PRÁCTICA 4

BACKTRACKING

Marino Fernández Pérez,
Mehdi Iabouten,
Jorge López Molina,
Gabriele Ruggeri
y Pablo Vega Romero

Problema 1 con Backtracking

```cpp
// Función auxiliar para calcular el valor del emparejamiento
int calculaUnion(const parejas &prefs, const vector<int> &asignaciones) {
    int value = 0;
    for (int i = 0; i < asignaciones.size()-1; i+=2) {
        int estudiante_i = asignaciones[i];
        int estudiante_j = asignaciones[i+1];
        value += prefs[estudiante_i][estudiante_j] * prefs[estudiante_j][estudiante_i];
    }
    return value;
}
```

```cpp
/*Función que aplica Backtracking*/
void ParejasBacktracking(const parejas& prefs, vector<int>& asignaciones, int est, int& mejor_valor, vector<int>& best_asignaciones) {
    int n = prefs.size();
    if (est == n) {
        // Hemos emparejado a todos los estudiantes
        int valor_union = calculaUnion(prefs, asignaciones);

        if (valor_union > mejor_valor) {
            mejor_valor = valor_union;
            best_asignaciones = asignaciones;

        }
        return;
    }

    for (int estudiante = 0; estudiante < n; ++estudiante) {
        if (esValido(estudiante, asignaciones)) {
            asignaciones[est] = estudiante;
            ParejasBacktracking(prefs, asignaciones, est + 1, mejor_valor, best_asignaciones);
            asignaciones[est] = -1; // Deshacer la asignación
        }
    }
}
```

Para resolver el problema, el algoritmo Backtracking implementado, para cada estudiante, comprueba si el estudiante ya ha sido asignado a una pareja con otro estudiante. Si es válido, es decir, si no está asignado a ninguna pareja, se agrega y se realiza una llamada recursiva con el siguiente estudiante. El proceso continúa de manera recursiva hasta que se hayan comprobado todas las posibles parejas. Cada vez que se alcanza el caso base, es decir, cuando ya todos los estudiantes están asignados, se evalúa el valor de las uniones y se actualiza la mejor disposición encontrada si es mayor al que ya teníamos. Luego, se retrocede y se eliminan las asignaciones actuales para explorar otras posibilidades.

Función para comprobar si un estudiante es válido:

```cpp
/*Función auxiliar que nos permite comprobar si un estudiante ha sido emparejado o no.*/
bool esValido(int estudiante, const vector<int>& asignaciones) {
    for (int i = 0; i < asignaciones.size(); ++i) {
        if (asignaciones[i] == estudiante) {
            return false;
        }
    }
    return true;
}
```

Diseño del algoritmo:

'esValido(int estudiante, const vector<int> &asignaciones)': *Esta función auxiliar comprueba si un estudiante dado ha sido emparejado o no. Recorre el vector de asignaciones y verifica si el estudiante ya ha sido asignado.*

'calculaUnion(const parejas &prefs, const vector<int> &asignaciones)': *Esta función auxiliar calcula el valor de la unión de todas las parejas emparejadas. Itera sobre las asignaciones de los estudiantes, obteniendo las preferencias de cada pareja y sumando los valores de emparejamiento.*

'ParejasBacktracking(const parejas &prefs, vector<int> &asignaciones, int est, int &mejor_valor, vector<int> &best_asignaciones)': *Esta es la función principal de backtracking que busca todas las posibles combinaciones de parejas emparejadas. Recibe las preferencias de los estudiantes, el vector de asignaciones actual, el índice del estudiante actual que se está considerando, el mejor valor de unión encontrado hasta el momento y las mejores asignaciones encontradas hasta el momento. Utiliza la recursión para explorar todas las posibles combinaciones de emparejamiento y actualiza el mejor valor y las mejores asignaciones cuando encuentra una combinación que produce un valor de unión mayor.*

Pseudocódigo:

```
ParejasBacktracking(prefs, asignaciones, est, mejor_valor, best_asignaciones)
    if est == n
        // Hemos emparejado a todos los estudiantes
        valor_union = calculaUnion(prefs, asignaciones)
        if valor_union > mejor_valor
            mejor_valor = valor_union
            best_asignaciones = asignaciones
        end if

        return
    end if

    for estudiante = 0 to n-1
        if el estudiante no ha sido emparejado todavía
            asignar el estudiante a la posición est en las asignaciones
            ParejasBacktracking(prefs, asignaciones, est + 1, mejor_valor, best_asignaciones)
            deshacer la asignación del estudiante en la posición est
        end if
    end for
end ParejasBacktracking
```

Funcionamiento del algoritmo:

A la hora de analizar el funcionamiento de nuestro algoritmo, hemos escogido un ejemplo de uso donde el número de estudiantes es 4 y los puntos de preferencia para la matriz(los cuales representan las preferencias de un estudiante para formar pareja con otro) son pequeños con el objetivo de poder observar su funcionamiento fácilmente.

```cpp
int main() {
    int n = 4; // Número de estudiantes

    parejas prefs = {
        {0, 9,10, 17},
        {3, 0, 13, 13},
        {9, 4, 0, 12},
        {4, 6, 11, 0}
    };

    vector<int> asignaciones(n, -1); // Inicialización de asignaciones
    int mejor_valor = 0;
    vector<int> best_asignaciones(n, -1);

    // Ejecutar el algoritmo de backtracking recursivo
    ParejasBacktracking(prefs, asignaciones, 0, mejor_valor, best_asignaciones);

    int cuenta_salto = 0;
    cout << "Parejas formadas:" << endl;
    for (int estudiante : best_asignaciones){
        cout << estudiante;
        cuenta_salto++;

        if((cuenta_salto)% 2 == 0){
            cout << "\n";
        }
        else{
            cout << " emparejado con estudiante ";
        }
    }

    cout << "Valor: " << mejor_valor << endl;

    return 0;
}
```

Ejemplo de funcionamiento:

