PRÁCTICA 4: ALGORITMOS DE EXPLORACIÓN DE GRAFOS

Integrantes del grupo:

Ana López Mohedano
Adrián Anguita Muñoz
Marina Jun Carranza Sánchez
Pedro Antonio Mayorgas Parejo
Rafael Jiménez Márquez

ÍNDICE

---

0. Aclaraciones previas

Para la resolución de problemas propuestos, se va a llevar a cabo el análisis de los siguientes algoritmos de exploración de grafos, cada cual se dividirá en dos apartados:

- **Diseño del algoritmo**: incluye … así como el pseudocódigo.
- **Ejemplo de uso**: una ejecución paso a paso de una instancia para explicar el funcionamiento del algoritmo.

Pautas para probar el código con makefile:

Se incluye una tarea “all” para compilar y probar todos los algoritmos, y también una orden de make de compilación y de ejecución para cada uno de ellos, por ejemplo: “make pX_compile” y “make pX_test”, siendo X el número del problema.

1. Problema de asignación de equipos

a) Diseño del algoritmo

El diseño es una adaptación de ***Backtracking***, un algoritmo de exploración de grafos que consiste en hacer una **búsqueda en profundidad** (Depth-First Search, DFS) de todas las soluciones posibles, para así encontrar la(s) mejor(es).

Este procedimiento requiere un gran uso de recursos de memoria en comparación con otros ya estudiados como Greedy, especialmente para tamaños de caso grandes, pero a diferencia de ellos, Backtracking se trata de un algoritmo **óptimo**.

Restricciones explícitas

- El número total de estudiantes (n) debe ser par para no dejar a nadie fuera.
- Cada estudiante debe emparejarse con exactamente otro estudiante.
- La matriz de niveles de preferencia (P) es de tamaño n×n.
- El objetivo principal es maximizar la suma total de los productos de las preferencias; es decir, maximizar $\sum P(i,j) \times P(j,i)$ para todas las parejas (i, j).

Restricciones implícitas

- El valor de P(i, i) se establecerá a -1 (inválido) ya que un estudiante no puede emparejarse consigo mismo.
- P(i, j) y P(j, i) pueden ser diferentes, por lo que el valor de emparejamiento considera los dos igual de importantes, pudiéndose calcular con el producto de ambos.
- No pueden repetirse parejas, una vez emparejado el estudiante, no puede aparecer en otro equipo.
- El proceso de formación de parejas debe continuar hasta que todos hayan sido emparejados y se hayan explorado todas las combinaciones válidas.

Pseudocódigo

```
Función P1_Backtracking(k, n, suma, parejas)
    Si k == n, hacer:
        Si suma > maxSum, hacer:
            maxSum = suma
            parejasÓptimas = parejas
        Fin-Si
        Retornar
    Si emparejado[k], hacer:
        P1_Backtracking(k + 1, n, suma, parejas)
    Fin-Si

    Para i desde k + 1 hasta n, hacer:
        Si !emparejado[i], hacer:
            emparejado[k] = emparejado[i] = True
```

```
            parejas.añadir({k,i})
            pairValue = P[k][i] * P[i][k]

            P1_Backtracking(k + 1, n, suma + pairValue, parejas)
            parejas.pop_back()
            emparejado[k] = emparejado[i] = False
        Fin-Si
    Fin-Para
Fin-Función
```

b) Ejemplo de uso

Se va a explicar una instancia del problema, usando la matriz de preferencias **P** siguiente:

	0	**1**	**2**	**3**
0	[-	3,	5,	7]
1	[2,	-	4,	6]
2	[9,	5,	-	10]
3	[1,	6,	3,	-]

Se supone que **emparejados** es un vector de booleanos de tamaño n inicializado a false, **maxSum** es el valor máximo de la suma de emparejamientos (inicialmente a 0) y se tiene un vector de pair llamado **parejasÓptimas** que almacenará la mejor solución hasta ahora.

La función toma **k** = 0 (índice del estudiante actual), **n** = 4 (número par de estudiantes), **suma** = 0 (suma de los valores de emparejamiento) y un vector de **parejas** vacío.

```
P1_Backtracking(0, n, 0, parejas);
```

Rama 1

- **k = 0**. Intenta emparejarlo con **i = 1**:
  - Marca k e i como emparejados y los añade al vector de parejas. Es decir: emparejados: {0, 1}; parejas: **{(0,1)}**
  - Calcula el valor de la pareja: P[0][1] * P[1][0] = 3 * 2 = 6
  - Llama recursivamente con k = 1, suma = **6**.
  - *Nueva rama 2**
- **k = 1**. Como ya está emparejado, incrementa k y llama recursivamente a la función.
- **k = 2**. Intenta emparejarlo con **i = 3**:
  - emparejados: {0, 1, 2, 3}; parejas: **{(0, 1), (2, 3)}**
  - Calcula P[2][3] * P[3][2] = 10 * 3 = 30.
  - Llama recursivamente con k = 3, suma = 6 + 30 = **36**.
  - Deshace el emparejamiento de (2, 3) para probar el resto de combinaciones que pueden formarse si ya se tiene (0, 1), pero no hay más.
- **k = 3**. Como k == (n-1), se trata del último estudiante, que como ya está emparejado, se puede concluir que todos estudiantes ya tienen pareja.

- suma = 36 > maxSum = 0, por lo que se actualiza maxSum = 36 y se guardan las parejas formadas.
- Ya se ha terminado de explorar la primera solución posible. Estado solución:
  - parejasÓptimas: **{(0, 1), (2, 3)}**; maxSum = **36**.

Rama 2: tras la inserción del primer emparejamiento y la exploración de combinaciones a partir de la misma, se procede a "deshacer" la pareja inicial (0, 1).

- Se vuelve a **k = 0**, y esta vez intenta emparejarlo con **i = 2**:
  - emparejados: {0, 2}; parejas: **{(0, 2)}**
  - Calcula P[0][2] * P[2][0] = 5 * 9 = 45.
  - Llama recursivamente con k = 1, suma = **45**.
  - *Nueva rama 3**
- **k = 1** se intenta emparejar con **i = 3** (porque i = 2 ya tiene pareja):
  - emparejados: {0, 1, 2, 3}; parejas: **{(0, 2), (1, 3)}**
  - Calcula P[1][3] * P[3][1] = 6 * 6 = 36.
  - Llama recursivamente con k = 2, suma = 45 + 36 = **81**.
  - Deshace la pareja (1, 3) pero no hay más combinaciones a probar.
- **k = 3**. Como k == (n-1), todos estudiantes están emparejados.
  - suma = 81 > maxSum = 36, entonces se actualiza maxSum = 81 y se guardan las parejas formadas.
  - Ya se ha terminado de explorar la segunda solución posible. Estado solución:
    - parejasÓptimas: **{(0, 2), (1, 3)}**; maxSum = **81**.

Rama 3: se deshace la pareja (0, 2).

- **k = 0** se intenta emparejar con **i = 3**:
  - emparejados: {0, 3}; parejas: **{(0, 3)}**
  - Calcula P[0][3] * P[3][0] = 7 * 1 = 7.
  - Llama recursivamente con k = 1, suma = **7**.
  - Deshace la pareja (0, 3) pero no hay más combinaciones a probar.
- **k = 1** se intenta emparejar con **i = 2**:
  - emparejados: {0, 1, 2, 3}; parejas: **{(0, 3), (1, 2)}**
  - Calcula P[1][2] * P[2][1] = 4 * 9 = 36.
  - Llama recursivamente con k = 2, suma = 7 + 36 = **43**.
  - Deshace la pareja (1, 2) pero no hay más combinaciones a probar.
- **k = 2** ya está emparejado, incrementa k y llama recursivamente a la función.
- **k = 3** == (n-1), por tanto todos los estudiantes están emparejados.
  - suma = 43 < maxSum = 81, no se actualiza maxSum ni las parejasÓptimas.
  - Ya se ha terminado de explorar la última solución posible. Estado solución:
    - parejasÓptimas: **{(0, 2), (1, 3)}**; maxSum = **81**.

