GRADO EN INGENIERÍA INFORMÁTICA

CURSO 2023-2024

UNIVERSIDAD
DE GRANADA

ALGORÍTMICA

Práctica 4 - Algoritmo Backtracking

Miguel Martinez Azor

Ángel Rodríguez Faya

Alejandro Botaro Crespo

Alberto Parejo Bellido

Alejandro Ocaña Sánchez

14/05/2024

ÍNDICE

Problema 1

Supongamos que tenemos n estudiantes en una clase y queremos crear con ellos equipos formados por parejas (podemos suponer que n es un número par). Se dispone de una matriz p de tamaño n×n en la que p(i, j) indica el nivel de preferencia que el estudiante i tiene para trabajar con el estudiante j. El valor del emparejamiento del estudiante i con el j es p(i, j) ∗ p(j, i). Se trata de encontrar un emparejamiento para todos los estudiantes de forma que se maximice la suma de los valores de los emparejamientos. Diseñar e implementar un algoritmo de backtracking para resolver el problema.

1.1.-Enunciado.

- Existen n estudiantes en una clase y queremos formar equipos de parejas, maximizando la suma de los valores de emparejamiento.
- DIsponemos de una matriz p de n*n filas/columnas donde cada posición de la misma p[i,j] corresponde al nivel de preferencia de emparejar al estudiante i con el estudiante j.

1.2.-Enunciado Formal.

- Dada una matriz p de $n \times n$, encontrar el emparejamiento de todos los estudiantes de manera que se maximice la suma de los valores de los emparejamientos.

1.3.-Vuelta atrás: Componentes de diseño.

Representación:

- El problema se puede representar como un grafo no dirigido donde los nodos representan a los estudiantes y las aristas representan las preferencias entre ellos. El objetivo es encontrar un camino en este grafo que maximice la suma de los valores de los emparejamientos.
- La solución estará formada por una tupla (X1,X2,X3,...,Xn) donde cada i corresponde al emparejamiento del estudiante i con Xi

Restricciones Implícitas:

- Maximización de la suma de los valores de los emparejamientos: El objetivo es maximizar la suma de los valores de los emparejamientos.

Restricciones Explícitas:

- Un estudiante no puede ser emparejado con otro estudiante que ya está emparejado.
- El número de emparejamientos debe ser par ya que queremos formar parejas.
- Un estudiante no puede ser emparejado con otro estudiante más de una vez.
- Un estudiante no puede ser emparejado consigo mismo

Pseudocódigo:

```
Function Bestudiantes(p[1-n][1-n], parejas[n], estudiante) begin
    Si restantes = 0 then
        Si sumaTotal > mejorSuma then
            mejorSuma = sumaTotal
        end
    Para cada estudiante i en restantes:
        Para cada estudiante j en restantes, distinto de i:
            //Emparejar i con j
            parejas[i] = j
            parejas[j] = i
            //Calcular la conveniencia de la pareja (i, j)
            conveniencia = p[i][j] * p[j][i]
            Sí conveniencia > mejorSuma:
                Si i es el último estudiante en restantes:
                mejorSuma = conveniencia
            Sino:
                Quitar i y j de la lista de restantes
                Estudiantes(p,parejas,restantes,mejorSuma)
            //Desemparejar i y j
```

```
                    parejas[i] = 0
                     parejas[j] = 0
                    //Agregar i y j nuevamente a la lista de restantes
                end
        end
end
```

1.4.-Ejemplo paso a paso del funcionamiento.

Número de estudiantes (n) = 4

Matriz de preferencias p = 0, 6, 2, 4
6, 0, 6, 2
2, 6, 0, 6
4, 2, 6, 0

parejas[n] = 0

restantes[n] = {1, 2, 3, 4}

suma total = 0

mejor suma = 0

-Comenzamos escogiendo el primer estudiante 1, y lo emparejamos con el resto de estudiantes 1, 2, 3, 4.

-Pareja con el 1 -> no se puede ya que no cumple con las restricciones

-Pareja con el 2 -> parejas(i) = j
parejas(j) = i
p(1,2) x p(2,1) = 6*6 = 36
marcamos los estudiante 1 y 2 como emparejados

Los estudiantes restantes que quedan por emparejar son el 3 y 4

Repetimos el proceso con estos y su nivel de conveniencia es :

p(3,4) x p(4,3) = 6*6 = 36

Suma total = 36 +36 =72

Mejor Suma = 72

Vuelta atrás y continuamos con el estudiante 3

-Pareja con el 3 -> parejas(i) = j

parejas(j) = i

p(1,3) x p(3,1) = 2*2 = 4

marcamos los estudiante 1 y 3 como emparejados

Los estudiantes restantes que quedan por emparejar son el 2 y 4

Repetimos el proceso con estos y su nivel de conveniencia es :

p(2,4) x p(4,2) = 2*2 = 4

Suma total = 4+4 =8

Mejor suma = 72

Vuelta atrás y continuamos con el estudiante 4

-Pareja con el 4 -> parejas(i) = j

parejas(j) = i

p(1,4) x p(4,1) = 4*4 = 16

marcamos los estudiante 1 y 4 como emparejados

Los estudiantes restantes que quedan por emparejar son el 2 y 3

Repetimos el proceso con estos y su nivel de conveniencia es :

p(2,3) x p(3,2) = 6*6 = 36

Suma total = 16+36 =52

Mejor suma = 72

-Como ya no quedan más combinaciones el mejor resultado sería:

parejas ={ 2,1,4,3} -> (1,2) (3,4)

