F2,C2)) )
    { Respuesta = NO; }

  // Contestamos
  return Respuesta;
  }

//---------------------------------------------------------------------------
// Dice si al añadir una celda a la solución quedaría un resto conexo
int RestoConexo(celda Fil, celda Col)
  {
  int i, PosicionInundacion, CeldasPorDesbordar;
  int Encontrada, InundacionTerminada, Respuesta;
  celda Celda, Vecina;

  // Añadimos temporalmente TIERRA en la celda que nos pasan, para probar
  CambiaEstado(Fil, Col, TIERRA);
```

```
  // Elegimos una celda cualquiera que tenga AIRE para comenzar una inundación
  Encontrada = NO;
  for ( i=0; i<Total && !Encontrada ; i++ )
    {
    if ( Estado[i] == AIRE )
      {
      Encontrada = SI;
      Estado[i] = AGUA;
      PosicionInundacion = 0;
      ListaInundadas[PosicionInundacion] = i;
      }
    }

  // A partir de esa celda, continuamos inundando las celdas vecinas
  // hasta que no se pueda inundar nada más
  CeldasPorDesbordar = 0;
  InundacionTerminada = NO;
  while ( !InundacionTerminada )
    {
    Celda = ListaInundadas[PosicionInundacion];
    // Inundamos todas sus celdas vecinas
    for ( i=0 ; i<NumVecinas[Celda] ; i++ )
      {
      Vecina = ListaVecinas[MAX_DIR*Celda+i];
      if ( Estado[Vecina] == AIRE )
        {
        Estado[Vecina] = AGUA;
        CeldasPorDesbordar++;
        ListaInundadas[PosicionInundacion+CeldasPorDesbordar] = Vecina;
        }
      }
    // Ya hemos desbordado una celda más
    PosicionInundacion++;

    // Y nos queda una menos por desbordar
    CeldasPorDesbordar--;

    // Vemos si hemos acabado la inundación
    if ( CeldasPorDesbordar == -1 )
      { InundacionTerminada = SI; }
    }

  // Vemos si hemos podido inundar todas las celdas
  if ( PosicionSolucion + PosicionInundacion == Total-2 )
    { Respuesta = SI; }
  else
    { Respuesta = NO; }

  // Eliminamos toda el AGUA que hemos puesto
  for ( i=0; i<Total; i++ )
    {
    if ( Estado[i] == AGUA )
      { Estado[i] = AIRE; }
    }

  // Devolvemos la celda al estado que tenía
  CambiaEstado(Fil, Col, AIRE);

  // Respondemos
  return Respuesta;
  }

//---------------------------------------------------------------------------
// Anota un salto de un punto a otro
int AnotaSalto(celda F1, celda C1, celda F2, celda C2)
  {
  int Origen, Destino, Base;

  // Convertimos en cada punto las dos dimensiones en una
  Origen = Indice(F1,C1);
  Destino = Indice(F2,C2);

  // La dirección base de las anotaciones de los saltos desde Origen
  Base = MAX_DIR * Origen;

  // Anotamos el salto
  SaltoDesde[Base+NumeroSaltosDesde[Origen]] = Destino;
```

```
  NumeroSaltosDesde[Origen]++;
  }

//---------------------------------------------------------------------------
// Dice si ya se ha dado un salto de un punto a otro
int SaltoYaProbado(celda F1, celda C1, celda F2, celda C2)
  {
  int i, Base, Origen, Destino, Respuesta, SigueBuscando;
  Respuesta = NO;
  SigueBuscando = SI;

  // Convertimos en cada punto las dos dimensiones en una
  Origen = Indice(F1,C1);
  Destino = Indice(F2,C2);

  // La dirección base de las anotaciones de los saltos desde Origen
  Base = MAX_DIR * Origen;

  for ( i=0; i<NumeroSaltosDesde[Origen] && SigueBuscando; i++ )
    {
    if ( SaltoDesde[Base+i] == Destino )
      {
      Respuesta = SI;
      SigueBuscando = NO;
      }
    }

  // Contestamos
  return Respuesta;
  }

//---------------------------------------------------------------------------
// Alarga la solución
void AlargaSolucion(celda Fil, celda Col)
  {
  PosicionSolucion++;
  SolFil[PosicionSolucion]=Fil;
  SolCol[PosicionSolucion]=Col;
  CambiaEstado(Fil, Col, TIERRA);
  }

//---------------------------------------------------------------------------
// Revuelve el array Permutacion
void RevuelveDirecciones(void)
  {
  int i, j, Valor;

  for (i = 0; i < MAX_DIR-1; i++)
    {
    j = random(0,MAX_DIR-1);
    Valor = Permutacion[j];
    Permutacion[j] = Permutacion[i];
    Permutacion[i] = Valor;
    }
  }

//---------------------------------------------------------------------------
// Cambia el estado de un punto de la matriz virtual
void CambiaPuntoMatriz(int Tamano, int fil, int col, int estado)
  {
  if ( Tamano == 6 )
    { CambiaPuntoMatrizFisico(fil+1,col+1,estado); }

  if ( Tamano == 4 )
    {
    CambiaPuntoMatrizFisico(2*fil,2*col,estado);
    CambiaPuntoMatrizFisico(2*fil+1,2*col,estado);
    CambiaPuntoMatrizFisico(2*fil,2*col+1,estado);
    CambiaPuntoMatrizFisico(2*fil+1,2*col+1,estado);
    }
  }

//---------------------------------------------------------------------------
// Cambia el estado de un punto de la matriz
// Con esta función arreglo que las conexiones no sean las habituales
void CambiaPuntoMatrizFisico(int fil, int col, int estado)
  { ControladorMatriz.setLed(NUM_MATRIZ,7-col,fil,estado); }
```