Miguel Martínez Azor

Ángel Rodríguez Faya

Alejandro Botaro Crespo

Alberto Parejo Bellido

Alejandro Ocaña Sánchez

Algoritmos de Exploración de Grafos

Backtracking/Branch&Bound

Problemas

- FORMAR PAREJAS
- CENA
- SOLITARIO CHINO (SENKU)
- Laberinto Backtracking
- Laberinto B&B

Diseño de Algoritmo

Problema 1 Formar parejas

Enunciado:

- Existen n estudiantes en una clase y queremos formar equipos de parejas
- 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.
- El objetivo es encontrar el emparejamiento de cada estudiante con otro de manera que se maximice la solución

0	6	2	4
6	0	6	2
2	6	0	6
4	2	6	0

Problema 1 Formar parejas

Diseño de Algoritmo

Representación del Problema:

- 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.

- La solución estará formada por un vector (tupla) parejas de n estudiantes/posiciones donde cada i corresponde al emparejamiento del estudiante i con Xi

Parejas[n] → | 2 | 1 | 4 | 3 | → | 1 , 2 | 3 , 4 |

i =	1	2	3	4
Parejas[n]	2	1	4	3

Diseño de Algoritmo

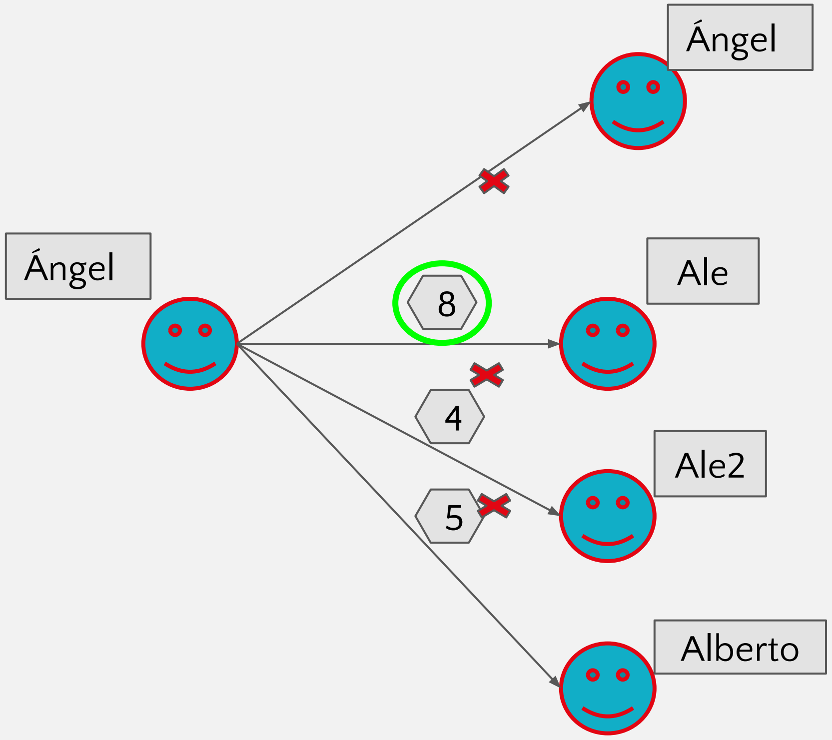
Problema 1 Formar parejas

Restricciones Implícitas:

- Maximización de 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



Diseño de Algoritmo

Problema 1 Formar parejas

Pseudocódigo:

```
Function Bestudiantes(M[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
```

Diseño de Algoritmo

Problema 1 Formar parejas

Ejemplo:

Nº estudiantes (n) = 4

Matriz de preferencias p(i,j)= $\begin{pmatrix} 0 & 6 & 2 & 4 \\ 6 & 0 & 6 & 2 \\ 2 & 6 & 0 & 6 \\ 4 & 2 & 6 & 0 \end{pmatrix}$

parejas[n] = {0}

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

suma total = 0

mejor suma = 0

1º- Escogemos el primer estudiante 1 y probamos a emparejarlo con el resto 2, 3, 4

2º-Pareja con el 1 → no cumple con las restricciones

3º-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 =(3, 4) Repetimos el proceso con estos