Conveniencia = 72

Problema 2

Se va a celebrar una cena de gala a la que asistirán n invitados. Todos se van a sentar alrededor de una única gran mesa circular, de forma que cada invitado tendrá sentados junto a él a otros dos comensales (uno a su izquierda y otro a su derecha). En función de las características de cada invitado (por ejemplo categoría o puesto, lugar de procedencia, ...) existen unas normas de protocolo que indican el nivel de conveniencia de que dos invitados se sienten en lugares contiguos (supondremos que dicho nivel es un número entero entre 0 y 100). El nivel de conveniencia total de una asignación de invitados a su puesto en la mesa es la suma de todos los niveles de conveniencia de cada invitado con cada uno de los dos invitados sentados a su lado. Se desea sentar a los invitados de forma que el nivel de conveniencia global sea lo mayor posible. Diseñar e implementar un algoritmo de backtracking para resolver el problema

2.1.-Enunciado.

- Existen n invitados a una cena y queremos sentar en una mesa circular dos comensales al lado de cada invitado, maximizando la conveniencia global de cada invitado con cada uno de los dos invitados sentados a su lado.

2.2.-Enunciado Formal.

- Dada una matriz M de $n \times n$, encontrar el emparejamiento de todos los comensales de manera que se maximice la suma de los valores de la convivencia.

2.3.-Vuelta atrás: Componentes de diseño.

Representación:

- El problema se puede representar como un grafo no dirigido donde los nodos representan a los comensales y las aristas representan las conveniencias entre ellos. La meta es encontrar un camino en este grafo que maximice la conveniencia global de los comensales.
- La solución estará formada por una tupla (X1, X2, X3, ..., Xn) donde cada i corresponde al emparejamiento del estudiante i con Xi.

Restricciones Implícitas:

- Maximización de la conveniencia global: El objetivo es maximizar la suma de los valores de conveniencia entre dos comensales.

Restricciones Explícitas:

- Un comensal no puede estar sentado junto a otro comensal que ya tenga sentado a un comensal a la derecha y otro a la izquierda.
- Un comensal no puede estar sentado dos o más veces.

Pseudocódigo:

```
// Función principal que llama al algoritmo de backtracking
Función AsignarAsientos(n, invitados):
    mejor_nivel ← -1
    MejorAsignación ← lista vacía
    AsignarAsientosBack(n, 0, invitados, [], 0, mejor_nivel,
MejorAsignación)
    Devolver MejorAsignación

// Función auxiliar recursiva para backtracking
Función AsignarAsientosBack(n, índice, invitados, asignación_actual, nivel_actual,
mejor_nivel, MejorAsignación):
    Si índice == n:
        Si nivel_actual > mejor_nivel:
            mejor_nivel ← nivel_actual
            MejorAsignación ← asignación_actual
        Devolver

    Para cada posición posible en la mesa:
    Si la posición está disponible:
        invitado_actual ← invitados[índice]
        asignación_actual[índice] ← posición
        nivel_actual ← nivel_actual + invitado_actual.conveniencia_izquierda +
invitado_actual.conveniencia_derecha
```

```
        AsignarAsientosBack(n, índice + 1, invitados, asignación_actual,
nivel_actual, mejor_nivel, MejorAsignación)

        nivel_actual ← nivel_actual - invitado_actual.conveniencia_izquierda -
invitado_actual.conveniencia_derecha

        asignación_actual[índice] ← posición no asignada
```

El pseudocódigo analiza todas la posibilidades de convivencia entre comensales y va asignando asientos hasta encontrar la mejor asignación y devolverla.

Profundizando un poquito en el algoritmo la función va sustituyendo llamándose a sí misma viendo si el nivel actual es mejor que el anterior, si es así la asignación dentro de ese nivel se guarda como mejor. Esto se hace repetidamente hasta ver todos los nodos y ver qué asignación queda como la mejor.

2.4.-Ejemplo paso a paso del funcionamiento.

En este ejemplo tenemos 5 invitados y 5 sitios en la mesa.

El nivel de conveniencia de sentar un invitado con otro es el siguiente:

Invitado 1	Invitado 2	Invitado 3	Invitado 4	Invitado 5	
X	10	20	30	40	Inv 1
10	X	15	25	35	Inv 2
20	15	X	10	20	Inv 3
30	25	10	X	15	Inv 4
40	35	20	15	X	Inv 5

1.- Partimos de la mesa vacía y invitados = 5

2.- Escogemos el invitado que maximiza la suma de niveles de conveniencia con los invitados ya asignados. Como la mesa está vacía, seleccionamos a cualquier invitado. Seleccionamos al invitado 1

3.- Al estar la mesa vacía el invitado 1 se puede sentar sin problema, es factible.

4.- Como quedan asientos por asignar continuamos escogiendo el siguiente candidato:

candidato 1 -> Inv 2, Suma de conveniencia -> 0+10 = 10

5.- Como todavía quedan invitados por sentar, escogemos otro candidato para sentar.

