Práctica 4.1 - Backtracking y Branch And Bound
Problema del Laberinto

Adrián Portillo Sánchez
Jose Juán Pérez González

May 2015

1. Introducción al Problema.

Para esta práctica se nos pide utilizar técnicas de Backtracking (vuelta atrás) para encontrar la salida de un laberinto.
Este laberinto estará representado como una matriz de *n filas x n columnas* que poseerá cada una de sus celdas el valor 0 si es transitable y el valor 1 si no, y con un punto de entrada y uno de salida.
Este algoritmo poseerá dos niveles de mejora, en la primera solución aportada, el algoritmo solamente se dedicará a encontrar una solución, y una vez encontrado un camino válido, terminará y mostrará dicha solución en pantalla; en el segundo nivel de mejora, el algoritmo deberá ser capaz de recorrer todos los caminos posibles y decidir de entre ellos cual es el camino más corto de entre todos los caminos posibles que el algoritmo pueda tomar, y ese camino es el que mostrará en la salida.

Para probar el problema hemos creado 3 laberintos de prueba, de un tamaño entre 12 y 20 filas y columnas, sobre los que ejecutaremos el algoritmo y calcularemos las medidas de tiempo de cada una de las soluciones.
Por ejemplo, uno ejemplo de los laberintos creados y un laberinto y una solución en el programa son las siguientes:

```
1 1 1 1 1 1 1 1 1 1 1 1
1 0 0 0 1 0 0 0 0 0 0 1
0 0 1 0 1 0 1 1 1 1 0 1
1 1 1 0 1 0 0 0 0 1 0 1
1 0 0 0 0 1 1 1 0 1 0 0
1 1 1 1 0 1 0 1 0 1 0 1
1 0 0 1 0 1 0 1 0 1 0 1
1 1 0 1 0 1 0 1 0 1 0 1
1 0 0 0 0 0 0 0 0 1 0 1
1 1 1 1 1 1 0 1 1 1 0 1
1 0 0 0 0 0 0 1 0 0 0 1
1 1 1 1 1 1 1 1 1 1 1 1
```

```
                Coste:
                43
                Solucion:
001111111111    221111111111
100010000001    122212222221
101010111101    101212111121
111010000101    111212222121
100001110101    100221112121
111101010101    111121012121
100101010101    100121012121
110101010101    110121012121
100000000101    100022222121
111111011101    111111011121
100000010001    100000010021
111111111100    111111111122
```

2. Funciones comunes en los algoritmos.

Los algoritmos creados poseen unas funciones comunes que servirán para el correcto desarrollo del programa, dichas funciones son las siguientes:

2.1. mostrar()

Este procedimiento mostrará el estado actual del laberinto, se utilizará tras las dos resoluciones para mostrar los resultados del camino creado.

```
void mostrar() {
        int i, j;

        for (i=0; i<FILAS; i++) {
        for (j=0; j<COLUMNAS; j++) {
                cout << laberinto[i][j];
        }
        cout << endl;
        }
    cout << endl;
}
```

2.2. reiniciar()

Este procedimiento retornará el laberinto al valor inicial, se hará cada vez que se modifique el laberinto para crear un camino o para recorrerlo de forma completa.

```
void reiniciar() {
        int i, j;

        for (i=0; i<FILAS; i++) {
        for (j=0; j<COLUMNAS; j++) {
                if(laberinto[i][j] == 2 || laberinto[i][j] == 3)
                laberinto[i][j] = 0;
        }
    }

}
```

2.3. factible(int fil, int col)

Esta función devolverá si la posición que se le pasa como parámetros es una posición del laberinto transitable o no, primero comprobará si la posición esta dentro de la matriz, y después si dicha posición es un muro o ya ha sido recorrida previamente.

```
int factible(int f, int c) {
   bool resultado = true;

   if ((f<0) || (f>=FILAS) || (c<0) || (c>=COLUMNAS))
      resultado = false;

   if (laberinto[f][c] == 3 || laberinto[f][c] == 2 || laberinto[f][c] == 1)
      resultado = false;

   return resultado;
}
```