- Inicialización de datos: Se inicializa una matriz de enteros con los siguientes valores:
prefs.p = {
{0, 9, 10, 17},
{3, 0, 13, 13},
{9, 4, 0, 12},
{4, 6, 11, 0}
};

A tener en cuenta que:

```cpp
/*Definición del tipo para representar a las alumnos
y sus puntos de preferencia como una matriz*/
typedef vector<vector<int>> parejas;
```

Una vez tenemos la matriz de preferencias de los estudiantes llamamos a la función ParejasBacktracking, y vamos a explicar paso a paso cómo llega hasta la solución.

```
aulas@aulas-VirtualBox:~/Escritorio/ALGP4$ g++ problema1.cpp -o problema1
aulas@aulas-VirtualBox:~/Escritorio/ALGP4$ ./problema1
Parejas formadas:
0 emparejado con estudiante 2
1 emparejado con estudiante 3
Valor: 168
aulas@aulas-VirtualBox:~/Escritorio/ALGP4$
```

- Vamos a usar todas las asignaciones que hemos visto en el apartado anterior(en la declaración de la matriz).
- Nuestro algoritmo iría probando el alumno 0 con los demás alumnos y va cogiendo el valor más alto, así sucesivamente. Después igual con el 1, y así sucesivamente.
- Cuando ya ha hecho todas las combinaciones posibles y ha calculado todos los emparejamientos. Se coge el valor_union y lo imprime por pantalla junto a las mejores parejas.

Debido a la gran cantidad de comprobaciones y combinaciones posibles, para que el número de pasos no sea excesivo, vamos a comentar lo que ocurre en las primeras iteraciones ya que el algoritmo siempre comprueba y combina de la misma manera. Por tanto un ejemplo inicial bastaría para mostrar los pasos que sigue este algoritmo.

Primera Llamada Recursiva:

- *Se verifica si la el estudiante (est = 0) tiene el mismo valor que el número total de estudiantes(asignaciones.size() = 4). Luego, "estudiante != asignaciones.size()", por lo que no se cumple la condición.*
- *Se comprueba para el estudiante si ya se encuentra emparejado, es decir , si se encuentra en el vector de asignaciones.*
    - *asignaciones[0,-1,-1,-1]*
    - *El estudiante 0 ya se encuentra en el vector y pasamos a comprobar el siguiente estudiante*
    - *Llamada recursiva a "ParejasBacktracking" con el nuevo vector de asignaciones y el siguiente estudiante.*

Segunda Llamada Recursiva:

- *Se verifica si el estudiante actual (est = 1) tiene el mismo valor que el número total de estudiantes(asignaciones.size() = 4). Luego, "estudiante != asignaciones.size()", por lo que no se cumple la condición.*
- *Se comprueba para el estudiante si ya se encuentra emparejado, es decir , si se encuentra en el vector de asignaciones.*
    - *asignaciones[0, 1,-1,-1]*

- *El estudiante 1 ya se encuentra en el vector y pasamos a comprobar el siguiente estudiante*
- *Llamada recursiva a "ParejasBacktracking" con el nuevo vector de asignaciones y el siguiente estudiante.*

Tercera Llamada Recursiva:

- *Se verifica si el estudiante actual (est = 2) tiene el mismo valor que el número total de estudiantes(asignaciones.size() = 4). Luego, "estudiante != asignaciones.size()", por lo que no se cumple la condición.*
- *Se comprueba para el estudiante si ya se encuentra emparejado, es decir , si se encuentra en el vector de asignaciones.*
  - *asignaciones[0, 1, 2,-1]*
  - *El estudiante 2 ya se encuentra en el vector y pasamos a comprobar el siguiente estudiante*
  - *Llamada recursiva a "ParejasBacktracking" con el nuevo vector de asignaciones y el siguiente estudiante.*

Cuarta Llamada Recursiva:

- *Se verifica si el estudiante actual (est = 3) tiene el mismo valor que el número total de estudiantes(asignaciones.size() = 4). Luego, "estudiante != asignaciones.size()", por lo que no se cumple la condición.*
- *Se comprueba para el estudiante si ya se encuentra emparejado, es decir , si se encuentra en el vector de asignaciones.*
  - *asignaciones[0, 1, 2, 3]*
  - *El estudiante 3 ya se encuentra en el vector y pasamos a comprobar el siguiente estudiante*
  - *Llamada recursiva a "ParejasBacktracking" con el nuevo vector de asignaciones y el siguiente estudiante.*

Quinta Llamada Recursiva:

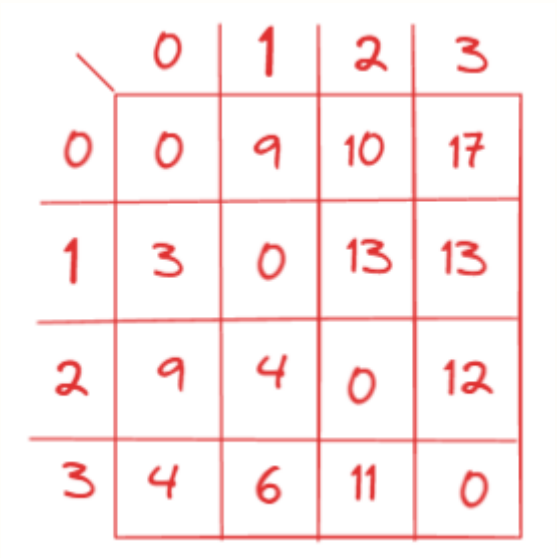
- *Se verifica si el estudiante actual (est = 4) tiene el mismo valor que el número total de estudiantes(asignaciones.size() = 4). Luego, "estudiante == asignaciones.size()", por lo que se cumple la condición.*
- *Se pasa a calcular los valores de las parejas y se retorna el valor, se comprueba si es mayor, en caso de que lo sea se modifica a dicho valor y se iguala el vector best_asignaciones al de asignaciones.*
- *En el caso de estas primeras iteraciones el estado quedaría de esta manera:*
  - *asignaciones[0, 1, 2, 3]*
  - *Calculámos el valor de las uniones actuales*
    *-Pareja 1(0,1) = 9 * 3 = 27*
    *-Pareja 2(2,3) = 11 * 12 = 132*
    La suma total de preferencia de estas parejas genera un valor de 159.
  - *mejor_valor = 159*

Esto anterior se trata de un ejemplo de cómo funciona el algoritmo en las primeras iteraciones y llamadas, proceso que se repite hasta que se hayan recorrido todas las posibilidades.