6.- Cogemos el siguiente candidato:

candidato 2 -> Inv 3, Suma de conveniencia -> 10+15 = 25

7.- Como todavía quedan invitados por sentar, escogemos otro candidato para sentar.

8.- Buscamos otro candidato con respecto al candidato 3:

candidato 3 -> Inv 4, Suma de conveniencia -> 25+10 = 35

9.- Como todavía quedan invitados por sentar, escogemos el siguiente candidato para sentar, que en este caso el único que queda es el 5.

Suma total = 15+35 = 50

10.- Como ya no quedan más invitados que sentar salimos de este primer ciclo en el que la conveniencia global es 50, que pasa a ser la mejor asignación. Así continuaremos una y otra vez hasta que no haya más posibilidades y solo quede devolver el valor de la mejor asignación.

Problema 3

3.1.-Enunciado.

En el juego (solitario) del senku, también llamado solitario chino, se colocan 32 piezas iguales en un tablero de 33 casillas, tal y como se indica en la siguiente figura (las “x" corresponden a posiciones no válidas):

x	x	o	o	o	x	x
x	x	o	o	o	x	x
o	o	o	o	o	o	o
o	o	o		o	o	o
o	o	o	o	o	o	o
x	x	o	o	o	x	x
x	x	o	o	o	x	x

Solo se permiten movimientos de las piezas en vertical y horizontal. Una pieza solo puede moverse saltando sobre otra y situándose en la siguiente casilla, que debe estar vacía. La pieza sobre la que se salta se retira del tablero. Se consigue terminar con éxito el juego cuando queda una sola pieza en la posición central del tablero (la que estaba inicialmente vacía). Diseñar e implementar un algoritmo de backtracking que encuentre una serie de movimientos para llegar con éxito al final del juego.

3.2.-Enunciado Formal.

Dado un tablero de tamaño n, tenemos que encontrar una solución en la que sólo quede una bola (también llamada “canica” o “cinco”) y esté en el centro del tablero. Solamente se pueden realizar movimientos para que una bola se coma a otra, de forma que la bola que es “comida” por la otra, queda eliminada. En el tablero sólo hay 33 casillas disponibles, las cuáles 32 están ocupadas por bolas y la casilla del centro está vacía. Como tenemos 32 bolas, la solución tiene que estar formada por 31 movimientos.

3.3.-Vuelta atrás: Componentes de diseño.

- **Representación:** El estado del tablero se representa como una matriz tablero de tamaño nxn, donde cada casilla puede ser libre, ocupada o prohibida. Cada movimiento se representa como un par de coordenadas (i, j) que indica la posición de la ficha que se moverá y un par de coordenadas (ni, nj) que indica la posición a la que se moverá la ficha.

- **Restricciones explícitas:** Las coordenadas deben estar dentro del rango del tablero, es decir, 0 <= i, j, ni, nj < n. Además, el movimiento debe ser válido según las reglas del juego.
- **Restricciones implícitas:**

    - **No puede haber más de una ficha en una casilla ocupada:** Para cada movimiento, la casilla de origen (i, j) debe contener una ficha (ocupada) y la casilla de destino (ni, nj) debe estar libre (libre).

    - **Una ficha sólo puede moverse en línea recta sobre fichas ocupadas:** Para cada movimiento, las casillas intermedias entre la posición de origen (i, j) y la posición de destino (ni, nj) deben estar ocupadas (ocupada).

- **Pseudocódigo:**

```
n = 7 ← tamaño de la matriz

contador = 0 ← variable contador

tipoCasilla = libre, ocupada, prohibida ← enumerado

tablero[n][n] ←tablero del juego de tipoCasilla

Función resolverSolitarioChino

        Si fin() devuelve true;

        Para i desde 0 hasta n -1, i++

                Para j desde 0 hasta n-1, j++

                        Si tablero[i][j] == ocupada

                                movimientos[4][2] = {{0, -2}, {-2, 0}, {0, 2}, {2,
                                0}}; ← se inicializa con los valores que
                                representan como quedaría la fila y columna
                                de cada movimiento, (en el orden izquierda,
                                arriba, derecha y abajo), que serían:

                                Para k desde 0 hasta 3 incluido, k++

                                        ni = i + movimientos[k][0] ←
                                        coordenada x de la casilla destino

                                        nj = j + movimientos[k][1] ←
                                        coordenada y de la casilla destino

                                        Si (esMovimientoValido(i, j, ni, nj))
```

```
                                    hacerMovimiento(i, j, ni, nj)
                                        Si resolverSolitarioChino()
                                            contador + 1
                                            Mostrar  movimiento y
                                            coordenadas origen y destino
                                            devolver true
                                        Fin SI
                                    deshacerMovimiento(i, j, ni, nj)
                                Fin SI
                            Fin Para
                        Fin Si
                    Fin Para
                Fin Para
            Fin función
```

3.4.-Ejemplo paso a paso del funcionamiento.