3. Main del programa.

En el main del programa se inicializan primero las variables, en segundo lugar se muestra el laberinto sin resolver, luego se llama a la función recorrer que resuelve el laberinto para un coste concreto o un coste sin concretar, y esta crea el laberinto resuelto y modifica la variable coste por la del coste del camino tomado.
Luego se muestra la solución de la función sin un coste concreto, es decir, la primera que encuentre y el coste de dicha solución, para luego reiniciar el laberinto.
Para la segunda solución se llama a recorrer_completo, función que recorre el laberinto completo y cada vez que encuentra una solución guarda el coste, encontrando así el coste mínimo, luego se reinicia el laberinto, y se vuelve a llamar a recorrer, esta vez pasando como parámetro el coste mínimo, para que encuentre la solución de dicho coste.
Por último mostrará dicha solución y su coste.

```
int main() {
   bool solucion; int coste, coste_minimo = 10000;

   mostrar();   // Laberinto sin resolver.

   solucion = recorrer(x_entrada,y_entrada, coste, 0);
   cout << "\nCoste: \n" << coste+2 <<"\nSolucion: \n" ;
   mostrar();   // Laberinto resuelto.

   reiniciar();   coste = 0;

   recorrer_completo(x_entrada, y_entrada, coste, coste_minimo);
   reiniciar();   coste = 0;
   solucion = recorrer(x_entrada,y_entrada, coste, coste_minimo);
   cout << "\nCoste Minimo: \n" << coste <<"\nSolucion: \n" ;
   mostrar();   // Camino minimo.
}
```

4. Primera Solución del Algoritmo

La primera solución que el algoritmo aporta será la que encuentre recorriendo el laberinto mediante Backtracking hasta llegar a una solución, una vez que llega a la solución la función acaba la función, y el laberinto queda modificado con el camino encontrado formado por 2.
Esta función pone la posición actual a 2 y luego actúa de forma recursiva, teniendo como caso base haber llegado a la solución, y como caso general no haber llegado, en el caso general se efectuán 4 condiciones, primero se comprobará si se puede recorrer el laberinto hacia arriba mediante una llamada a la función factible(), si se puede, se hará una llamada recursiva a recorrer con la posición de arriba, lo mismo para el resto de direcciones, y por último si ya se ha pasado por las cuatro direcciones y el algoritmo aún no ha acabado, se realizará la vuelta atrás (de ahí la lógica backtracking), en la cual, desenrollando llamadas recursivas, se cambiará el valor de la posición actual por 0.

```
bool recorrer(int fil, int col, int & coste, int coste_busqueda) {
    bool listo = false;

    coste++;

    // Casilla visitada
    laberinto[fil][col] = 2;

    if (fil == x_salida && col == y_salida
    && (coste_busqueda == coste || coste_busqueda == 0)){
        listo = true;
    }
    else{
        if (!listo && factible(fil,col-1))    // Continuamos para abajo
            listo = recorrer(fil,col-1, coste, coste_busqueda);
        if (!listo && factible(fil,col+1))    // Continuamos para arriba
            listo = recorrer(fil,col+1, coste, coste_busqueda);
        if (!listo && factible(fil-1,col))    // Continuamos para la izquierda
            listo = recorrer(fil-1,col, coste, coste_busqueda);
        if (!listo && factible(fil+1,col))    // Continuamos para la derecha
            listo = recorrer(fil+1,col, coste, coste_busqueda);

        if (!listo){
            laberinto[fil][col] = 0;
            coste--;
        }
    }
    // Devuelve true si se recorre con exito y false en caso contrario
    return listo;
}
```