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

4º-Pareja con el 3 →

parejas = 3, 4, 1, 2

p(1,3) x p(3,1) = 2*2 = 4   Suma total = 4+4 =8

p(2,4) x p(4,2) = 2*2 = 4   Mejor suma = 72

Vuelta atrás y continuamos con el estudiante 4

5º-Pareja con el 4 →

parejas = 4, 3, 2, 1

p(1,4) x p(4,1) = 4*4 = 16   Suma total = 16+36 =52

p(2,3) x p(3,2) = 6*6 = 36   Mejor suma = 72

RESULTADO parejas ={ 2,1,4,3} -> (1,2) (3,4) conveniencia =72

Problema 2
Cena

Diseño del Algoritmo

Enunciado:

- Existen n invitados a una cena y queremos sentarlos a todos en una mesa circular.
- 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.
- El objetivo es encontrar es encontrar la máxima conveniencia de cada comensal con los invitados que se sientan a su lado de manera global

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

Problema 2
Cena

Diseño del Algoritmo

Representación del Problema:

- 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 solución estará formada por un vector que incluya la posición de cada uno de los comensales

Problema 2
Cena

Diseño del Algoritmo

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.

Diseño del Algoritmo

Problema 2 Cena

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
```

Problema 2
Cena

Ejemplo

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

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

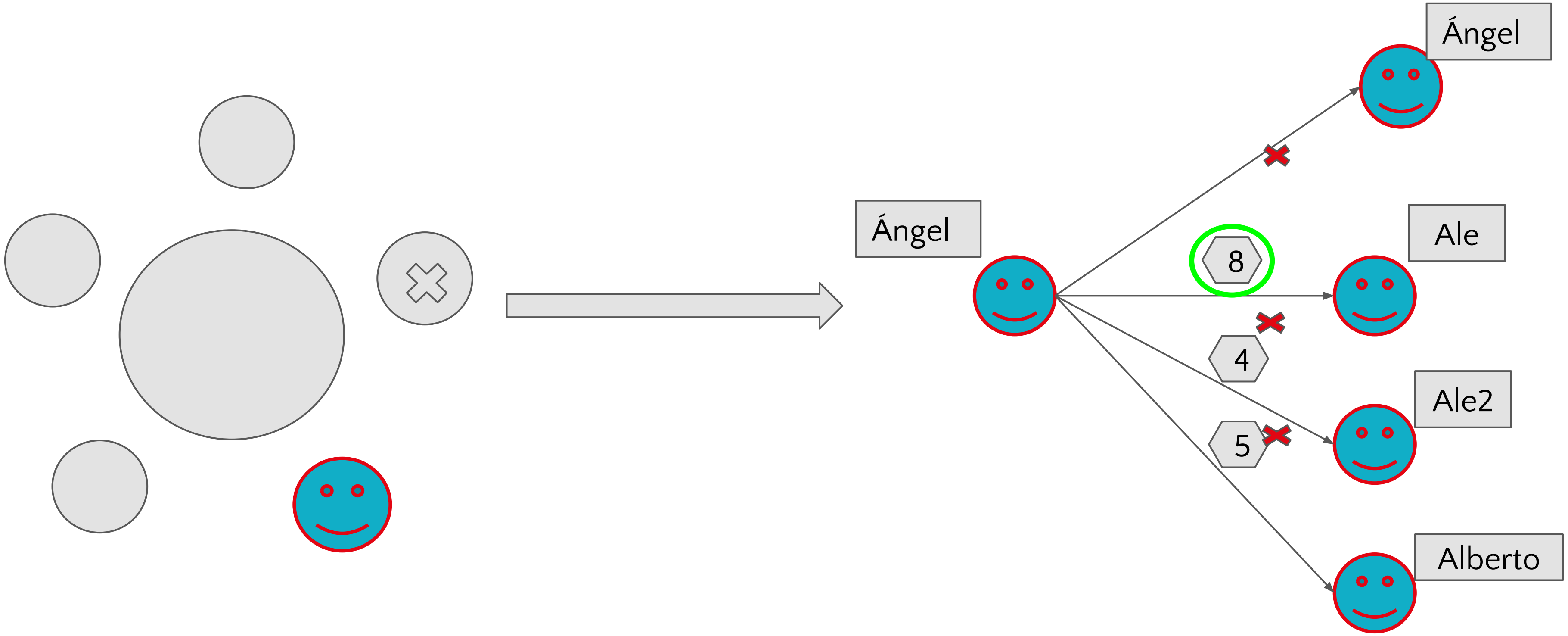
9.- Como todavía quedan invitados por sentar, escogemos el siguiente candidato para sentar, que en este caso el único que queda es el 5.

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 2
Cena

Ejemplo gráfico



PROBLEMA3 SOLITARIO CHINO (SENKU)

Presentación

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

PROBLEMA3 SOLITARIO CHINO (SENKU)

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):

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 cuales 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.

PROBLEMA3 SOLITARIO CHINO (SENKU)

Representación:

- **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).**

3.3.-Vuelta atrás: Componentes de diseño.

PROBLEMA3 SOLITARIO CHINO (SENKU)

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
```