En este problema, como ya se ha indicado anteriormente, hay 32 casillas, sólo se pueden realizar 31 movimientos, y nuestro algoritmo es los que hace, ni uno de más ni uno de menos. En este ejemplo vamos a ver los primeros 5 movimientos:

Movimiento nº1 : (5, 3) a (3, 3)

Movimiento nº2 : (4, 5) a (4, 3)

Movimiento nº3 : (2, 4) a (4, 4)

Movimiento nº4 : (4, 3) a (4, 5)

Movimiento nº5 : (4, 1) a (4, 3)

De forma resumida vuelvo a explicar esto para entrar en contexto: una vez dentro del cuerpo de la función que resuelve el juego, que es la función resolverSolitarioChino como su nombre indica. Empezamos con los dos bucles for que nos van a ir recorriendo para la fila y columna del tablero, casilla por casilla. Una vez dentro encontramos otro bucle for que controla la variable k para elegir un movimiento.

Una vez aquí dentro, comprobamos si es un movimiento válido, y en caso afirmativo hacemos una llamada recursiva a esta función (resolverSolitarioChino) que si nos devuelve true, se incrementa la variable

contador y nos muestra lo que estoy mostrando de movimiento nº… y todo eso, y devolverá true, es decir, devolverá un movimiento válido.

Pues bien, una vez visto esto,vamos a ver cómo se resuelve este juego. Primero de todo indicar que nos encontramos el tablero así:

	0	1	2	3	4	5	6
0	X	X	O	O	O	X	X
1	X	X	O	O	O	X	X
2	O	O	O	O	O	O	O
3	O	O	O		O	O	O
4	O	O	O	O	O	O	O
5	X	X	O	O	O	X	X
6	X	X	O	O	O	X	X

- **Movimiento nº1 : (5, 3) a (3, 3).**

  Nos quedaría el tablero de esta forma:

	0	1	2	3	4	5	6
0	X	X	O	O	O	X	X
1	X	X	O	O	O	X	X
2	O	O	O	O	O	O	O
3	O	O	O	O	O	O	O
4	O	O	O		O	O	O
5	X	X	O		O	X	X
6	X	X	O	O	O	X	X

- **Movimiento nº2 : (4, 5) a (4, 3)**

  Nos quedaría el tablero de esta forma:

	0	1	2	3	4	5	6
0	X	X	O	O	O	X	X
1	X	X	O	O	O	X	X
2	O	O	O	O	O	O	O
3	O	O	O	O	O	O	O
4	O	O	O	O			O
5	X	X	O		O	X	X
6	X	X	O	O	O	X	X

- **Movimiento nº3 : (2, 4) a (4, 4)**

  Nos quedaría el tablero de esta forma:

	0	1	2	3	4	5	6
0	X	X	O	O	O	X	X
1	X	X	O	O	O	X	X
2	O	O	O	O	O	O	O
3	O	O	O	O	O	O	O
4	O	O	O	O	O		O
5	X	X	O		O	X	X
6	X	X	O	O	O	X	X

- **Movimiento nº4 : (4, 3) a (4, 5)**

  Nos quedaría el tablero de esta forma:

	0	1	2	3	4	5	6
0	X	X	O	O	O	X	X
1	X	X	O	O	O	X	X
2	O	O	O	O	O	O	O
3	O	O	O	O	O	O	O
4	O	O	O			O	O
5	X	X	O		O	X	X
6	X	X	O	O	O	X	X