FINAL

Una vez que se han realizado todas y cada una de las iteraciones con sus respectivas llamadas recursivas obtenemos que el mejor valor es de ***168*** *debido a que ninguna de las uniones restantes a evaluar han obtenido un nivel de conveniencia mayor al màximo que encontramos en el vector de asignaciones con los valores [0,2,1,3].*

Para comprobarlo vamos a mostrarlo con un dibujo que lo represente:



	0	1	2	3
0	0	9	10	17
1	3	0	13	13
2	9	4	0	12
3	4	6	11	0

Esta sería la representación como matriz de las preferencias, donde cada fila indica la preferencia de un estudiante de formar pareja con el resto. Como es obvio, un estudiante no puede formar pareja consigo mismo, por lo que el valor es 0, lo que hace que se forme una diagonal de 0.
A continuación procederemos con los cálculos a mano, de las distintas combinaciones:
-Para una asignación de (0, 1) y (2, 3):
El valor sería: (9x3) + (12x11) = 159
-Para una asignación de (0, 2) y (1, 3):
El valor sería: (10x9) + (13x6) = 168
-Para una asignación de (0, 3) y (1, 2):
El valor sería: (17x4) + (13x4) = 120

Como vemos el mayor valor es el que el programa nos muestra como resultado, es decir, ***168.***
Esto nos permite ver que el programa funciona correctamente.

Problema 2 con Backtracking

```cpp
int calcularConvenienciaTotal(const vector<vector<int>>& conveniencia, const vector<int>& disposicion){
    int nivel_total = 0;
    int num_invitados = disposicion.size();

    for (int i = 0; i < num_invitados; i++){
        int actual = disposicion[i];
        int dcha = disposicion[(i + 1) % num_invitados];
        int izda = disposicion[(i - 1 + num_invitados) % num_invitados];

        nivel_total += conveniencia[actual][izda] + conveniencia[actual][dcha];
    }

    return nivel_total;
}

void disposicionBacktracking(const vector<vector<int>> &conveniencia, vector<bool> &sentados, int num_invitados, vector<int> &
disposicion_actual, int &mejor_conveniencia, vector<int> &mejor_disposicion) {
    if (disposicion_actual.size() == num_invitados) {
        int conveniencia_actual = calcularConvenienciaTotal(conveniencia, disposicion_actual);

        if (conveniencia_actual > mejor_conveniencia) {
            mejor_disposicion = disposicion_actual;
            mejor_conveniencia = conveniencia_actual;
        }
        return;
    }

    for (int i = 0; i < num_invitados; i++) {
        if (not sentados[i]){
            sentados[i] = true;
            disposicion_actual.push_back(i);
            disposicionBacktracking(conveniencia, sentados, num_invitados, disposicion_actual, mejor_conveniencia, mejor_disposicion);
            disposicion_actual.pop_back();
            sentados[i] = false;
        }
    }
}
```

Diseño del algoritmo:

Para resolver el problema, el algoritmo Backtracking implementado comprueba, para cada invitado, si se encuentra ya en la disposición provisional a evaluar. Si no está, se agrega y se realiza una llamada recursiva con la nueva disposición. El proceso continúa de manera recursiva hasta que se hayan explorado todas las posibles disposiciones de invitados. Cada vez que se alcanza el caso base, es decir, cuando ya todos los invitados están sentados, se evalúa la disposición actual y se actualiza la mejor disposición encontrada si es necesario, es decir, si el nivel de conveniencia obtenido por medio de la función auxiliar "calcularConveniencia Total" es mayor a todos los conseguidos en las disposiciones provisionales ya evaluadas. Luego, se retrocede y se eliminan los invitados agregados para explorar otras posibilidades.

- ***calcularConveniencia Actual:***

  Esta función nos permite calcular el nivel total de conveniencia para una disposición dada de invitados en una mesa. Toma como entrada una matriz de conveniencia que contiene los niveles de conveniencia entre cada par de invitados, y un vector que representa la disposición actual de invitados en la mesa.

  - *Recorre cada invitado en la mesa y calcula la conveniencia de sentarse junto al invitado de la izquierda y al de la derecha.*
  - *Suma estos valores al nivel total de conveniencia de forma acumulativa*
  - *Devuelve el nivel total de conveniencia.*

Esta nos va a servir para calcular la conveniencia de cada una de las disposiciones provisionales/candidatas que pueden optar a ser la mejor disposición para dicha matriz de conveniencia

```
function calcularConvenienciaTotal(conveniencia, disposicion)
   "conveniencia" es una matriz de enteros donde cada posición (i, j) representa el nivel de conveniencia entre el invitado i y el invitado j

   "disposicion" es un vector que representa la disposición final de invitados

   nivel total = 0
   numero de invitados = tamaño de la disposicion obtenida

   Para cada uno de los invitados (i) :
      actual = invitado actual
      dcha = invitado que se encuentra a la derecha en el vector mejor_disposicion
      izda = invitado que se encuentra a la izquierda en el vector mejor_disposicion

      nivel total += suma de conveniencia entre el invitado actual y los invitados que están a su izquierda y derecha
   end for

   devolvemos nivel total
end function
```

- ***disposicionBacktracking:***

La siguiente función puede dividirse en dos bloques:

Por un lado, el bloque encargado de comprobar si la disposición obtenida y candidata donde todos los invitados se encuentran sentados en la mesa tiene un nivel de conveniencia mayor al "supuesto" nivel máximo obtenido con otra disposición.