Una vez se han terminado de explorar todas las combinaciones posibles de parejas, se imprime por pantalla la solución óptima:

```
Máxima suma de valores de emparejamiento: 81
Emparejamientos óptimos:
- Pareja 1: 0 y 2
- Pareja 2: 1 y 3
```

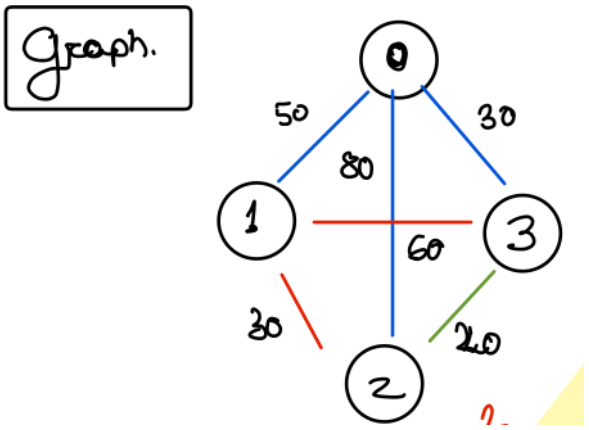
2. Problema de invitados de una cena de gala

a) Diseño del algoritmo y ejemplo

El diseño es una adaptación de ***Backtracking***, un algoritmo de exploración de grafos que consiste en hacer una **búsqueda en profundidad** (Depth-First Search, DFS) de todas las soluciones posibles, para así encontrar las mejores opciones de maximización de los invitados en una cena de gala.

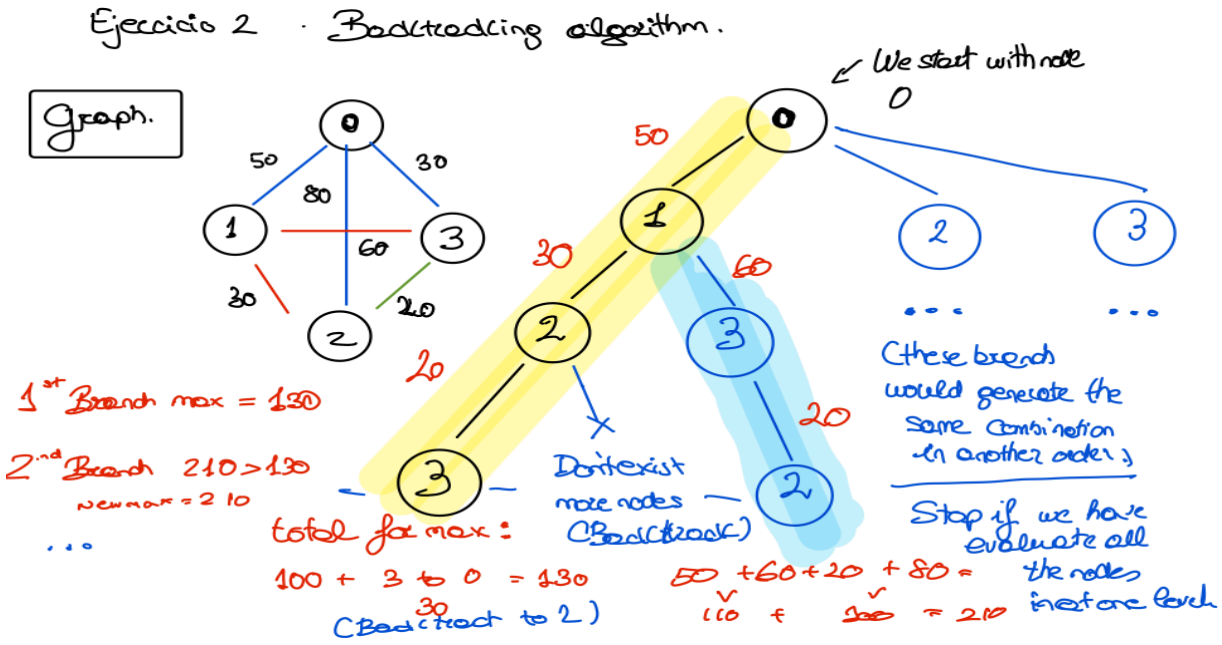
No es un algoritmo como tal óptimo, pero la solución que alcanza siempre será la mejor, ya que tiene que mirar todas las posibles combinaciones del grafo en forma de árbol. La profundidad del árbol depende de la cantidad de nodos que existan, es decir si existen 4 nodos, lo máximo que desciende la rama del árbol es 4, así como se tienen que generar 3 ramas partiendo de un nodo en la raíz del árbol.

El grafo del que se parte es como el que sigue:



Cada comensal tiene una adyacencia en la cual tiene un valor comprendido entre 0<=K<=100.

El árbol resultante es una combinación de cada uno de los nodos posibles, la parada del árbol es cuando llega al máximo nivel que existen en los nodos. Esto es que todos los nodos han sido visitados por las subramas.



Cuando se llega al máximo nivel se calcula la suma acumulada y se evalúa si esa suma es un máximo de todo el problema.

Sus ventajas son, que va a encontrar la mejor solución siempre independientemente del punto de partida. Ya que una de sus subramas de combinación dará al menos un máximo asegurado.

Sus desventajas son, que aunque tenga un máximo absoluto, tiene que resolver todo el problema ya que no tiene otros mecanismos de corte que aseguren la máxima solución.

Otro de las desventajas es que te va a dar de manera repetida el máximo, ya que son circuitos hamiltonianos, donde 0>1>3>2>0 es igual que 0>2>3>1>0.