3.3.-Vuelta atrás: Componentes de diseño.

PROBLEMA3 SOLITARIO CHINO (SENKU)

Código en C++:

```
bool resolverSolitarioChino() {
    if (fin()) {
        return true;
    }

    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            if (tablero[i][j] == ocupada) {
                int movimientos[4][2] = {{0, -2}, {-2, 0}, {0, 2}, {2, 0}};
                for (int k = 0; k < 4; k++) {
                    int ni = i + movimientos[k][0]; // cordenada x de la casilla destino
                    int nj = j + movimientos[k][1]; // cordenada y de la casilla destino
                    if (esMovimientoValido(i, j, ni, nj)) {
                        hacerMovimiento(i, j, ni, nj);
                        if (resolverSolitarioChino()) {
                            contador++;
                            cout << "Movimiento nº"<< contador <<" : (" << i << ", " << j
                                << ") a (" << ni << ", " << nj << ")" << endl;
                            return true;
                        }
                        deshacerMovimiento(i, j, ni, nj);
                    }
                }
            }
        }
    }
    return false;
```

3.3.-Vuelta atrás: Componentes de diseño.

PROBLEMA3 SOLITARIO CHINO (SENKU)

Código en C++:

```cpp
void imprimirTablero() {
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            switch (tablero[i][j]) {
                case libre:
                    cout << " ";
                    break;
                case ocupada:
                    cout << "o";
                    break;
                case prohibida:
                    cout << "x";
                    break;
            }
            cout << " ";
        }
        cout << endl;
    }
    cout << endl;
}

bool fin(){
    int cont = 0, posx = 0, posy = 0;
    for(int i = 0; i < n; i++){
        for(int j = 0; j < n; j++){
            if(tablero[i][j] == ocupada){
                cont++;
                posx = i;
                posy = j;
            }
        }
    }

    // Sólo hay una posición ocupada y es la del centro
    return (cont == 1 && posx == 3 && posy == 3);
}
```

3.3.-Vuelta atrás: Componentes de diseño.

PROBLEMA3 SOLITARIO CHINO (SENKU)

Código en C++:

```cpp
bool esMovimientoValido(int i, int j, int ni, int nj) {
    int mid_i = (i + ni) / 2;
    int mid_j = (j + nj) / 2;
    return (ni >= 0 && ni < n && nj >= 0 && nj < n && tablero[ni][nj] == libre
            && mid_i >= 0 && mid_i < n && mid_j >= 0 && mid_j < n
            && tablero[mid_i][mid_j] == ocupada);
}

void hacerMovimiento(int i, int j, int ni, int nj) {
    tablero[i][j] = libre;
    tablero[(i + ni) / 2][(j + nj) / 2] = libre;
    tablero[ni][nj] = ocupada;
}

void deshacerMovimiento(int i, int j, int ni, int nj) {
    tablero[i][j] = ocupada;
    tablero[(i + ni) / 2][(j + nj) / 2] = ocupada;
    tablero[ni][nj] = libre;
}
```

3.4.-Ejemplo paso a paso del funcionamiento.

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)

	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

3.4.-Ejemplo paso a paso del funcionamiento.

PROBLEMA3 SOLITARIO CHINO (SENKU)

- **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

3.4.-Ejemplo paso a paso del funcionamiento.

PROBLEMA3 SOLITARIO CHINO (SENKU)

- **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
3	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
3	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

PROBLEMA3 SOLITARIO CHINO (SENKU)

3.4.-Ejemplo paso a paso del funcionamiento.

- **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
3	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.