Por otro lado, el bloque encargado de construir todas y cada una de las posibles disposiciones teniendo en cuenta las distintas combinaciones y posiciones en las cuales se pueden asignar todos y cada uno de los invitados y por medio de llamadas recursivas.

- *Verifica si la disposición actual contiene a todos los invitados.*
    - *Si es así, calcula el nivel total de conveniencia para esa disposición y lo compara con el mejor nivel de conveniencia encontrado hasta el momento.*
- *Si la disposición actual no contiene a todos los invitados, prueba a sentar a cada invitado que aún no está sentado en la mesa y realiza una llamada recursiva para probar todas las posibles disposiciones*

```
68  function disposicionBacktracking(conveniencia, sentados, num_invitados, disposicion_actual, mejor_conveniencia, mejor_disposicion)
69     si la disposicion_actual tiene un tamaño igual al numero de invitados entonces:
70        conveniencia_actual = nivel de conveniencia de la disposicion actual
71
72        si conveniencia_actual es mayor a mejor_conveniencia entonces:
73           mejor_disposicion = disposicion_actual
74           mejor_conveniencia = conveniencia_actual
75        end si
76
77        retornamos
78     end Si
79
80     Para cada uno de los invitados (i):
81        si el invitado actual no está sentado entonces:
82           sentado(invitado actual) = verdadero
83           agregamos el invitado actual a la disposicion actual
84           llamamos a la funcion disposicionBacktracking con la disposicion actual y la conveniencia actual
85           eliminamos el invitado actual de la disposicion actual
86           sentado(invitado actual) = falso
87        end si
88     end for
89  end function
```

Funcionamiento del algoritmo

A la hora de analizar el funcionamiento de nuestro algoritmo, hemos escogido un ejemplo de uso donde el número de invitados es 4 y los niveles de conveniencia para la matriz son pequeños con el objetivo de poder observar su funcionamiento fácilmente. Anteriormente se escogió un ejemplo de uso compuesto por 3 invitados, luego hemos pensado que realmente no es capaz de mostrar al cien por cien el funcionamiento de nuestro algoritmo ya que, independientemente de dónde estén colocados los invitados, siempre van a estar relacionados todos con todos y por lo tanto el nivel de conveniencia siempre va a ser el mismo.

```
45      // matriz ejemplo
46      int num_invitados = 4;
47      vector<vector<int>> conveniencia = {
48      {0, 20, 15, 10},
49      {20, 0, 10, 5},
50      {15, 10, 0, 10},
51      {10, 5, 10, 0}
52      };
53
54
55      vector<int> mejor_disposicion;
56      vector<int> disposicion_actual;
57      vector<bool> sentados(num_invitados, false);
58      int mejor_conveniencia = 0;
59
60      disposicionBacktracking(conveniencia, sentados, num_invitados, disposicion_actual, mejor_conveniencia, mejor_disposicion);
61
62      cout << "La disposición de invitados que maximiza el nivel de conveniencia total es:" << endl;
63      for (int i = 0; i < num_invitados; i++) {
64          cout << "[" << mejor_disposicion[i] << "]" << " ";
65      }
66
67      cout << endl;
68      cout << "El nivel de conveniencia total es: " << mejor_conveniencia << endl;
69  }
```

Para una disposición vacía que finalmente acabará tomando un tamaño = 4 debido al número de invitados tenemos que su tamaño inicial es: disposicion_actual.size() = 0;

- *Para la disposición actual con la que contamos comprobamos si su tamaño es igual al del número de invitados, es decir, si todos los invitados están asignados provisionalmente a cada uno de los asientos. Como disposicion_actual.size() != 0, abandonamos el bloque condicional y por lo tanto, el caso base.*

- *Para cada uno de los invitados, comprobamos si se encuentra ya asignado a un asiento en la mesa por medio del vector de booleanos "sentados". Si no está sentado, lo añadimos a la disposición actual provisionalmente, al igual que hacemos con sentados, atribuyéndole el valor true.*
  Seguidamente, por medio de recursividad, se evalúa la disposición actual con el nuevo invitado añadido, de forma que se partirá con una disposición con un invitado más, estudiando todas y cada una de las distintas posibilidades. Una vez finaliza la llamada recursiva, es decir, una vez que tenemos una disposición de tamaño "num_invitados", se evalúa su nivel de conveniencia y para finalizar, abandonamos la recursividad por medio de un return.
  Una vez que hemos vuelto a la llamada anterior a la función, eliminamos al último invitado agregado a la disposición.

El invitado 0 aún no ha sido sentado a la mesa, luego:

***)** *sentados[true, false, false, false]*

***)** *disposición_actual[0]*

<u>Primera Llamada Recursiva:</u>

- *Se verifica si la disposición actual (tamaño = 1) tiene el mismo tamaño que el número total de invitados (num_invitados = 4). Luego, "disposicion_actual.size() != num_invitados", por lo que no se cumple la condición.*
- *Se comprueba para cada uno de los invitados cuál ya se encuentra sentado en la mesa, y por lo tanto, en la disposición actual.*
    - *disposicion_actual[0]*
    - *sentados[true, false, false, false]*
    - *El invitado 0 ya se encuentra sentado en la mesa y pasamos a comprobar el siguiente invitado*
    - *El invitado 1 no está sentado en la mesa, luego:*
        - *sentados[true, true, false, false]*
        - *disposicion_actual[0, 1]*
        - *Llamada recursiva a "disposiciónBacktracking" con la nueva disposición*