Pseudocódigo

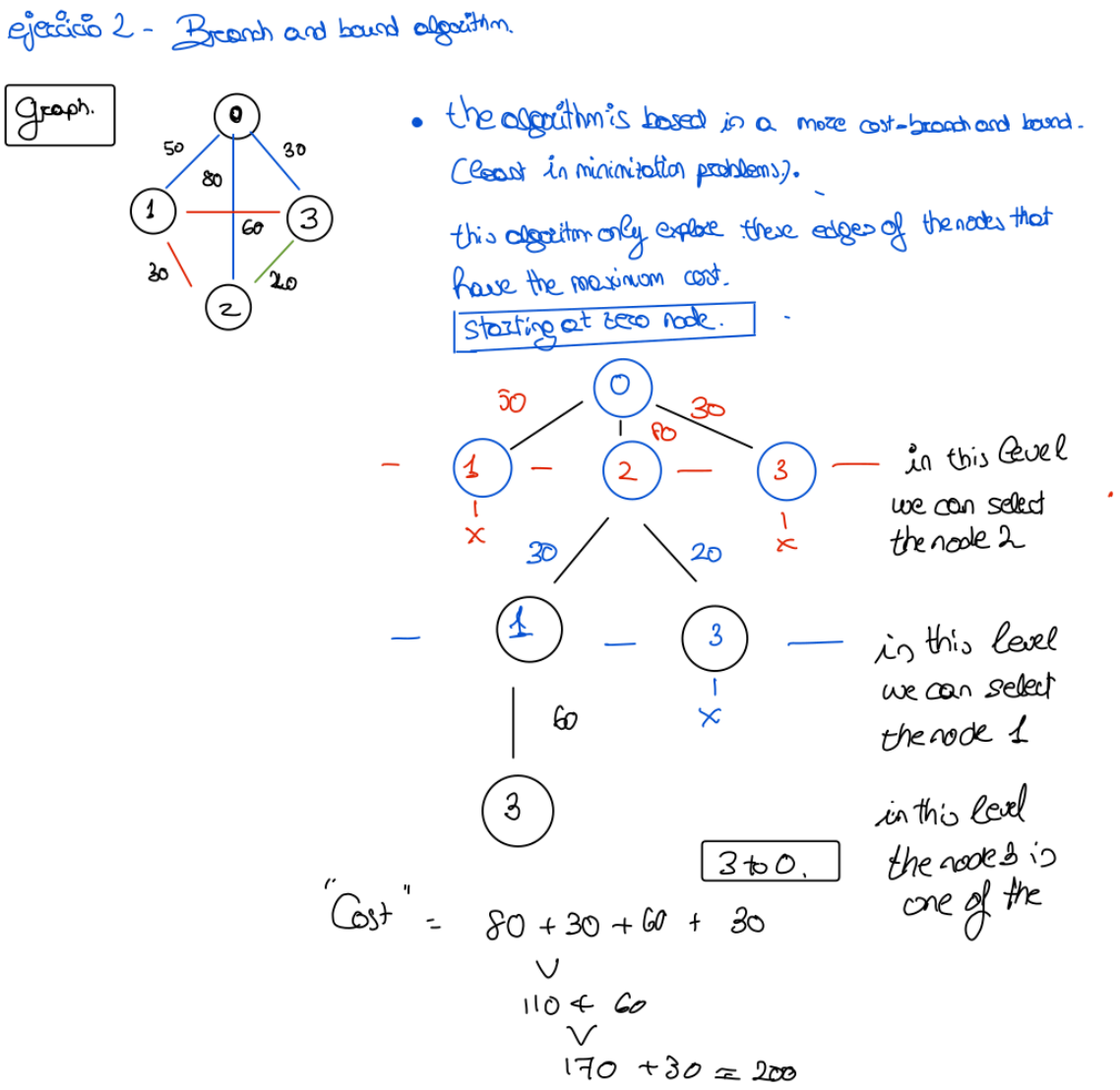
```
Function P2_Backtracking(C: array, CV: vector, currPos: int, nodes: int, level: int, current_sum: int,
max_sum: int, starting_node: int)
{
    // STOP if the level is the max
    if (level == nodes) then
        // SUM the starting node to complete the circuit and eval if is the maximal
        max_sum = max(max_sum, current_sum + C[currPos][starting_node]);
        return;
    endif

    // Generate the branches
    foreach (nodes as i) {
        if (CV[i] is false) {
            // Mark as visited
            CV[i] = true;
            // Recursivity, summing the level and the valuation
            backtrackComensales(C, CV, i, nodes, level + 1, current_sum + C[currPos][i], max_sum,
starting_node, branch);
            CV[i] = false;
        endif
    endfor
endfunction

// Find the maximum weight Hamiltonian Cycle
backtrackComensales(C, CV, starting_node, n, 1, 0, max_sum, starting_node);
```

Execution

```
./p2_comensales.bin
JUMP FROM 0 TO: 1 VALUE: 50 ACTUAL VALUE: 50
JUMP FROM 1 TO: 2 VALUE: 30 ACTUAL VALUE: 80
JUMP FROM 2 TO: 3 VALUE: 20 ACTUAL VALUE: 100
JUMP FROM 3 TO: 0 VALUE: 30 FINAL MAX: 130
BRANCH OF STARTING POINT OF:  0
0
1
2
3
0
JUMP FROM 1 TO: 3 VALUE: 60 ACTUAL VALUE: 110
JUMP FROM 3 TO: 2 VALUE: 20 ACTUAL VALUE: 130
JUMP FROM 2 TO: 0 VALUE: 80 FINAL MAX: 210
BRANCH OF STARTING POINT OF:  0
0
1
3
2
0
JUMP FROM 0 TO: 2 VALUE: 80 ACTUAL VALUE: 80
JUMP FROM 2 TO: 1 VALUE: 30 ACTUAL VALUE: 110
JUMP FROM 1 TO: 3 VALUE: 60 ACTUAL VALUE: 170
JUMP FROM 3 TO: 0 VALUE: 30 FINAL MAX: 200
BRANCH OF STARTING POINT OF:  0
0
2
1
3
0
JUMP FROM 2 TO: 3 VALUE: 20 ACTUAL VALUE: 100
JUMP FROM 3 TO: 1 VALUE: 60 ACTUAL VALUE: 160
JUMP FROM 1 TO: 0 VALUE: 50 FINAL MAX: 210
BRANCH OF STARTING POINT OF:  0
0
2
3
1
0
JUMP FROM 0 TO: 3 VALUE: 30 ACTUAL VALUE: 30
JUMP FROM 3 TO: 1 VALUE: 60 ACTUAL VALUE: 90
JUMP FROM 1 TO: 2 VALUE: 30 ACTUAL VALUE: 120
JUMP FROM 2 TO: 0 VALUE: 80 FINAL MAX: 200
BRANCH OF STARTING POINT OF:  0
0
3
1
2
0
JUMP FROM 3 TO: 2 VALUE: 20 ACTUAL VALUE: 50
JUMP FROM 2 TO: 1 VALUE: 30 ACTUAL VALUE: 80
JUMP FROM 1 TO: 0 VALUE: 50 FINAL MAX: 130
BRANCH OF STARTING POINT OF:  0
0
3
```

2.1 Problema de invitados de una cena de gala Branch And Bound *Extra*

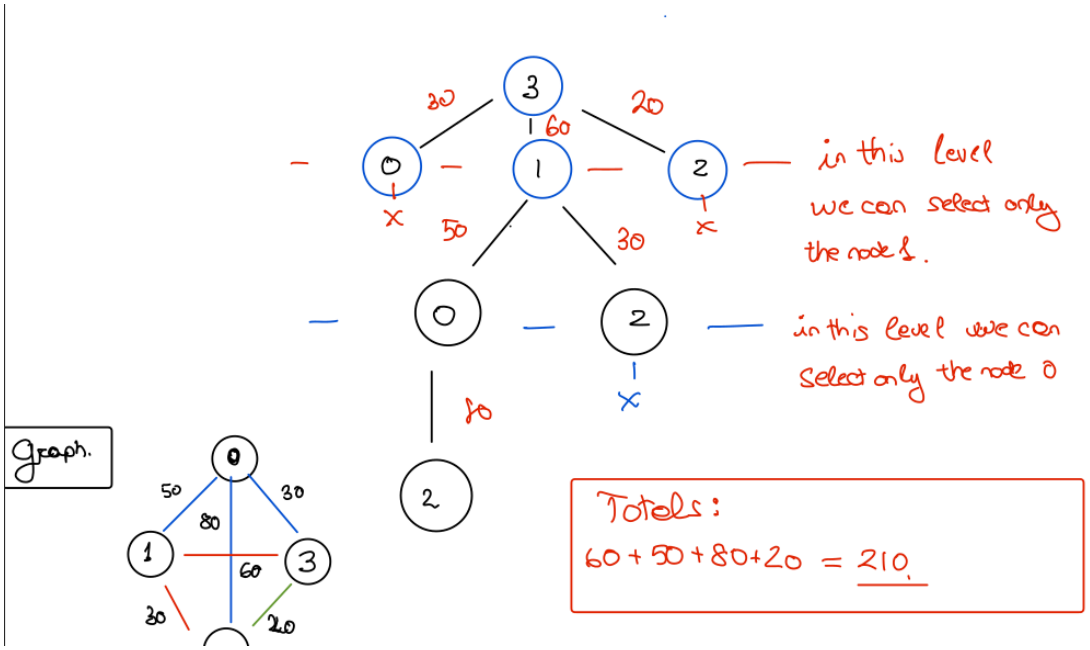
a) Diseño del algoritmo y ejemplo



El algoritmo tiene como criterio de corte (bound), la selección de evitar navegar por aquellas ramas que no maximicen el coste. Siempre descenderá un nivel a través del nodo que maximice.