5. Camino Mínimo del Laberinto.

La segunda solución que el algoritmo aporta encuentra el camino mínimo, esto lo hace recorriendo el algoritmo mediante Backtracking de la misma forma que la anterior función, sólo que esta vez no se para en la salida si no al terminar de recorrer todos los caminos, para ellos el caso base será poner la entrada a 3, ya que al volver en lugar de modificar la casilla a 0 de nuevo, la pone a 3, para indicar que por ahí ya ha pasado, y una vez vuelve al punto de entrada, habrá recorrido todos los caminos posibles del laberinto.
Cada vez que pasa por la salida, el laberinto guardará el coste del nuevo camino, y si es menor que el coste guardado anteriormente como mínimo, este será el nuevo mínimo.
Después de esto se ejecuta de nuevo el recorrer esta vez pasándole como parámetro el coste mínimo, para que encuentre el camino de ese coste, y lo muestre.

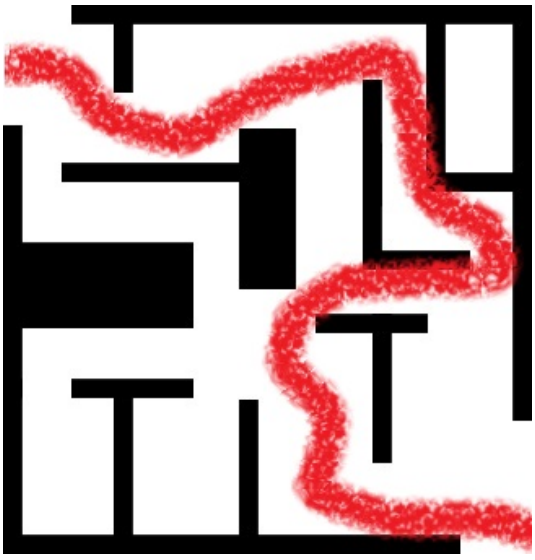
```
bool recorrer_completo(int fil, int col, int & coste, int & menor_coste){
    bool listo = false;
    coste++;

    laberinto[fil][col] = 2;

    if (fil == x_salida && col == y_salida){
        if (coste < menor_coste)
            menor_coste = coste;
    }

    if( laberinto[x_entrada][y_entrada] == 3 )
        listo = true;
    else{
        if (!listo && factible(fil,col-1))    // Continuamos para abajo
            listo = recorrer_completo(fil,col-1, coste, menor_coste);
        if (!listo && factible(fil,col+1))    // Continuamos para arriba
            listo = recorrer_completo(fil,col+1, coste, menor_coste);
        if (!listo && factible(fil-1,col))    // Continuamos para la izquierda
            listo = recorrer_completo(fil-1,col, coste, menor_coste);
        if (!listo && factible(fil+1,col))    // Continuamos para la derecha
            listo = recorrer_completo(fil+1,col, coste, menor_coste);

        if (!listo){
            laberinto[fil][col] = 3;
            coste--;
        }
    }
    // Devuelve true si se recorre con exito y false en caso contrario
    return listo;
}
```

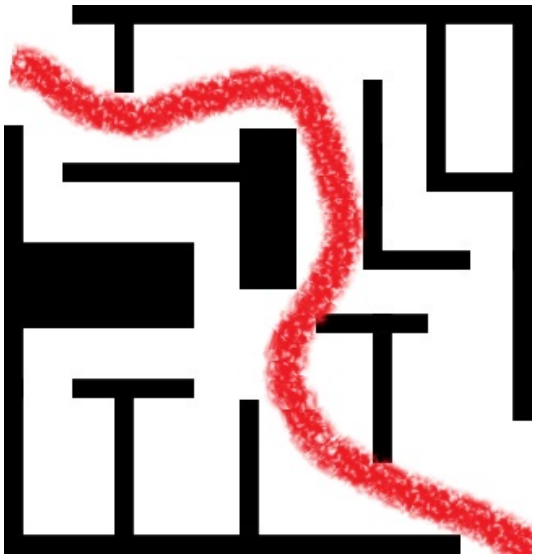
6. Resultados del Programa para las Dos mejoras.