<u>Segunda Llamada Recursiva:</u>

- *Se verifica si la disposición actual (tamaño = 2) tiene el mismo tamaño que el número total de invitados (num_invitados = 4). Luego, "disposicion_actual.size() != num_invitados", por lo que no se cumple la condición.*
- *Se comprueba para cada uno de los invitados cuál ya se encuentra sentado en la mesa, y por lo tanto, en la disposición actual.*
    - *disposicion_actual[0, 1]*
    - *sentados[true, true, false, false]*
    - *El invitado 0 ya se encuentra sentado en la mesa y pasamos a comprobar el siguiente invitado*
    - *El invitado 1 ya se encuentra sentado en la mesa y pasamos a comprobar el siguiente invitado*
    - *El invitado 2 no está sentado en la mesa, luego:*
        - *sentados[true, true, true, false]*
        - *disposicion_actual[0, 1, 2]*
        - *Llamada recursiva a "disposiciónBacktracking" con la nueva disposición*

Tercera Llamada Recursiva:

- *Se verifica si la disposición actual (tamaño = 3) tiene el mismo tamaño que el número total de invitados (num_invitados = 4). Luego, "disposicion_actual.size() != num_invitados", por lo que no se cumple la condición.*
- *Se comprueba para cada uno de los invitados cuál ya se encuentra sentado en la mesa, y por lo tanto, en la disposición actual.*
    - *disposicion_actual[0, 1, 2]*
    - *sentados[true, true, true, false]*
    - *El invitado 0 ya se encuentra sentado en la mesa y pasamos a comprobar el siguiente invitado*
    - *El invitado 1 ya se encuentra sentado en la mesa y pasamos a comprobar el siguiente invitado*
    - *El invitado 2 ya se encuentra sentado en la mesa y pasamos a comprobar el siguiente invitado*
    - *El invitado 3 no está sentado en la mesa, luego:*
        - *sentados[true, true, true, true]*
        - *disposicion_actual[0, 1, 2, 3]*
        - *Llamada recursiva a "disposiciónBacktracking" con la nueva disposición*

Cuarta Llamada Recursiva:

- *Se verifica si la disposición actual (tamaño = 4) tiene el mismo tamaño que el número total de invitados (num_invitados = 4). Luego, "disposicion_actual.size() == num_invitados", por lo que se cumple la condición.*
- *Calculamos nivel de conveniencia para la disposición provisional por medio de la función "calcularConvenienciaTotal":*
    - *NIVEL DE CONVENIENCIA ACTUAL = 100*

- *Como el nivel de conveniencia es mayor que el mejor nivel hasta ahora (0):*
    - *mejor_conveniencia = 100;*
    - *mejor_disposicion = disposicion_actual = [0,1,2,3]*

- *Se ejecuta un "return" para poner fin a la llamada recursiva y retroceder a la llamada anterior, donde nos volvemos a encontrar en el bloque de cómputo del bucle for:*
    - *Eliminamos el último invitado que se sentó en la mesa → invitado 3*
        - *disposicion_actual[0,1,2]*
        - *sentado[true, true, true, false]*
    - *Como no hay más invitados que explorar, retrocedemos a la llamada anterior*
        - *Eliminamos el último invitado que se sentó en la mesa → invitado 2*
        - *disposicion_actual[0,1]*
        - *sentado[true, true, false, false]*
    - *Como nos encontramos en la tercera iteración del bucle for, aún nos queda por evaluar el invitado 3 (tercera iteración), que ya no está sentado a la mesa:*
        - *sentados[true, true, false, true]*
        - *disposicion_actual[0, 1, 3]*
        - *Llamada recursiva a "disposiciónBacktracking" con la nueva disposición*

<u>Quinta Llamada Recursiva:</u>

- *Se verifica si la disposición actual (tamaño = 3) tiene el mismo tamaño que el número total de invitados (num_invitados = 4). Luego, "disposicion_actual.size() != num_invitados", por lo que no se cumple la condición.*
- *Se comprueba para cada uno de los invitados cuál ya se encuentra sentado en la mesa, y por lo tanto, en la disposición actual.*
    - *disposicion_actual[0, 1, 3]*
    - *sentados[true, true, false, true]*
    - *El invitado 0 ya se encuentra sentado en la mesa y pasamos a comprobar el siguiente invitado*
    - *El invitado 1 ya se encuentra sentado en la mesa y pasamos a comprobar el siguiente invitado*
    - *El invitado 3 ya se encuentra sentado en la mesa y pasamos a comprobar el siguiente invitado*
    - *El invitado 2 no está sentado en la mesa, luego:*
        - *sentados[true, true, true, true]*
        - *disposicion_actual[0, 1, 3, 2]*
        - *Llamada recursiva a "disposiciónBacktracking" con la nueva disposición*

Sexta Llamada Recursiva:

- *Se verifica si la disposición actual (tamaño = 4) tiene el mismo tamaño que el número total de invitados (num_invitados = 4). Luego, "disposicion_actual.size() == num_invitados", por lo que se cumple la condición.*
- *Calculamos nivel de conveniencia para la disposición provisional por medio de la función "calcularConveniencia Total":*
    - *NIVEL DE CONVENIENCIA ACTUAL = 100*

- *Como el nivel de conveniencia no es mayor que el mejor nivel hasta ahora (100), se mantienen los valores anteriores:*
    - *mejor_conveniencia = 100;*
    - *mejor_disposicion = disposicion_actual = [0,1,2,3]*

- *Se ejecuta un "return" para poner fin a la llamada recursiva y retroceder a la llamada anterior, donde nos volvemos a encontrar en el bloque de cómputo del bucle for:*
    - *Eliminamos el último invitado que se sentó en la mesa → invitado 2*
        - *disposicion_actual[0,1,3]*
        - *sentado[true, true, false, true]*
    - *Como no hay más invitados que explorar, retrocedemos a la llamada anterior*
        - *Eliminamos el último invitado que se sentó en la mesa → invitado 3*
        - *disposicion_actual[0, 1]*
        - *sentado[true, true, false, false]*
    - *Como no hay más invitados que explorar, retrocedemos a la llamada anterior*
        - *Eliminamos el último invitado que se sentó en la mesa → invitado 1*
        - *disposicion_actual[0]*
        - *sentado[true, false, false, false]*
    - *Como nos encontramos en la segunda iteración del bucle for, aún nos queda por evaluar el invitado 2 y 3, que ya no están sentados a la mesa:*
        - *sentado[true, false, true, false]*
        - *disposicion_actual[0, 2]*
        - *Llamada recursiva a "disposiciónBacktracking" con la nueva disposición*