Sin embargo este algoritmo está “limitado” desde el punto de vista del nodo de partida, como se puede ver en el ejemplo que hemos dado en comparación con el de backtracking nos podemos dar rápidamente cuenta de que no es la mejor solución, como se parte desde un nodo, la mejor solución desde ese nodo es esa.

Sin embargo si elegimos el nodo 3 obtenemos la mejor solución.



Con todo conociendo esto, todas las ramas generadas en branch and bound está contenida como uno de los subconjuntos de backtracking. Por lo que tenemos que:

Soluciones de **B&B** ⊆ Soluciones de **Backtracking.**

Ventajas: Obtenemos una solución muy rápida del problema vs backtracking.
Desventajas: Dependiendo de cómo se haga el algoritmo y del nodo de partida. La solución no es la máxima absoluta.

Pseudocódigo

```
function branchAndBoundMaximal(const std::vector<std::vector<int>> & C, std::vector<bool> & CV, int
cummulated, int actual_comensal, const int comensal_starting)

    max_actual = 0;
    new_comensal = -1;

    if (allVisited(CV))
        // SATISFIED: WE COMPLETE THE CIRCUIT
        max_actual = C[actual_comensal][comensal_starting];
        cummulated += max_actual;

        return cummulated;
    endif

    // SELECTING THE BEST NODE FOR GENERATING THE BRANCH
    foreach (CV as i){
        if (i != actual_comensal && CV[i]==false && C[actual_comensal][i] > max_actual){
            max_actual = C[actual_comensal][i];
            new_comensal = i;
        endif
    endfor

    // mark the visited commensal and set as the next to eval
    cummulated += max_actual;
    CV[new_comensal] = true;

    return branchAndBoundMaximal(C,CV,cummulated,new_comensal,comensal_starting);
}
```

Execution

```
executing Backtrack algorithm of p2
./p2_comensales_branchnbound.bin
STARTING AT: 0
FROM: 0 TO: 2 FOR ADYACENCY *MAX* : 80 CUMMULATED: 80
FROM: 2 TO: 1 FOR ADYACENCY *MAX* : 30 CUMMULATED: 110
FROM: 1 TO: 3 FOR ADYACENCY *MAX* : 60 CUMMULATED: 170
FROM: 3 TO: 0 FOR ADYACENCY *MAX* : 30 CUMMULATED: 200
200
STARTING AT: 1
FROM: 1 TO: 3 FOR ADYACENCY *MAX* : 60 CUMMULATED: 60
FROM: 3 TO: 0 FOR ADYACENCY *MAX* : 30 CUMMULATED: 90
FROM: 0 TO: 2 FOR ADYACENCY *MAX* : 80 CUMMULATED: 170
FROM: 2 TO: 1 FOR ADYACENCY *MAX* : 30 CUMMULATED: 200
200
STARTING AT: 2
FROM: 2 TO: 0 FOR ADYACENCY *MAX* : 80 CUMMULATED: 80
FROM: 0 TO: 1 FOR ADYACENCY *MAX* : 50 CUMMULATED: 130
FROM: 1 TO: 3 FOR ADYACENCY *MAX* : 60 CUMMULATED: 190
FROM: 3 TO: 2 FOR ADYACENCY *MAX* : 20 CUMMULATED: 210
210
STARTING AT: 3
FROM: 3 TO: 1 FOR ADYACENCY *MAX* : 60 CUMMULATED: 60
FROM: 1 TO: 0 FOR ADYACENCY *MAX* : 50 CUMMULATED: 110
FROM: 0 TO: 2 FOR ADYACENCY *MAX* : 80 CUMMULATED: 190
FROM: 2 TO: 3 FOR ADYACENCY *MAX* : 20 CUMMULATED: 210
210
```

3. Problema Senku

a) Diseño del algoritmo

El diseño sigue el modelo de ***Backtracking***, un algoritmo de exploración de grafos que consiste en hacer una **búsqueda en profundidad** (Depth-First Search, DFS) de todas las soluciones posibles, para así encontrar las soluciones posibles del senku.

No es siempre un algoritmo óptimo, pero en este caso lo que buscamos son las soluciones posibles y no una solución óptima. La profundidad del árbol como máximo será de 31, ya que para alcanzar la solución deben comerse 31 fichas (cada movimiento se come una ficha, por lo tanto serán 31 movimientos). La solución se alcanza cuando quede esa última ficha en el centro del tablero, así que no todos los caminos de 31 nodos serán válidos para ser solución, o habrá caminos con menos movimientos, ya que puede haber situaciones donde no se puedan mover fichas y quede en un punto muerto.

Representación

El tablero se representará como una matriz de enteros, donde 0 será una ficha, 1 una casilla vacía y 2 no evaluable (no se puede poner ficha). Tendremos una matriz 7x7. Para almacenar la solución, tenemos un vector movimientosSolucion, donde en cada posición se almacena un par de enteros que será la posición de la ficha movida, y un entero que indicará la dirección del movimiento (0: arriba, 1: derecha, 2: abajo, 3: izquierda).

Restricciones explícitas:

- ➢ Las posiciones sean válidas (dentro del tablero y en una casilla que no sea un 2) y que ese movimiento se pueda realizar (la casilla que quiere saltar contiene una ficha y a la que salta está desocupada, y entra dentro de los límites de la matriz).
- ➢ Las fichas solo pueden colocarse en las posiciones de la cuadrícula que no estén bloqueadas por otras fichas o por casillas no ocupables.

Restricciones implícitas:

- ➢ Una ficha solo puede moverse a una posición si hay una ficha en la posición intermedia que puede ser saltada y eliminada del tablero.
- ➢ Para que un movimiento sea válido, la posición final debe estar vacía, que no esté ocupada por otra ficha.

Pseudocódigo

- ➢ **posActual:** un par de enteros, que indicarán la fila y columna de la ficha que se debe mover para explorar ese nuevo estado.
- ➢ **movimientoSiguiente:** será el movimiento que hará esa ficha, que puede haber 4 posibilidades (0: hacia arriba, 1: hacia la derecha, 2: hacia abajo, 3: hacia la izquierda).
- ➢ **n:** número de nodos totales, que en este caso es 49, el tamaño de la matriz. Esto es porque se comprueba todas las posiciones de la matriz, teniendo cuidado de las

casillas no ocupables (representadas con un 2). En total hay 33 casillas ocupables, y 32 fichas.