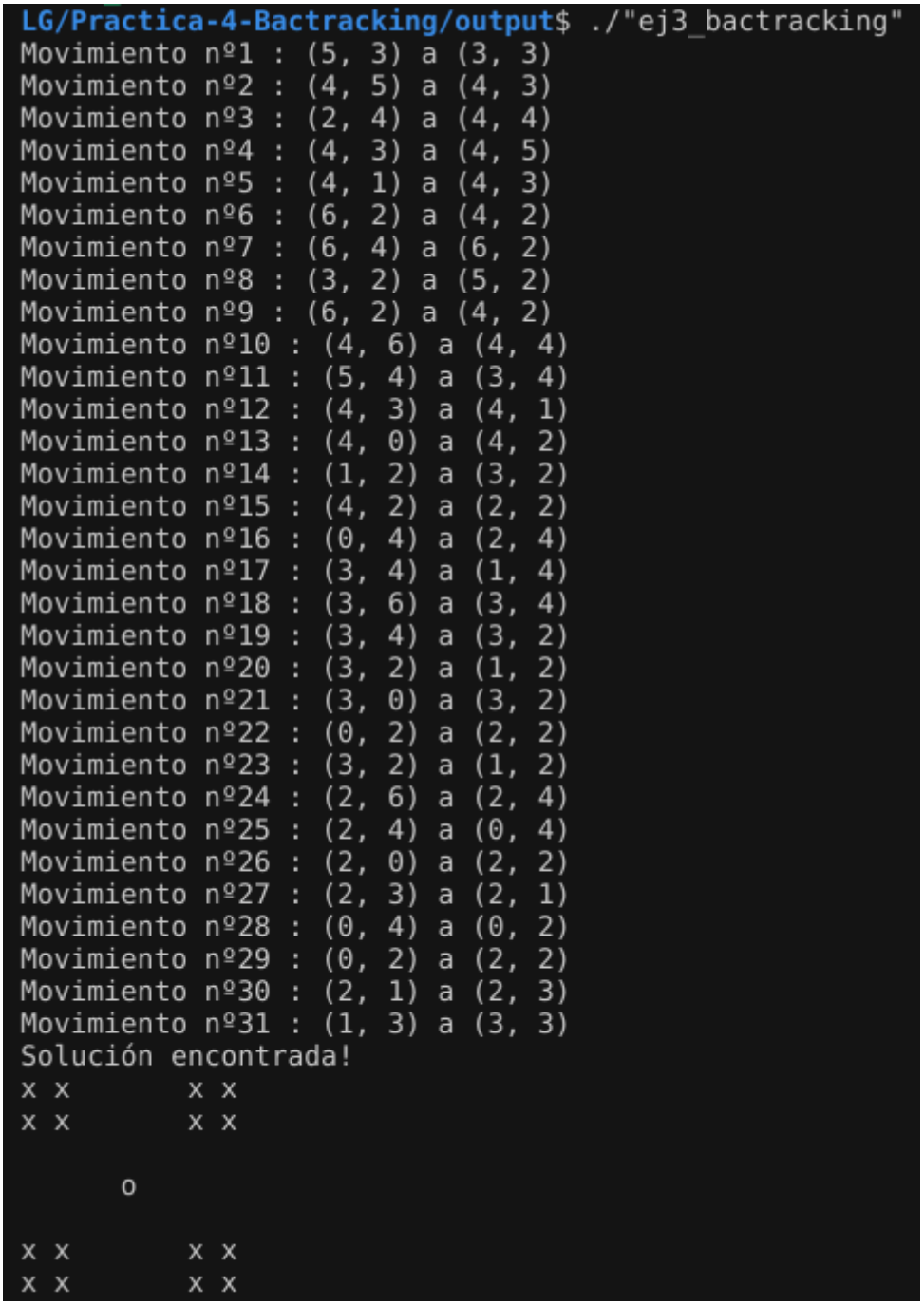
- **Movimiento nº5 : (4, 1) a (4, 3)**

  Nos quedaría el tablero de esta forma:

	0	1	2	3	4	5	6
0	X	X	O	O	O	X	X
1	X	X	O	O	O	X	X
2	O	O	O	O	O	O	O
3	O	O	O	O	O	O	O
4	O			O		O	O
5	X	X	O		O	X	X
6	X	X	O	O	O	X	X

3.5.-Ejemplo de ejecución.

Si ejecutamos el programa como: ./ej3_bactracking nos daría la siguiente salida:


LG/Practica-4-Bactracking/output$ ./"ej3_bactracking"
Movimiento nº1 : (5, 3) a (3, 3)
Movimiento nº2 : (4, 5) a (4, 3)
Movimiento nº3 : (2, 4) a (4, 4)
Movimiento nº4 : (4, 3) a (4, 5)
Movimiento nº5 : (4, 1) a (4, 3)
Movimiento nº6 : (6, 2) a (4, 2)
Movimiento nº7 : (6, 4) a (6, 2)
Movimiento nº8 : (3, 2) a (5, 2)
Movimiento nº9 : (6, 2) a (4, 2)
Movimiento nº10 : (4, 6) a (4, 4)
Movimiento nº11 : (5, 4) a (3, 4)
Movimiento nº12 : (4, 3) a (4, 1)
Movimiento nº13 : (4, 0) a (4, 2)
Movimiento nº14 : (1, 2) a (3, 2)
Movimiento nº15 : (4, 2) a (2, 2)
Movimiento nº16 : (0, 4) a (2, 4)
Movimiento nº17 : (3, 4) a (1, 4)
Movimiento nº18 : (3, 6) a (3, 4)
Movimiento nº19 : (3, 4) a (3, 2)
Movimiento nº20 : (3, 2) a (1, 2)
Movimiento nº21 : (3, 0) a (3, 2)
Movimiento nº22 : (0, 2) a (2, 2)
Movimiento nº23 : (3, 2) a (1, 2)
Movimiento nº24 : (2, 6) a (2, 4)
Movimiento nº25 : (2, 4) a (0, 4)
Movimiento nº26 : (2, 0) a (2, 2)
Movimiento nº27 : (2, 3) a (2, 1)
Movimiento nº28 : (0, 4) a (0, 2)
Movimiento nº29 : (0, 2) a (2, 2)
Movimiento nº30 : (2, 1) a (2, 3)
Movimiento nº31 : (1, 3) a (3, 3)
Solución encontrada!
x x x x
x x x x
o
x x x x
x x x x

Problema 4

El problema consiste en encontrar la salida de un laberinto. Más concretamente, supondremos que el laberinto se representa mediante una matriz cuadrada bidimensional de tamaño n X n. Cada posición almacena un valor boolenao “true” si la casilla es transitable y “false” si la casilla no es transitable. Los movimientos permitidos son a casillas adyacentes de la misma fila o la misma columna. Podemos suponer que las casillas de entrada y salida del laberinto son (0,0) y (n -1, n -1) respectivamente. Por tanto, el problema consiste en, dada una matriz que representa el laberinto, encontrar si existe un camino para ir desde la entrada hasta la salida.

1. Indicar las restricciones (implícitas y explícitas) que nos aseguren un árbol de estados finito para el problema.

2. Diseñar e implementar un algoritmo vuelta atrás para resolver el problema (backtracking).

4.1.-Enunciado.

- Se desea llegar desde la casilla inicial a la final de laberinto
- A partir de la primera casilla se debe encontrar un camino solución transitable hasta la casilla objetivo

4.2.-Enunciado Formal.