Séptima Llamada Recursiva:

- *Se verifica si la disposición actual (tamaño = 2) tiene el mismo tamaño que el número total de invitados (num_invitados = 4). Luego, "disposicion_actual.size() != num_invitados", por lo que no se cumple la condición.*
- *Se comprueba para cada uno de los invitados cuál ya se encuentra sentado en la mesa, y por lo tanto, en la disposición actual.*
    - *disposicion_actual[0, 2]*
    - *sentado[true, false, true, false]*
    - *El invitado 0 ya se encuentra sentado en la mesa y pasamos a comprobar el siguiente invitado*
    - *El invitado 2 ya se encuentra sentado en la mesa y pasamos a comprobar el siguiente invitado*
    - *El invitado 1 no está sentado en la mesa, luego:*
        - *sentados[true, true, true, false]*
        - *disposicion_actual[0, 2, 1]*
        - *Llamada recursiva a "disposiciónBacktracking" con la nueva disposición*

Octava Llamada Recursiva:

- *Se verifica si la disposición actual (tamaño = 3) tiene el mismo tamaño que el número total de invitados (num_invitados = 4). Luego, "disposicion_actual.size() != num_invitados", por lo que no se cumple la condición.*
- *Se comprueba para cada uno de los invitados cuál ya se encuentra sentado en la mesa, y por lo tanto, en la disposición actual.*
    - *disposicion_actual[0, 2, 1]*
    - *sentados[true, true, true, false]*
    - *El invitado 1 ya se encuentra sentado en la mesa y pasamos a comprobar el siguiente invitado*
    - *El invitado 0 ya se encuentra sentado en la mesa y pasamos a comprobar el siguiente invitado*
    - *El invitado 2 ya se encuentra sentado en la mesa y pasamos a comprobar el siguiente invitado*
    - *El invitado 3 no está sentado en la mesa, luego:*
        - *sentados[true, true, true, true]*
        - *disposicion_actual[0, 2, 1, 3]*
        - *Llamada recursiva a "disposiciónBacktracking" con la nueva disposición*

Novena Llamada Recursiva:

- *Se verifica si la disposición actual (tamaño = 4) tiene el mismo tamaño que el número total de invitados (num_invitados = 4). Luego, "disposicion_actual.size() == num_invitados", por lo que se cumple la condición.*
- *Calculamos nivel de conveniencia para la disposición provisional por medio de la función "calcularConveniencia Total":*
    - *NIVEL DE CONVENIENCIA ACTUAL = 70*

- *Como el nivel de conveniencia es menor que el mejor nivel hasta ahora (100), no se cumple la condición y se mantienen los mismos valores para las variables:*
    - *mejor_conveniencia = 100;*
    - *mejor_disposicion = disposicion_actual = [0,1,2,3]*

- *Se ejecuta un "return" para poner fin a la llamada recursiva y retroceder a la llamada anterior, donde nos volvemos a encontrar en el bloque de cómputo del bucle for:*
    - *Eliminamos el último invitado que se sentó en la mesa → invitado 3*
        - *disposicion_actual[0, 2, 1]*
        - *sentado[true, true, true, false]*
    - *Como no hay más invitados que explorar, retrocedemos a la llamada anterior*
        - *Eliminamos el último invitado que se sentó en la mesa → invitado 1*
        - *disposicion_actual[0,2]*
        - *sentado[true, false, true, false]*
    - *Como nos encontramos en la tercera iteración del bucle for, aún nos queda por evaluar el invitado 3, que ya no está sentado a la mesa:*
        - *sentado[true, false, true, true]*
        - *disposicion_actual[0, 2, 3]*
        - *Llamada recursiva a "disposiciónBacktracking" con la nueva disposición*

Décima Llamada Recursiva:

- *Se verifica si la disposición actual (tamaño = 3) tiene el mismo tamaño que el número total de invitados (num_invitados = 4). Luego, "disposicion_actual.size() != num_invitados", por lo que no se cumple la condición.*
- *Se comprueba para cada uno de los invitados cuál ya se encuentra sentado en la mesa, y por lo tanto, en la disposición actual.*
    - *disposicion_actual[0, 2, 3]*
    - *sentado[true, false, true, true]*
    - *El invitado 0 ya se encuentra sentado en la mesa y pasamos a comprobar el siguiente invitado*
    - *El invitado 2 ya se encuentra sentado en la mesa y pasamos a comprobar el siguiente invitado*
    - *El invitado 3 ya se encuentra sentado en la mesa y pasamos a comprobar el siguiente invitado*
    - *El invitado 1 no está sentado en la mesa, luego:*
        - *sentados[true, true, true, true]*
        - *disposicion_actual[0, 2, 3, 1]*
        - *Llamada recursiva a "disposiciónBacktracking" con la nueva disposición*

Onceava Llamada Recursiva:

- *Se verifica si la disposición actual (tamaño = 4) tiene el mismo tamaño que el número total de invitados (num_invitados = 4). Luego, "disposicion_actual.size() == num_invitados", por lo que se cumple la condición.*
- *Calculamos nivel de conveniencia para la disposición provisional por medio de la función "calcularConvenienciaTotal":*
    - *NIVEL DE CONVENIENCIA ACTUAL = 100*

- *Como el nivel de conveniencia no es mayor que el mejor nivel hasta ahora (100), se mantienen los valores anteriores:*
    - *mejor_conveniencia = 100;*
    - *mejor_disposicion = disposicion_actual = [0,1,2,3]*