- ➢ **tablero_actual:** matriz bidimensional donde almacenaremos el estado del tablero en ese momento, siendo 0 una posición ocupada, 1 vacía, y 2 no evaluable.
- ➢ **movimientos:** vector que almacenará los movimientos que llevamos hasta ahora.

```
P3_BackTracking(posActual, movSiguiente, n, tablero_actual, movimientos)
      Si es solucion
            movimientosSolucion = movimientos
            Devuelve
      Fin-Si
      Si ya hay 32 movimientos
            Devuelve
      Fin-Si

      tablero_actualizado = tablero_actual

      Si movSiguiente es un movimiento válido
            Meter movimiento en “movimientos”
            Actualizar tablero con ese movimiento
      Fin-Si

      Para todos las casillas del tablero (de 0 hasta n)
            nodo_actual = tablero_actual[fila][columna]
            posicionActual = (fila, columna)

            Si el nodo_actual tiene ficha
                  Almacenar posibles movimientos en “posiblesMovimientos”

                  Si posiblesMovimientos no está vacío
                        Para todos los movimientos de posiblesMovimientos
                              Llamada a P3_Backtracking

                              Sacar último elem. del vector “movimientos”
                        Fin-Para
                  Fin-Si
            Fin-Si
      Fin-Para
Fin-Función
```

Tendremos también funciones auxiliares, para actualizar el tablero, para evaluar si es solución el estado actual, para poder ver movimientos posibles en el estado actual, etc.

b) Ejemplo de uso

En un principio, tenemos un tablero, con el estado inicial.

{2, 2, 0, 0, 0, 2, 2},
{2, 2, 0, 0, 0, 2, 2},
{0, 0, 0, 0, 0, 0, 0},
{0, 0, 0, 1, 0, 0, 0},
{0, 0, 0, 0, 0, 0, 0},
{2, 2, 0, 0, 0, 2, 2},
{2, 2, 0, 0, 0, 2, 2}

Donde la casilla del medio siempre está desocupada. El algoritmo recorrerá fila a fila cada ficha que encuentre y evaluará si es posible hacer movimientos con la ficha que está explorando en alguna de las direcciones (arriba, derecha, abajo o izquierda). La primera ficha que encuentra que es posible mover, que en este caso debería ser la (3, 1) y hacia abajo, acaba ocupando la casilla del medio y comiéndose la ficha de justo abajo, pasando a ser este el estado:

{2, 2, 0, 0, 0, 2, 2},
{2, 2, 0, **1**, 0, 2, 2},
{0, 0, 0, 1, 0, 0, 0},
{0, 0, 0, **0**, 0, 0, 0},
{0, 0, 0, 0, 0, 0, 0},
{2, 2, 0, 0, 0, 2, 2},
{2, 2, 0, 0, 0, 2, 2}

Aquí empieza una rama de exploración. Ahora, a partir de este nuevo estado, vuelve a hacer lo mismo que antes, empieza a recorrer desde el principio la matriz yendo por filas. Encuentra la primera ficha donde se pueda hacer movimiento, lo hace y sigue por esa rama. Repite el proceso de la misma manera.

Si se encuentra en un punto muerto donde no puede mover ninguna ficha, va sacando movimientos de la pila hasta que haya un movimiento que no haya sido explorado antes, y sigue por esa rama hasta llegar a un punto muerto o llegar a solución. Así la primera combinación que se encuentre de solución es la que se devolverá. Esto se puede hacer gracias a la función hayMovimientosPosibles, que devolverá los movimientos que pueda hacer la ficha en la posición que se le pase, con el estado del tablero de ese momento. Esto facilita que no se pase a la siguiente ficha hasta que no se hayan comprobado todos los posibles movimientos de esa ficha, siendo estos todos sus nodos hijos. Cuando se hayan comprobado todos, se pasará a la siguiente ficha y se verán todas sus posibles combinaciones. Si se llega a una solución, se almacena en un vector movimientosSolucion y devuelve el control de la función.

La función esSolucion evaluará si el estado actual es solución, comprobando que hay una ficha en el centro del tablero y que las demás casillas no tienen fichas (la única ficha que debe haber es la del centro).

El estado solución debería ser el siguiente:

{2, 2, 1, 1, 1, 2, 2},
{2, 2, 1, 1, 1, 2, 2},
{1, 1, 1, 1, 1, 1, 1},
{1, 1, 1, 0, 1, 1, 1},
{1, 1, 1, 1, 1, 1, 1},
{2, 2, 1, 1, 1, 2, 2},
{2, 2, 1, 1, 1, 2, 2}

Este siempre se conseguirá con la cantidad exacta de 31 movimientos, ya que hay 32 fichas y tienen que comerse 31 fichas, necesitando un salto para comer cada ficha.

4. Problema del laberinto Backtracking

a) Diseño del algoritmo

Para el diseño de este algoritmo, usaremos al igual que en ejercicios anteriores una adaptación de Backtracking.

Representación

Para el diseño del algoritmo, necesitamos representar de forma adecuada tanto el laberinto como la solución. Para representar el laberinto, vamos a usar una matriz bidimensional de booleanos tal y como se propone en el guión. Por tanto, tendremos una matriz cuadrada de n filas y n columnas en las que cada posición almacene el valor true si la casilla es transitable o false en caso contrario, esto es, que haya un muro. La posición de inicio, es decir, la entrada del laberinto será la posición (0,0) de la matriz y la casilla de salida, la (n-1, n-1).

En el caso de la solución, necesitamos una estructura que almacene de alguna forma el camino seguido desde la entrada hasta la salida. Como una casilla se representa mediante el valor de la fila y la columna, utilizaremos un pair que contenga ambos valores. La solución será un vector de estos pair que nos indicarán las casillas recorridas.

Dentro de las restricciones descritas posteriormente, tendremos que representar otra matriz equivalente a la matriz laberinto (de booleanos) que almacene las casillas que han sido visitadas para no retroceder a la casilla anterior en caso de que podamos continuar en alguna dirección. Las casillas estarán todas a false y las pondremos a true cuando sean visitadas.

Restricciones explícitas:

- ➢ El laberinto consistirá en una matriz bidimensional de tamaño nxn.
- ➢ Cada casilla del laberinto(matriz) tendrá un valor booleano es decir será true en caso de que sea transitable es decir podamos movernos a esa casilla o false en caso de que no sea transitable es decir que no podamos movernos a esa casilla.
- ➢ La entrada del Laberinto será en el (0,0) es decir la casilla de arriba a la izquierda del laberinto(matriz) y la salida será en la casilla (n-1,n-1) es decir la esquina de abajo a la derecha del laberinto(matriz).
- ➢ Los movimientos que se permiten realizar en el laberinto serían a las casillas adyacentes es decir si las casillas de alrededor son transitables se podría mover tanto arriba como abajo e izquierda como derecha.

Restricciones implícitas:

- ➢ Tendríamos que evitar salirnos del Laberinto(matriz) es decir no podríamos realizar movimientos fuera del laberinto todos tendrían que ser dentro de este.
- ➢ No podríamos visitar dos veces la misma casilla es decir una vez que pasamos por una casilla no podríamos volver a visitarla para así evitar ciclos y que el algoritmo busque un camino más eficazmente.

- ➢ El objetivo es conectar la entrada con la salida es decir encontrar un camino que conecte la casilla (0,0) que sería la entrada del laberinto y la casilla (n-1,n-1) que sería la salida de este.
- ➢ Los movimientos que se realizan deben ser movimientos que se hagan hacia casillas que se puedan visitar es decir se podrán hacer movimientos hacia casillas que sean true es decir transitables y no se podrán hacer movimientos hacia casillas que sean false es decir no transitables.