PROBLEMA3 SOLITARIO CHINO (SENKU)

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)
```

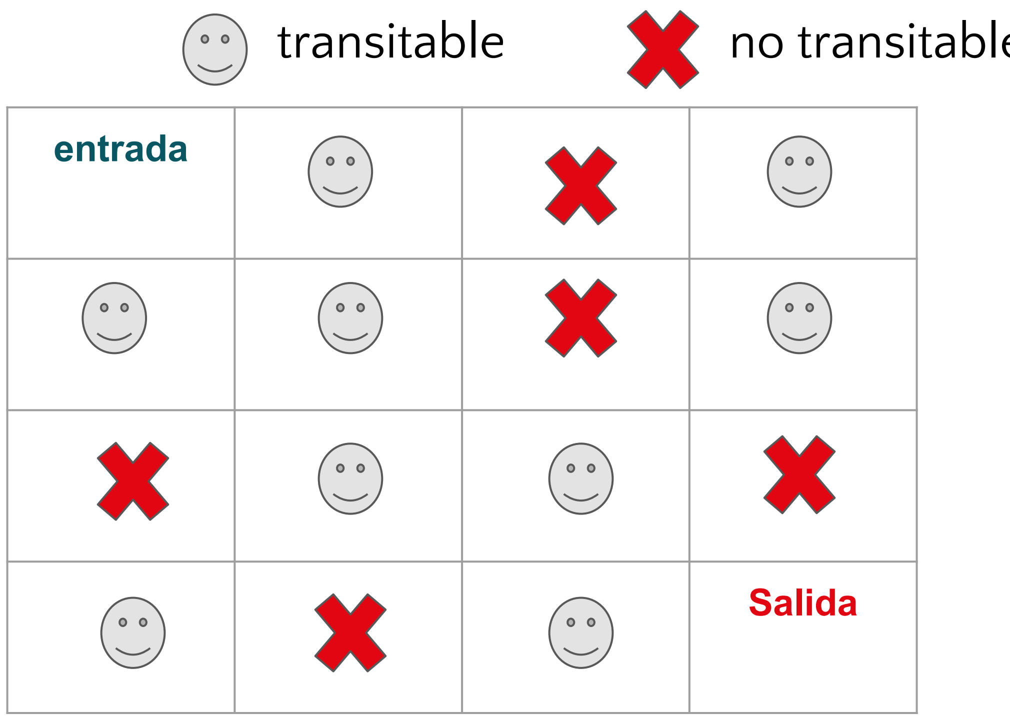
```
Movimiento nº22 : (0, 2) a (2, 2)
Movimiento nº23 : (3, 2) a (1, 2)
Movimiento nº24 : (2, 6) a (2, 4)
Movimiento nº25 : (2, 4) a (0, 4)
Movimiento nº26 : (2, 0) a (2, 2)
Movimiento nº27 : (2, 3) a (2, 1)
Movimiento nº28 : (0, 4) a (0, 2)
Movimiento nº29 : (0, 2) a (2, 2)
Movimiento nº30 : (2, 1) 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
```

4.1.-enunciado

PROBLEMA 4 Laberinto (backtracking)



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

- 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

4.2.-Restricciones

PROBLEMA 4
Laberinto
(backtracking)

Restricciones implícitas:

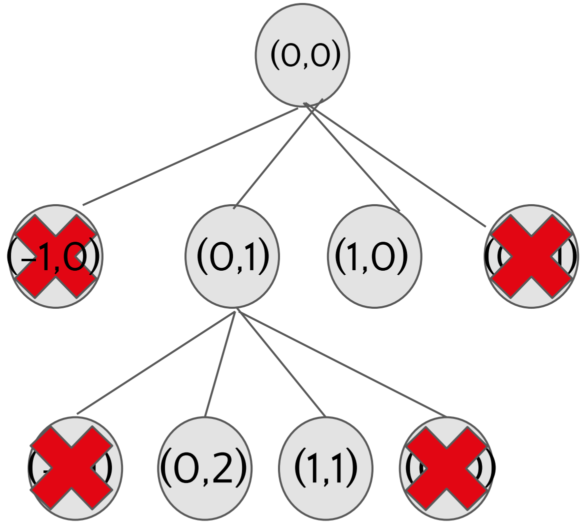
- Solo se permiten movimientos en posiciones adyacentes de la matriz en la misma fila o columna en la que nos encontramos
- Solo se permiten movimientos a casillas transitables(están marcadas como True en la matriz)
- Solo se permiten movimientos a posiciones que están dentro de los límites de la matriz

Restricciones explícitas:

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

4.3.-representacion del problema y algoritmo

PROBLEMA 4
Laberinto
(backtracking)



- 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

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.

4.2.-Pseudocodigo

PROBLEMA 4 Laberinto (backtracking)