6.1. Laberinto 1.

Para el laberinto 1 el algoritmo devuelve el siguiente resultado

```
                Coste:                Coste Minimo:
                45                    31
                Solucion:             Solucion:
0011111111111111 2211111111111111 2211111111111111
1001010001000102 1221012221222102 1221010001000102
1101000100010102 1121222122212102 1121222100010102
1000010101010102 1022210101012102 1022212101010102
1011110111010112 1011110111012112 1011112111010112
1000100011010002 1000100011012202 1000102211010002
1111111011011012 1111111011011212 1111111211011012
1111111011000012 1111111011022212 1111111211000012
1111111011101012 1111111011121012 1111111211101012
1000000010001012 1000000012221012 1000000210001012
1011111000111012 1011111002111012 1011111222111012
1001000010010002 1001000012210002 1001000012210002
1001000010010002 1001000012210002 1001000010210002
1001000010000002 1001000012222222 1001000010222222
1111111111111102 1111111111111112 1111111111111112
```

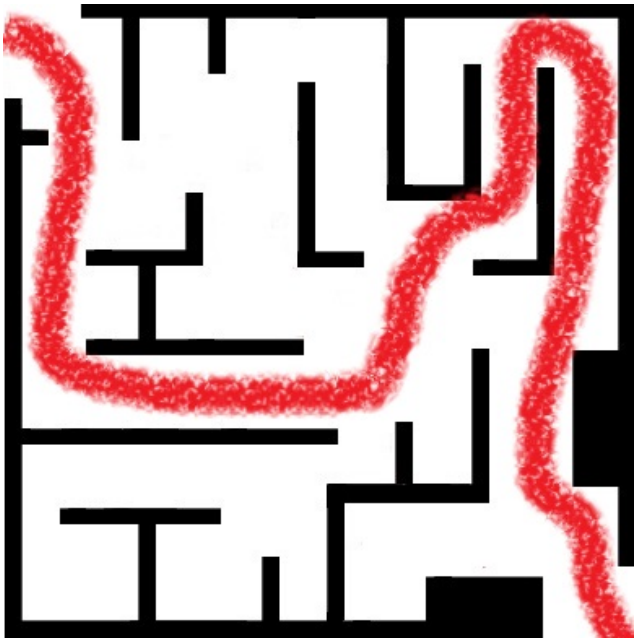


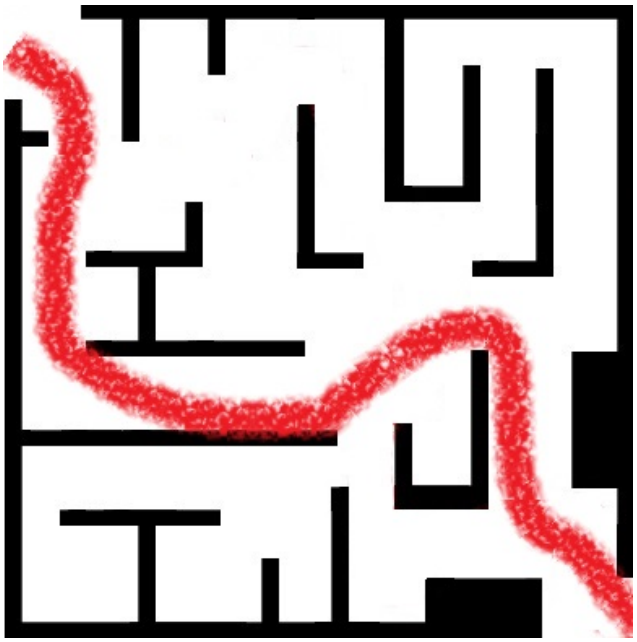


6.2. Laberinto 2.

Para el laberinto 2 el algoritmo devuelve el siguiente resultado

```
                    Coste:
                    51
                    Solucion:
00111111111111111111 22111111111111111111
10010101010001000001 12210101010001002222 1
11010001000101010010 11210001000101012212 1
10000101110101011010 12200101110101011212 1
10111101010101110010 12111101010101112212 1
10001001010100000110 12001001010102222112 1
10111111010110100000 12111111010112100022 1
10000000000000010110 12222222222222101121 1
11111111111010100101 11111111111010100121 1
10000000100011111101 10000000100011111121 1
10111110001110100000 10111110001110100022 1
10010000100100001110 10010000100100001112 1
11111111111111111100 11111111111111111122
```