Pseudocódigo

Además de la representación y las restricciones descritas anteriormente, vamos a añadir dos vectores con los distintos valores que pueden tomar x e y. A su vez también tenemos definido ADYACENTES el cual es el número de casillas adyacentes el nuestro caso es 4.

```
const int ADYACENTES = 4;
int posicionesx[] = {-1, 1, 0, 0}
int posicionesy[] = {0, 0, -1, 1}

Function P4_Backtracking(x, y, xfin, yfin, laberinto, solucion, visitados, camino)

    visitados[x][y] = true

    if(x == xfin y y == yfin){
        solucion.push_back({x,y})
        camino[x][y] = true
        return true
    Fin if

    Para i desde 0 hasta ADYACENTES hacer:

        sigX = x + posicionesx[i]
        sigY = y + posicionesy[i]

        if (puedeAvanzar(sigX, sigY, laberinto) y !visitado[sigX][sigY])
            Si(P4_Backtracking(sigX, sigY, xfin, yfin, laberinto, solucion, visitados, camino))
                solucion.push_back({x,y})
                camino[x][y] = true
                return true
            Fin if

        Fin if

    Fin for

return false
```

b) Ejemplo de uso

Consideraciones Iniciales:
Nosotros hemos elegido el siguiente orden de movimientos para ir comprobando.

```
// Direcciones posibles: arriba, abajo, izquierda, derecha
int posicionesx[] = {-1, 1, 0, 0};
int posicionesy[] = {0, 0, -1, 1};
```

Es decir nuestro algoritmo siempre intentará primero ir hacia arriba, luego hacia abajo, luego izquierda y por último derecha.

Nuestro ejemplo lo vamos a probar en un laberinto de 5x5 con la siguiente distribución:

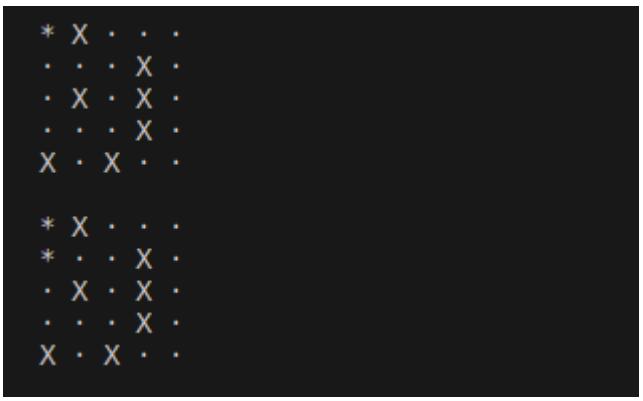
```
bool laberinto[FIL][COL] = {
    {true, false, true, true, true},
    {true, true, true, false, true},
    {true, false, true, false, true},
    {true, true, true, false, true},
    {false, true, false, true, true},
};
```

```
adrian@adrian-HP-Laptop-14s-dq1xxx:~/Algoritmica/Practica 4$ ./laberinto
La matriz ingresada es:
· X · · ·
· · · X ·
· X · X ·
· · · X ·
X · X · ·
```

Donde podemos ver que los “puntos” representan el camino por el que podemos avanzar y la “X” son los obstáculos los cuales no podemos pasar.

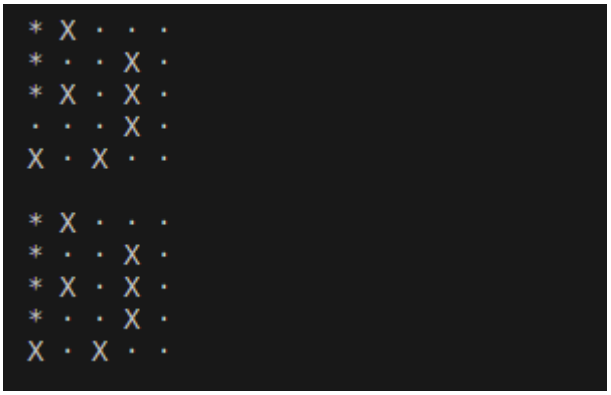
Empezaremos nuestro algoritmo como hemos comentado en las restricciones en la casilla (0,0).

Siguiendo el orden de direcciones que hemos establecido anteriormente marcamos como visitada la casilla (0,0) y exploramos las casillas adyacentes en este caso sería la (1,0) y la (0,1). Viendo esto comprobamos si podemos avanzar y no hemos visitado la casilla (1,0) como no es el caso avanzamos a esta y la marcamos como visitada en el caso de la casilla (1,0) como no tiene la posibilidad de avanzar la descartamos ya que es un muro.



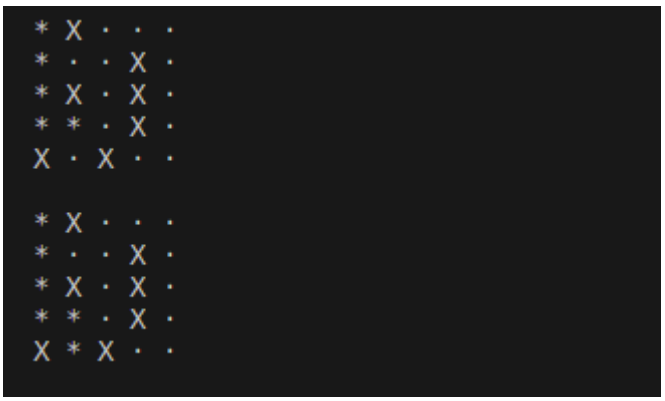
Una vez explorada esa casilla exploramos las casillas adyacentes en este caso serían la (0,0), la (2,0) y la (1,1) como la (0,0) ya la hemos visitado no la exploramos y continuamos con la (2,0) como no la hemos visitado y tiene la posibilidad de avanzar la exploramos y la marcamos como visitada.

A continuación exploramos las casillas adyacentes a este las cuales serían (1,0) y (3,0) ya que la (2,1) presenta un muro y no podemos avanzar a la misma. La casilla (1,0) sucedería como la (0,0) anteriormente es decir ya la hemos explorado y visitado por lo que escogemos la casilla (3,0) la cual exploramos y marcamos como visitada.



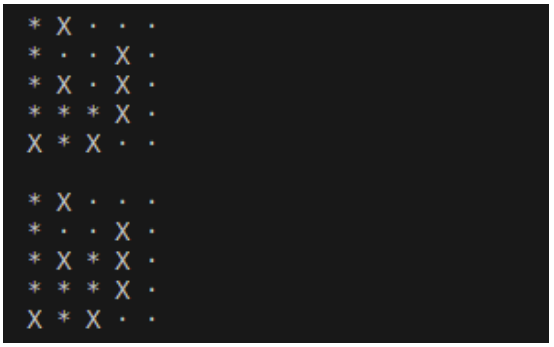
Estando en la casilla (3,0) volvemos a repetir el proceso exploramos las casillas adyacentes en este caso la casilla (4,0) presentaría un muro por lo que no podemos avanzar así que la descartamos la casilla (2,0) pasaria como anteriormente es decir ya la hemos explorado, por lo que exploramos la casilla (3,1) la cual podemos avanzar y no hemos visitado por lo que la marcamos como visitada.

Estando en la casilla (3,1) exploramos las casillas adyacentes y vemos que hacia arriba no podemos ir dado que hay un muro pero sí podemos avanzar a la (4,1) y a la (3,2) pero siguiendo el orden que hemos establecido anteriormente exploramos la casilla (4,1) por lo que avanzamos y la marcamos como visitada.