- *Se ejecuta un "return" para poner fin a la llamada recursiva y retroceder a la llamada anterior, donde nos volvemos a encontrar en el bloque de cómputo del bucle for:*
    - *Eliminamos el último invitado que se sentó en la mesa → invitado 1*
        - *disposicion_actual[0, 2, 3]*
        - *sentado[true, false, true, true]*
    - *Como no hay más invitados que explorar, retrocedemos a la llamada anterior*
        - *Eliminamos el último invitado que se sentó en la mesa → invitado 3*
        - *disposicion_actual[0, 2]*
        - *sentado[true, false, true, false]*
    - *Como no hay más invitados que explorar, retrocedemos a la llamada anterior*
        - *Eliminamos el último invitado que se sentó en la mesa → invitado 2*
        - *disposicion_actual[0]*
        - *sentado[true, false, false, false]*
    - *Como nos encontramos en la cuarta iteración del bucle for, aún nos queda por evaluar el invitado 3, que ya no está sentado a la mesa:*
        - *sentado[true, false, false, true]*
        - *disposicion_actual[0, 3]*
        - *Llamada recursiva a "disposiciónBacktracking" con la nueva disposición*

<u>Doceava Llamada Recursiva:</u>

- *Se verifica si la disposición actual (tamaño = 2) tiene el mismo tamaño que el número total de invitados (num_invitados = 4). Luego, "disposicion_actual.size() != num_invitados", por lo que no se cumple la condición.*
- *Se comprueba para cada uno de los invitados cuál ya se encuentra sentado en la mesa, y por lo tanto, en la disposición actual.*
    - *disposicion_actual[0, 3]*
    - *sentado[true, false, false, true]*
    - *El invitado 0 ya se encuentra sentado en la mesa y pasamos a comprobar el siguiente invitado*
    - *El invitado 3 ya se encuentra sentado en la mesa y pasamos a comprobar el siguiente invitado*
    - *El invitado 1 no está sentado en la mesa, luego:*
        - *sentado[true, true, false, true]*
        - *disposicion_actual[0, 3, 1]*
        - *Llamada recursiva a "disposiciónBacktracking" con la nueva disposición*

Treceava Llamada Recursiva:

- *Se verifica si la disposición actual (tamaño = 3) tiene el mismo tamaño que el número total de invitados (num_invitados = 4). Luego, "disposicion_actual.size() != num_invitados", por lo que no se cumple la condición.*
- *Se comprueba para cada uno de los invitados cuál ya se encuentra sentado en la mesa, y por lo tanto, en la disposición actual.*
    - *disposicion_actual[0, 3, 1]*
    - *sentado[true, true, false, true]*
    - *El invitado 0 ya se encuentra sentado en la mesa y pasamos a comprobar el siguiente invitado*
    - *El invitado 1 ya se encuentra sentado en la mesa y pasamos a comprobar el siguiente invitado*
    - *El invitado 3 ya se encuentra sentado en la mesa y pasamos a comprobar el siguiente invitado*
    - *El invitado 2 no está sentado en la mesa, luego:*
        - *sentado[true, true, true, true]*
        - *disposicion_actual[0, 3, 1, 2]*
        - *Llamada recursiva a "disposiciónBacktracking" con la nueva disposición*

Catorceava Llamada Recursiva:

- *Se verifica si la disposición actual (tamaño = 4) tiene el mismo tamaño que el número total de invitados (num_invitados = 4). Luego, "disposicion_actual.size() == num_invitados", por lo que se cumple la condición.*
- *Calculamos nivel de conveniencia para la disposición provisional por medio de la función "calcularConveniencia Total":*
    - *NIVEL DE CONVENIENCIA ACTUAL = 70*

- *Como el nivel de conveniencia es menor que el mejor nivel hasta ahora (100):*
    - *mejor_conveniencia = 100;*
    - *mejor_disposicion = disposicion_actual = [0,1,2,3]*

- *Se ejecuta un "return" para poner fin a la llamada recursiva y retroceder a la llamada anterior, donde nos volvemos a encontrar en el bloque de cómputo del bucle for:*
    - *Eliminamos el último invitado que se sentó en la mesa → invitado 2*
        - *disposicion_actual[0, 3, 1]*
        - *sentado[true, true, false, true]*
    - *Como no hay más invitados que explorar, retrocedemos a la llamada anterior*
        - *Eliminamos el último invitado que se sentó en la mesa → invitado 1*
        - *disposicion_actual[0, 3]*
        - *sentado[true, false, false, true]*
    - *Como nos encontramos en la tercara iteración (it = 2) del bucle for, aún nos queda por evaluar el invitado 2, que ya no está sentado a la mesa:*
        - *sentado[true, false, true, true]*
        - *disposicion_actual[0, 3, 2]*
        - *Llamada recursiva a "disposiciónBacktracking" con la nueva disposición*

<u>Quinceava Llamada Recursiva:</u>

- *Se verifica si la disposición actual (tamaño = 3) tiene el mismo tamaño que el número total de invitados (num_invitados = 4). Luego, "disposicion_actual.size() != num_invitados", por lo que no se cumple la condición.*
- *Se comprueba para cada uno de los invitados cuál ya se encuentra sentado en la mesa, y por lo tanto, en la disposición actual.*
    - *disposicion_actual[0, 3, 2]*
    - *sentado[true, false, true, true]*
    - *El invitado 0 ya se encuentra sentado en la mesa y pasamos a comprobar el siguiente invitado*
    - *El invitado 2 ya se encuentra sentado en la mesa y pasamos a comprobar el siguiente invitado*
    - *El invitado 3 ya se encuentra sentado en la mesa y pasamos a comprobar el siguiente invitado*
    - *El invitado 1 no está sentado en la mesa, luego:*
        - *sentados[true, true, true, true]*
        - *disposicion_actual[0, 3, 2, 1]*
        - *Llamada recursiva a "disposiciónBacktracking" con la nueva disposición*

Dieciseisava Llamada Recursiva:

- *Se verifica si la disposición actual (tamaño = 4) tiene el mismo tamaño que el número total de invitados (num_invitados = 4). Luego, "disposicion_actual.size() == num_invitados", por lo que se cumple la condición.*
- *Calculamos nivel de conveniencia para la disposición provisional por medio de la función "calcularConveniencia Total":*
    - *NIVEL DE CONVENIENCIA ACTUAL = 90*