- Dada una matriz nXn con posiciones de 0,0(casilla inicio) a n-1,n-1(casilla salida) de booleanos con valores true para posiciones transitables y false para no transitables, encontrar un camino transitable entre la entrada y la salida solo con movimientos en la misma fila o columna

4.3.-Vuelta atrás: Componentes de diseño.

Representación del problema:

- Se podría representar con un grafo cuyos nodos son las acciones posibles dentro de la matriz(arriba, abajo derecha e izquierda) que contendrían su posición dentro de la matriz.
- La solución sera una lista (Xi…Xn) tal que Xi es la primera acción que lleva al camino solución y Xn la última

Restricciones implícitas:

- Solo se permiten movimientos en posiciones adyacentes de la matriz en la misma fila o columna en la que nos encontramos
- Solo se permiten movimientos a casillas transitables(están marcadas como True en la matriz)

- Solo se permiten movimientos a posiciones que están dentro de los límites de la matriz

Restricciones explícitas:

- para evitar ciclos, debe evitarse todo lo posible visitar posiciones ya visitadas

Pseudocódigo:

```
Función buscarSalida(M[n-1[n-1], salida, fila, columna){
MAPA [fila][columna] = false
    pair<bool,list<pair<int,int> > >casilla;

  if fila an col son iguales a casilla objetivo {
     casilla.first = true
     casilla.second =meter_hijo_actual_en_lista
     return cas;
  }
  for  i=arriba hasta izquierda incrementar i {
     if arriba
        if casilla dentro de laberinto y transitable
          hijo=fila  -1
           z=buscarSalida(mapa,objetivo,child.first,col);
          if z=Solucion
            z.lista.meter_hijo_actual_en_lista
              return z;

      if derecha
        if casilla dentro de laberinto y transitable
           hijo=columna+1:
          z=buscarSalida(mapa,objetivo,fil,child.second);
           if z=Solucion
              z.lista.meter_hijo_actual_en_lista
```

```
                return z;
        if abajo
            if casilla dentro de laberinto y transitable
              hijo=fila+1:
               z=buscarSalida(mapa,objetivo,child.first,col);
               if z=Solucion
                  z.lista.meter_hijo_actual_en_lista
                  return z;
        if izquierda
            if casilla dentro de laberinto y transitable
              hijo=columna-1;
               z=buscarSalida(mapa,objetivo,fil,child.second);
              if z=Solucion
                  z.lista.meter_hijo_actual_en_lista
                  return z;


    return cas;
}
```

Este pseudocódigo consiste en crear una casilla que contenga una lista con las posiciones que las han llevado a ella y un valor de si ha encontrado la solución.

Devuelve la casilla con la solución encontrada y la lista de movimientos y en el bucle se hace todo el recorrido en grafo para cada acción (arriba, abajo, derecha, izquierda) hasta encontrar la casilla solucion y si no la encuentra devuelve la casilla a false y con una lista vacía. Se verá mejor en el siguiente ejemplo.

4.4.-Ejemplo paso a paso del funcionamiento.

True=transitable F=no transitable

Entrada T	T	T	F
T	F	T	F
F	T	T	T
F	T	T	**Salida**

Se empezaria comprobando si es la casilla solución (no lo es porque es la entrada) y se empezaría a llamar recursivamente arriba ,derecha,abajo e izquierda . En este caso arriba no se podría avanzar y la siguiente dirección sería derecha:

Entrada F	**T**	T	F
T	F	F	F
F	T	T	T
F	T	T	**Salida**

Se guardaría en la secuencia la posición (0,1), se pondría la casilla anterior a false y se llamaría recursivamente desde esta posición y ahora al volver no poder ir hacia arriba se volvería a a ir a la derecha:

Entrada F	**F**	**T**	F
T	F	T	F
F	T	T	T
F	T	T	**Salida**

Se guardaría en la secuencia la posición (0,2), se pondría la casilla anterior a false y se llamaría recursivamente desde esta posición y ahora al volver no poder ir hacia arriba ni derecha se dirigiría hacia abajo:

Entrada F	**F**	**F**	F
T	F	**T**	F
F	T	T	T
F	T	T	**Salida**

Se guardaría en la secuencia la posición (1,2), se pondría la casilla anterior a false y se llamaría recursivamente desde esta posición y ahora al volver no poder ir hacia arriba(ya está visitada por lo que está a false) ni derecha se dirigiría hacia abajo:

Entrada F	**F**	**F**	F
T	F	**F**	F
F	T	**T**	T
F	T	T	**Salida**

Se guardaría en la secuencia la posición (2,2), se pondría la casilla anterior a false y se llamaría recursivamente desde esta posición y ahora al volver no poder ir hacia arriba(ya está visitada por lo que está a false) se dirigiría hacia la derecha:

Entrada F	**F**	**F**	F
T	F	**F**	F
F	T	**F**	**T**
F	T	T	**Salida**

Se guardaría en la secuencia la posición (2,3), se pondría la casilla anterior a false y se llamaría recursivamente desde esta posición y ahora al volver no poder ir hacia arriba(ya está visitada por lo que está a false) ni derecha tiraria hacia abajo que es la casilla salida:

Entrada F	**F**	**F**	F
T	F	**F**	F
F	T	**F**	**F**
F	T	T	**Salida**

Se guardaría en la secuencia la posición (3,3), se pondría la casilla anterior a false y al ser solución se irían devolviendo las llamadas recursivas y metiendo las posiciones en la secuencia hasta llegar a la entrada y devolverá la función la casilla con solución=true y la siguiente secuencia: (0,0)--(0,1)--(0,2)--(1,2)--(2,2)--(2,3)--(3,3) que seria el órden de posiciones que llevará desde la entrada hasta la salida por un camino transitable.