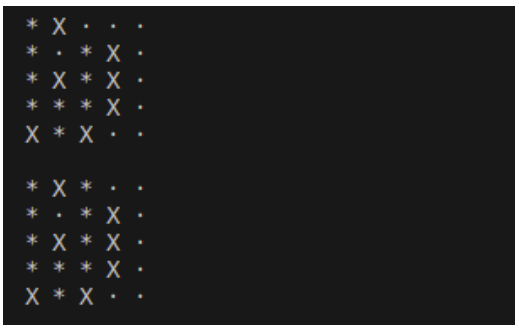
Actualmente estamos en la casilla (4,1) y al decidir explorar los adyacentes nos damos cuenta que no podemos avanzar hacia ninguna dirección ya que hacia abajo nos salimos del laberinto y hacia los lados existen muros por lo que comprobamos si esta última casilla es la casilla destino es decir el final del laberinto, al comprobarlo vemos que es falso es decir no es la casilla destino que en este caso seria la casilla (4,4) por lo que tenemos que realizar un **Backtracking** es decir tenemos que retroceder un paso dado que este camino no es solución para resolver el laberinto y no podemos avanzar por lo que esta casilla no la añadimos al camino solucion asi que ahora exploramos la otra casilla que teníamos disponible anteriormente que era la casilla (3,2) la cual podemos avanzar y no hemos visitado por lo que avanzamos y la marcamos como visitada.

Una vez estamos en la casilla (3,2) volvemos a hacer uso de las direcciones declaradas anteriormente en este caso cogeriamos la casilla de arriba siendo esta la (2,2) ya que la casilla de la derecha y la de abajo no las podemos explorar ya que poseen un muro como no la hemos explorado lo que hacemos es movernos hacia ella y la marcamos como visitada.



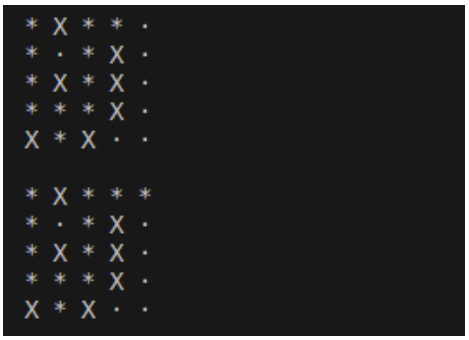
Continuamos explorando las casillas la siguiente seria la (1,2) ya que podemos seguir explorando hacia arriba como hemos declarado que será la primera dirección que comprobaremos este nodo tampoco estaba explorado por lo que avanzamos y lo marcamos como visitado.

A continuación seguimos explorando y el siguiente nodo que seleccionamos es el (0,2) el cual tampoco estaba explorado por lo que avanzamos y lo marcamos como visitado.

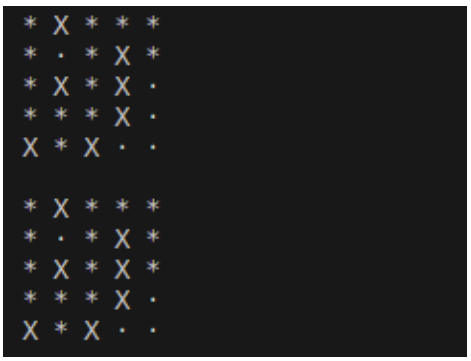


Estando en el nodo (0,2) continuamos con la exploración ahora como no podemos ir hacia arriba dado que nos salimos del laberinto y hacia abajo ya los hemos explorado la opción que nos queda es ir hacia la derecha porque hacia la izquierda hay un muro por lo que vamos a explorar el nodo adyacente (0,3) el cual no habíamos explorado por lo que avanzamos y lo marcamos como visitado.

En este nodo pasa lo mismo hacia arriba nos salimos del laberinto y hacia abajo hay un muro por lo que solo podemos ir hacia la izquierda por que la derecha ya la hemos explorado así que exploramos el nodo (0,4) el cual no habíamos explorado asi que lo marcamos como visitado.



A partir del nodo (0,4) continuamos con la exploración de los nodos adyacentes hacia arriba sucede lo mismo que en casos anteriores nos salimos del laberinto por lo que no podemos explorar hacia arriba así que podemos seguir explorando hacia abajo el siguiente nodo sería el (1,4) el cual no hemos explorado así que avanzamos y lo marcamos como visitado.
Desde aquí sucede igual que en el caso anterior así que exploramos el siguiente nodo el cual seria el (2,4) así que avanzamos y como no lo habíamos explorado y lo marcamos como visitado.



El siguiente nodo sucede lo mismo por lo que exploramos hacia el siguiente nodo el cual sería el que está situado debajo el cual sería el (3,4) como no lo habíamos explorado avanzamos y lo marcamos como visitado.

Y por último para este último nodo sucede lo mismo por lo que exploramos el nodo de abajo el cual seria el (4,4) y como en cada iteración vamos comprobando si es el nodo solución al explorar y marcar como visitado este nodo comprobamos si es el nodo solución es decir si es la salida del laberinto como este nodo es la salida del laberinto ya estaríamos en el destino y habríamos encontrado un camino el cual nos lleva desde la entrada la cual es el (0,0) y la salida del laberinto que es el (4,4) y el camino encontrado seria la solucion.

```
El camino para salir del laberinto es:
(0,0) (1,0) (2,0) (3,0) (3,1) (3,2) (2,2) (1,2) (0,2) (0,3) (0,4) (1,4) (2,4) (3,4) (4,4)

* X * * *
* · * X *
* X * X *
* * * X *
X · X · *
```

El camino solución que hemos encontrado el cual nos lleva desde la entrada a la salida del laberinto sería:

(0,0) (1,0) (2,0) (3,0) (3,1) (3,2) (2,2) (1,2) (0,2) (0,3) (0,4) (1,4) (2,4) (3,4) (4,4).

5. Problema del laberinto B&B

a) Diseño del algoritmo

Para realizar el algoritmo Branch & Bound nos vamos a basar en el ejercicio anterior ya que el enunciado del problema es el mismo. La diferencia es que con este algoritmo vamos a obtener el camino óptimo para salir del laberinto.

Representación

En cuanto a la representación, vamos a mantener la matriz bidimensional de booleanos para representar el laberinto al igual que las posiciones de entrada (0,0) y de salida (n-1, n-1). También mantenemos la representación de la solución (vector de pair con fila y columna de cada casilla) .

El primer cambio que vamos a realizar es que como este algoritmo trabaja con nodos, vamos a crear un struct que represente un nodo. Tiene como atributos tres enteros: x que representa el valor de la fila, y que representa el valor de la columna y costo, que almacena el costo de ese nodo. Más adelante detallaremos cómo se calcula. Además, el struct almacena un vector de pair llamado camino que tiene los valores de las casillas que forman parte del camino hasta ese nodo.

Otro de los cambios realizados es cambiar el tipo de la matriz que almacena las casillas que han sido visitadas, cambiándolo a int ya que almacenarán el costo de las casillas del laberinto. Las casillas intransitables tendrán valor infinito y a las además se le asignará el coste aunque inicialmente tendrán como valor infinito. El algoritmo tendrá como estructura para seleccionar el siguiente nodo una cola con prioridad, marcando la prioridad el coste del nodo.

Restricciones explícitas:

Las restricciones explícitas se mantienen como en el ejercicio anterior.

Restricciones implícitas:

Al igual que con las restricciones explícitas, las restricciones implícitas se mantienen como en el ejercicio anterior aunque podemos añadir alguna como:

- ➢ El objetivo es también conectar la entrada con la salida, pero en este caso, el camino debe ser el óptimo.
- ➢ El coste de las casillas debe ser positivo.