```
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;
}
```

4.2.-Compilación y ejecución

PROBLEMA 4
Laberinto
(backtracking)

./ej4.1 4

```
T   T   T   F
T   T   F   T
F   T   T   T
F   T   T   T
( 0 , 0 ), ( 0 , 1 ), ( 1 , 1 ), ( 2 , 1 ), ( 2 , 2 ), ( 2 , 3 ), ( 3 , 3 ),
```

4.2.-Pseudocodigo

PROBLEMA 4
Laberinto
(backtracking)

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

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

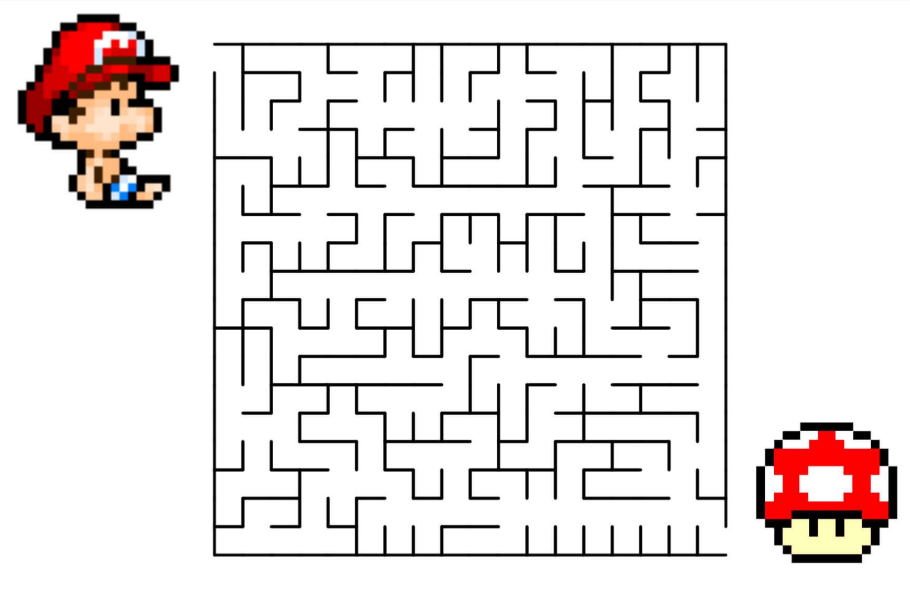
Entrada F	**F**	**T**	F
T	F	T	F
F	T	T	T
F	T	T	**Salida**

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

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

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

Implementación básico



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

Enunciado

Problema 5
Labertinto
B&B

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).

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.

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 el mejor camino transitable entre la entrada y la salida solo con movimientos en la misma fila o columna

Restricciones

Problema 5
Labertinto
B&B

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.

Diseño del algoritmo

Problema 5
Labertinto
B&B

- 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
```

Ejemplo de ejecucion

Problema 5
Labertinto
B&B

```
alberto@alberto-VirtualBox:~/Descargas/ALGP1$ g++ -o Bound bound.cpp
^[[Aalberto@alberto-VirtualBox:~/Descargas/ALGP1$ ./Bound 4
T   T   T   F
T   T   F   T
F   T   T   T
F   T   T   T
( 0 , 0 ), ( 1 , 0 ), ( 1 , 1 ), ( 2 , 1 ), ( 3 , 1 ), ( 3 , 2 ), ( 3 , 3 ),
```

Ejemplo paso a paso

Problema 5
Labertinto
B&B

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)]

Ejemplo paso a paso

Problema 5
Labertinto
B&B

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)]

Ejemplo paso a paso

Problema 5 Labertinto B&B

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)].

Ejemplo paso a paso

Problema 5 Labertinto B&B

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)].

Ejemplo paso a paso

Problema 5
Labertinto
B&B

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 = [].

La cola está vacía

Mejor camino = 6

Gracias