Problema 5

El problema consiste en encontrar la salida de un laberinto. Más concretamente, supondremos que el laberinto se representa mediante una matriz cuadrada bidimensional de tamaño n X n. Cada posición almacena un valor booleano “true” si la casilla es transitable y “false” si la casilla no es transitable. Los movimientos permitidos son a casillas adyacentes de la misma fila o la misma columna. Podemos suponer que las casillas de entrada y salida del laberinto son (0,0) y (n -1, n -1) respectivamente. Por tanto, el problema consiste en, dada una matriz que representa el laberinto, encontrar si existe un camino para ir desde la entrada hasta la salida.

3. (Problema 5) Modificar el algoritmo para que encuentre el camino más corto. En este caso no pararemos cuando encontremos una solución, sino que se seguirá la exploración. No obstante, se puede (y debe) realizar una poda para no explorar soluciones parciales que ya tengan una longitud mayor que la mejor hallada hasta el momento.

5.1.-Diseño del Algoritmo Branch and Bound

El algoritmo branch and bound explora todas las posibles soluciones del problema, pero evita explorar ramas que no pueden conducir a una solución mejor que la mejor encontrada hasta el momento.

Restricciones Implícitas:

- Movimientos a Casillas Adyacentes:
  - Sólo se permiten movimientos a casillas adyacentes en la misma fila o columna.
  - Los movimientos posibles son: derecha, izquierda, arriba y abajo.
- Posiciones Transitables:
  - Solo se pueden mover a casillas que son transitables, es decir, que están marcadas como true en la matriz.
- Límites de la Matriz:
  - Los movimientos deben mantenerse dentro de los límites de la matriz. No se pueden realizar movimientos que resulten en posiciones fuera del rango de la matriz ($0 \leq x < n$ y $0 \leq y < n$).

Restricciones Explícitas:

- Evitación de Ciclos:
    - Se debe evitar visitar casillas que ya han sido visitadas para prevenir ciclos y redundancias en el camino.
- Podar Caminos Subóptimos:
    - Durante la exploración, si el costo del camino actual ya es mayor o igual al mejor camino encontrado hasta el momento, se debe podar ese camino para no explorar soluciones parciales que no puedan mejorar la mejor solución encontrada.

Inicialización:

- Crear una cola de prioridad para explorar los nodos del árbol de estados.
- Insertar la posición de inicio (0,0) en la cola con un costo inicial de 0.
- Crear una matriz para llevar registro de las posiciones visitadas.
- Inicializar una variable para la longitud del camino más corto como un valor muy grande (infinito).

Exploración:

- Mientras la cola no esté vacía:
    - Extraer el nodo con el menor costo (la posición actual y el costo del camino actual).
    - Si la posición actual es la posición de salida, actualizar la longitud del camino más corto si es menor que la longitud actual.
    - Si la longitud del camino actual ya es mayor que la mejor longitud encontrada, continuar con la siguiente iteración (poda).
    - Marcar la posición actual como visitada.
    - Explorar las posiciones adyacentes (arriba, abajo, izquierda, derecha) y añadirlas a la cola si son transitables y no han sido visitadas, actualizando el costo del camino.

Salida:

- Al final del proceso, la variable que lleva la cuenta de la longitud del camino más corto contendrá la longitud del camino más corto desde la entrada hasta la salida, si existe un camino.

Pseudocódigo

```
FUNCION branch_bound(matriz)

  n ← longitud(matriz)

  SI matriz[0][0] ES false O matriz[n-1][n-1] ES false ENTONCES
     RETORNAR -1  // Si la salida o la llegada no son válidos
  FIN SI

  movimientos ← [(0, 1), (0, -1), (1, 0), (-1, 0)]  // derecha, izquierda, abajo, arriba

  visitado ← matriz de tamaño n X n inicializada a false
  cola ← cola de prioridad con elemento (0, 0, 0)
```

```
   mejor_solucion ← infinito

   MIENTRAS cola NO esté vacía HACER // while
      (costo, x, y) ← extraemos el elemento con menor costo de cola

      SI x = n-1 Y y = n-1 ENTONCES
         mejor_solucion ← min(mejor_solucion, costo)//Actualizamos mejor_solucion si
costo es menor
         CONTINUAR
      FIN SI

      SI costo ≥ mejor_solucion ENTONCES
         CONTINUAR  // Poda
      FIN SI

      PARA CADA (dx, dy) EN movimientos HACER
         nx ← x + dx
         ny ← y + dy

         SI casilla_valida(matriz, visitado, nx, ny) ENTONCES
            visitado[nx][ny] ← true
            insertar (costo + 1, nx, ny) en cola
         FIN SI //Cerramos todos los bucles
      FIN PARA
   FIN MIENTRAS

   SI mejor_solucion = infinito ENTONCES
      RETORNAR -1  // No se encontró camino
   SINO
      RETORNAR mejor_solucion
   FIN SI
FIN FUNCION

FUNCION casilla_valida(matriz, visitado, x, y) //Comprobamos la casilla
   n ← longitud(matriz)
   RETORNAR 0 ≤ x < n Y 0 ≤ y < n Y matriz[x][y] Y NO visitado[x][y]
FIN FUNCION
```

5.4.-Ejemplo paso a paso del funcionamiento.