Pseudocódigo

```
Function P5_Backtracking(x, y, xfin, yfin, laberinto, solucion)
    visitados(FIL, vector<int>(COL, INT_MAX))
    priority_queue<Nodo> = pq

    caminoInicio = {{x, y}}
    costoInicial = calcularHeuristica(x, y, xfin, yfin)
    pq.push(Nodo(x, y, costoInicial, caminoInicio))

    visitados[x][y] = costoInicial

    Mientras priority_queue no este vacia hacer
        nodoActual = pq.top()
        pq.pop()

        xActual = nodoActual.x
        yActual = nodoActual.y
        nivelActual = nodoActual.costo
        caminoActual = nodoActual.camino


        Si (xActual == xfin y yActual == yfin) entonces
            solucion = caminoActual
            return True

        Para i desde 0 hasta ADYACENTES hacer
            sigX = xActual + posiciones[i]
            sigY = yActual + posicionesy[i]

            Si (puedeAvanzar(sigX, sigY, laberinto)) entonces

                nuevoCosto = caminoActual.size() + 1
                heuristica = calcularHeuristica(sigX, sigY, xfin, yfin)
                costoTotal = nuevoCosto + heuristica

                Si (nuevoCosto < visitados[sigX][sigY]) entonces
                    visitados[sigX][sigY] = nuevoCosto

                    nuevoCamino = caminoActual
                    nuevoCamino.push_back({sigX,sigY})
                    pq.push(Nodo(sigX, sigY, costoTotal, nuevoCamino))

                Fin Si

            Fin Si

        Fin Para

    Fin Mientras

    return False
```

b) Ejemplo de uso

El algoritmo empieza con el nodo (0,0). En el while, se coge el tope de la cola que en este caso es el (0,0). Al igual que en Backtracking, se comprueba si ese nodo es la salida para terminar la función. En caso contrario, hacemos un bucle para comprobar las posiciones adyacentes a la posición actual que son transitrables y calculamos su coste mediante la función calcular heurística que hace lo siguiente: abs(x - xfin) + abs(y - yfin); siendo x e y las posiciones de la casilla a explorar (adyacente) y xfin e yfin las posiciones de la casilla de salida del laberinto. En el último if se entra si el coste es menor que el que tiene la casilla de visitados (inicialmente a infinito) por lo que entrará y añadirá el nodo a la cola con prioridad solo si ese nodo no se ha visitado antes.

Por tanto, nuestra función de poda hará poda cuando los hijos de un nodo hayan sido visitados por otro nodo ya que tendrán un valor de coste mayor o igual que la casilla de visitados y, por tanto, no entrará en el if, no pudiéndose añadir a la cola de nodos estos nodos hijos. De esta manera, conseguimos podar ese camino y la función cogerá en la siguiente iteración el tope de la cola con prioridad, continuando la función su ejecución.

Vamos a ver todo el proceso de la ejecución del algoritmo para el mismo laberinto del ejercicio 4, iteración por iteración:

```
Ejercicio 5: Laberinto Branch & Bound

* X . . .
. . . X .
. X . X .
. . . X .
X . X . .

explorando la casilla: (0,0)
casilla a explorar: (1,0)


****************** Siguiente iteración******************
* X . . .
* . . X .
. X . X .
. . . X .
X . X . .

explorando la casilla: (1,0)
casilla a explorar: (2,0)
casilla a explorar: (1,1)


****************** Siguiente iteración******************
* X . . .
* . . X .
* X . X .
. . . X .
X . X . .

explorando la casilla: (2,0)
casilla a explorar: (3,0)
```

Podemos ver en color azul las casillas adyacentes que se pueden explorar y el camino que se va siguiendo en las distintas iteraciones. Las siguientes iteraciones siguen de la misma manera:

```
******************** Siguiente iteración********************
* X . . .
* * . X .
. X . X .
. . . X .
X . X . .

explorando la casilla: (1,1)
casilla a explorar: (1,2)


******************** Siguiente iteración********************
* X . . .
* . . X .
* X . X .
* . . X .
X . X . .

explorando la casilla: (3,0)
casilla a explorar: (3,1)


******************** Siguiente iteración********************
* X . . .
* * * X .
. X . X .
. . . X .
X . X . .

explorando la casilla: (1,2)
casilla a explorar: (0,2)
casilla a explorar: (2,2)
```

```
******************** Siguiente iteración********************
* X . . .
* . . X .
* X . X .
* * . X .
X . X . .

explorando la casilla: (3,1)
casilla a explorar: (4,1)
casilla a explorar: (3,2)


******************** Siguiente iteración********************
* X . . .
* * * X .
. X * X .
. . . X .
X . X . .

explorando la casilla: (2,2)


******************** Siguiente iteración********************
* X . . .
* . . X .
* X . X .
* * . X .
X * X . .

explorando la casilla: (4,1)


******************** Siguiente iteración********************
* X . . .
* . . X .
* X . X .
* * * X .
X . X . .

explorando la casilla: (3,2)
```

En estas últimas iteraciones hemos marcado en rojo aquellas casillas adyacentes que podrían ser visitadas pero no se añaden a la cola ya que otros nodos las han visitado anteriormente, por tanto se poda ese camino. También hemos marcado en verde la casilla (4,1) que es un nodo hoja. El algoritmo sigue así:

```
******************** Siguiente iteración********************
* X * • .
* * * X .
· X · X .
· · · X .
X · X · ·

explorando la casilla: (0,2)
casilla a explorar: (0,3)


******************** Siguiente iteración********************
* X * * •
* * * X .
· X · X .
· · · X .
X · X · ·

explorando la casilla: (0,3)
casilla a explorar: (0,4)


******************** Siguiente iteración********************
* X * * *
* * * X •
· X · X .
· · · X .
X · X · ·

explorando la casilla: (0,4)
casilla a explorar: (1,4)


******************** Siguiente iteración********************
* X * * *
* * * X *
· X · X •
· · · X .
X · X · ·

explorando la casilla: (1,4)
casilla a explorar: (2,4)
```

Podemos ver que el algoritmo ha podado el camino que estaba siguiendo y ha continuado por la casilla (0,2) que estaba almacenada en la cola pero que por la prioridad que marca el coste del nodo no se había visitado todavía. Continuamos hasta el final:

```
****************** Siguiente iteración******************
* X * * *
* * * X *
· X · X *
· · · X •
X · X · ·

explorando la casilla: (2,4)
casilla a explorar: (3,4)


****************** Siguiente iteración******************
* X * * *
* * * X *
· X · X *
· · · X *
X · X · •

explorando la casilla: (3,4)
casilla a explorar: (4,4)


****************** Siguiente iteración******************
* X * * *
* * * X *
· X · X *
· · · X *
X · X · *

explorando la casilla: (4,4)
```

Una vez explorada la casilla (4,4) que coincide con la casilla de salida, el algoritmo muestra el camino seguido y el dibujo final del laberinto con el camino seguido:

```
El camino para salir del laberinto es:
(0,0) (1,0) (1,1) (1,2) (0,2) (0,3) (0,4) (1,4) (2,4) (3,4) (4,4)

* X * * *
* * * X *
· X · X *
· · · X *
X · X · *
```

Nota: Todo el proceso del algoritmo mostrado paso a paso como se ha hecho no se encuentra en el código entregado ya que contiene un número importante de cout y hace que el código no sea tan legible pero como se ha podido comprobar en este ejemplo de uso, se puede comprobar que el algoritmo obtiene una solución óptima tal y como se pide.