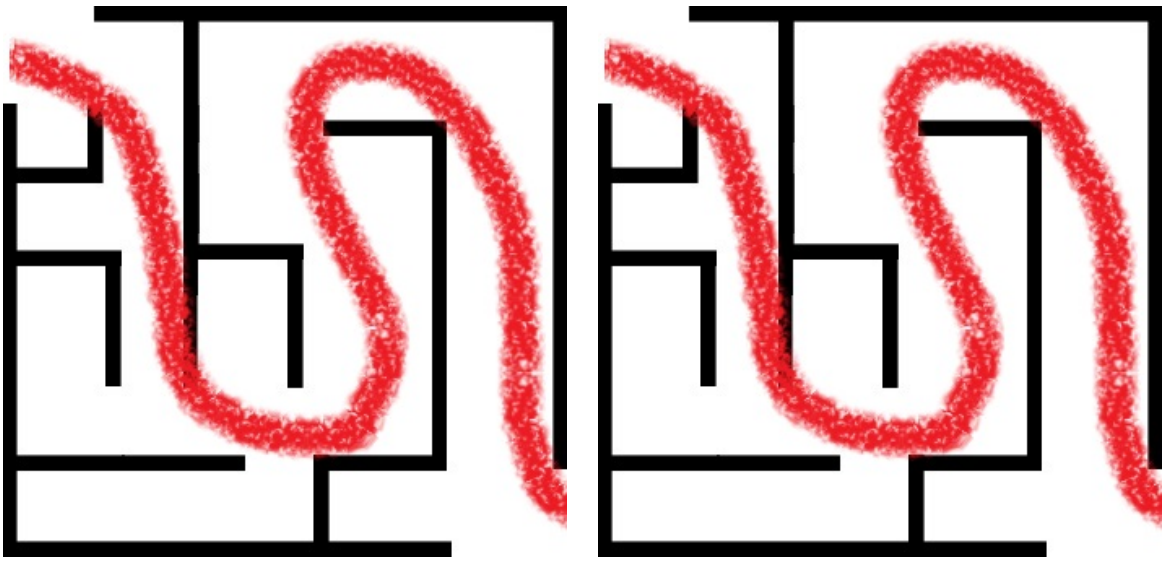




6.3. Laberinto 3.

Para el laberinto 3 el algoritmo devuelve el siguiente resultado

	Coste:	Coste Minimo:
	43	43
	Solucion:	Solucion:
001111111111	221111111111	221111111111
100010000001	122212222221	122212222221
101010111101	101212111121	101212111121
111010000101	111212222121	111212222121
100001110101	100221112121	100221112121
111101010101	111121012121	111121012121
100101010101	100121012121	100121012121
110101010101	110121012121	110121012121
100000000101	100022222121	100022222121
111111011101	111111011121	111111011121
100000010001	100000010021	100000010021
111111111100	111111111122	111111111122



	Laberinto1	Laberinto2	Laberinto3
Primera solución	5,13E-05	1,32E-05	7,47E-06
Solución mas corta	5,58E-03	9,39E-05	1,81E-05

Como podemos observar en la tabla, la modificación del algoritmo para encontrar el camino mas corto es significativamente mas lento encontrando la salida para el Laberinto1 que el primer algoritmo, para el Laberinto2, sigue siendo mas lento, aunque no hay tanta diferencia como para Laberinto1. Por ultimo para el tercer laberinto podemos ver como tarda menos en encontrar la salida el algoritmo modificado.