True=transitable F=no transitable

costo=0 x=0 y=0

Entrada T	F	T	T
T	T	F	T
F	T	T	T
F	F	T	**Salida T**

Primer paso
Actualizamos el estado de la entrada y visitado[0][0]=True

Exploramos movimientos: Arriba e izquierda están fuera de los límites y derecha no es transitable, luego solo nos queda ir hacia abajo

cola= [(1,1,0)] costo=1, x=1, y=0

Segundo paso

Extraemos 1,1,0 de la cola

Actualizamos visitado [1][0]= true

Entrada T	F	T	T
T	T	F	T
F	T	T	T
F	F	T	**Salida T**

Exploramos movimientos:

Solo es posible el movimiento a derecha por lo que agregamos (2,1,1) a la cola

cola= [(2,1,1)]

Tercer paso

Extraemos 2,1,1 de la cola

visitado[1][1]=True

Entrada T	F	T	T
T	T	F	T

F	T	T	T
F	F	T	**Salida T**

Exploramos movimientos

Solo podemos ir hacia abajo, (2,1) agregamos (3,2,1) a la cola

cola= [(3,2,1)]

Cuarto paso

Extraemos 3,2,1 de la cola

visitado[2][1]=True

Entrada T	F	T	T
T	T	F	T
F	T	T	T
F	F	T	**Salida T**

Exploramos movimientos

- Derecha: (2, 2) es transitable y no visitado, agregamos (4, 2, 2) a la cola.
- Izquierda: (2, 0) es no transitable.
- Abajo: (3, 1) es no transitable.
- Arriba: (1, 1) ya ha sido visitado.

cola= [(4,2,2)]

Quinto paso

Extraemos 4,2,2 de la cola

visitado[2][2]=True

Entrada T	F	T	T
T	T	F	T
F	T	T	T
F	F	T	**Salida T**

Exploramos movimientos

- Derecha: (2, 3) es transitable y no visitado, agregamos (5, 2, 3) a la cola.
- Izquierda: (2, 1) ya ha sido visitado.
- Abajo: (3, 2) es transitable y no visitado, agregamos (5, 3, 2) a la cola.
- Arriba: (1, 2) es no transitable

cola = [(5, 2, 3), (5, 3, 2)].

Sexto paso

Extraemos 5,2,3 de la cola

visitado[2][3]=True

Entrada T	F	T	T
T	T	F	T
F	T	T	T
F	F	T	**Salida T**

Exploramos movimientos

- Derecha: (2, 4) está fuera de los límites.
- Izquierda: (2, 2) ya ha sido visitado.
- Abajo: (3, 3) es transitable y no visitado, agregamos (6, 3, 3) a la cola.
- Arriba: (1, 3) es transitable y no visitado, agregamos (6, 1, 3) a la cola.

cola = [(5, 3, 2), (6, 3, 3), (6, 1, 3)].

Séptimo paso

Extraemos 5,3,2 de la cola

visitado[3][2]=True

Entrada T	F	T	T
T	T	F	T
F	T	T	T
F	F	T	**Salida T**

Exploramos movimientos

- Derecha: (3, 3) es transitable y no visitado, agregamos (6, 3, 3) a la cola.
- Izquierda: (3, 1) es no transitable.
- Abajo: (4, 2) está fuera de los límites.
- Arriba: (2, 2) ya ha sido visitado.

cola = [(6, 3, 3), (6, 1, 3),(6, 3, 3)].

Octavo paso

Extraemos 6,3,3 de la cola

Como hemos llegado a la meta, actualizamos la variable mejor_camino=6

Entrada T	F	T	T
T	T	F	T
F	T	T	T
F	F	T	**Salida T**

cola = [(6, 1, 3),(6, 3, 3)].

Noveno paso

Extraemos 6,1,3 de la cola

Cómo hemos llegado a la meta, con un costo de 6 que es mayor o igual al coste actual, hacemos poda y no continuamos explorando el camino.

Entrada T	F	T	T
T	T	F	T
F	T	T	T
F	F	T	**Salida T**

cola = [(6, 1, 3)].

Décimo paso

Extraemos 6,3,3 de la cola

Cómo hemos llegado a la meta, con un costo de 6 que es mayor o igual al coste actual, hacemos poda y no continuamos explorando el camino.

Entrada T	F	T	T
T	T	F	T
F	T	T	T
F	F	T	**Salida T**

-

cola = [].

Salida:

La cola está vacía

Mejor camino = 6

return mejor camino=6

Como compilar

ejercicio 4

-g++ ej4.1.cpp -o ej4.1

./ej4.1 4 //4 es el tamaño nXn de la matriz

generará la matriz 4x4 y la secuencia de posiciones desde la casilla (0,0) hasta la (3,3)