- *Como el nivel de conveniencia no es mayor que el mejor nivel hasta ahora (100), se mantienen los valores anteriores:*
    - *mejor_conveniencia = 100;*
    - *mejor_disposicion = disposicion_actual = [0,1,2, 3]*

- *Se ejecuta un "return" para poner fin a la llamada recursiva y retroceder a la llamada anterior, donde nos volvemos a encontrar en el bloque de cómputo del bucle for:*
    - *Eliminamos el último invitado que se sentó en la mesa → invitado 1*
        - *disposicion_actual[0, 3, 2]*
        - *sentados[true, false, true, true]*
    - *Como no hay más invitados que explorar, retrocedemos a la llamada anterior*
        - *Eliminamos el último invitado que se sentó en la mesa → invitado 2*
        - *disposicion_actual[0, 3]*
        - *sentados[true, false, false, true]*
    - *Como no hay más invitados que explorar, retrocedemos a la llamada anterior*
        - *Eliminamos el último invitado que se sentó en la mesa → invitado 3*
        - *disposicion_actual[0]*
        - *sentados[true, false, false, false]*
    - *Como no hay más invitados que explorar, retrocedemos a la llamada anterior*
        - *Eliminamos el último invitado que se sentó en la mesa → invitado 0*
        - *disposicion_actual[]*
        - *sentados[false, false, false, false]*
    - *Como nos encontramos en la primera iteración del bucle for, aún nos queda por evaluar el invitado 1, que ya no está sentado a la mesa:*
        - *sentados[false, true, false, false]*
        - *disposicion_actual[1]*
        - *Llamada recursiva a "disposiciónBacktracking" con la nueva disposición*

A partir de este punto, se vuelve a repetir todo el proceso realizado anteriormente probando todas y cada una de las disposiciones así como calculando su nivel de conveniencia total.

Finalmente, la solución obtenida es la siguiente:

FINAL

Una vez que se han realizado todas y cada una de las iteraciones con sus respectivas llamadas recursivas obtenemos que el nivel de conveniencia máximo es de ***100*** *debido a que ninguna de las disposiciones restantes a evaluar han obtenido un nivel de conveniencia mayor al màximo que encontramos en la disposición [0,1,2,3]*

Problema 3: Senku

```
bool movimientoValido(int fila, int columna, int direccion_f, int direccion_c){
    if (fila+2*direccion_f < 0 || fila+2*direccion_f >= 7 || columna+2*direccion_c < 0 || columna+2*direccion_c >= 7)
        return false;
    return tablero[fila][columna] == 'O' && tablero[fila+direccion_f][columna+direccion_c] == 'O' && tablero[fila+2*direccion_f][columna+2*direccion_c] ==
}

void realizaMovimiento(int fila, int columna, int direccion_f, int direccion_c){
    tablero[fila][columna] = ' ';
    tablero[fila+direccion_f][columna+direccion_c] = ' ';
    tablero[fila+2*direccion_f][columna+2*direccion_c] = 'O';
    movimientos.push({{fila, columna},{2*direccion_f, 2*direccion_c}});
}

void deshacerMovimiento(int fila, int columna, int direccion_f, int direccion_c){
    tablero[fila][columna] = 'O';
    tablero[fila+direccion_f][columna+direccion_c] = 'O';
    tablero[fila+2*direccion_f][columna+2*direccion_c] = ' ';
    movimientos.pop();
}

bool backtracking(int piezas_restantes){
    if (piezas_restantes == 1 && tablero[3][3] == 'O')
        return true;

    for (int fila = 0; fila < 7; fila++){
        for (int columna = 0; columna < 7; columna++){
            if (tablero[fila][columna] != 'O')
                continue;
            for (int direccion_f = -1; direccion_f <= 1; direccion_f++){
                for (int direccion_c = -1; direccion_c <= 1; direccion_c++){
                    if ((direccion_f == 0 && direccion_c == 0) || (direccion_f != 0 && direccion_c != 0)) //Para que no haya movimientos diagonales
                        continue;
                    if (movimientoValido(fila, columna, direccion_f, direccion_c)){
                        realizaMovimiento(fila, columna, direccion_f, direccion_c);
                        if (backtracking(piezas_restantes - 1))
                            return true;
                        deshacerMovimiento(fila, columna, direccion_f, direccion_c);
                    }
                }
            }
        }
    }
    return false;
}
```

1. Diseño del algoritmo

1. Inicialización:

● Comenzamos creando un tablero de dimensión 8x8 donde ‘O’ son las casillas ocupadas ‘ ’ los espacios libres y ‘X’ las casillas en las que no podemos ir.

2. Exploración del laberinto:

● Recorremos todas las filas y columnas del laberinto, si una casilla es diferente a ‘O` pasamos a la siguiente casilla.

● Para cada casilla ‘O’ comprobamos si hay movimientos válidos, si hay un movimiento válido lo aplicamos (aplicar el movimiento es comer una ficha, es decir avanzar de 2 casillas en cualquier dirección que no sea diagonal y eliminar la ficha sobre la que pasamos, nunca podemos movernos si no pasamos por encima de una ficha o si la casilla dónde queremos ir no esté vacía.), luego usamos recursividad y si hemos encontrado solución devolvemos true, si no deshacemos el movimiento.

3. Exploración Recursiva:

● Como he comentado antes, si encontramos un movimiento válido utilizamos recursividad, para que a partir de ese movimiento se prueben todos los demás movimientos posibles, hasta que quede una sola ficha y esta esté en el centro, si solo queda una ficha y esta no se encuentra en el medio la función devolverá false.

4. Backtracking:

● Después de que exploremos todos los movimientos posibles desde una posición del tablero si no encontramos solución, deshacemos el movimiento.

5. Repetir el Proceso:

● Continuar el proceso de exploración y backtracking hasta que encontremos una solución en la que la última ficha esté en el medio del tablero.

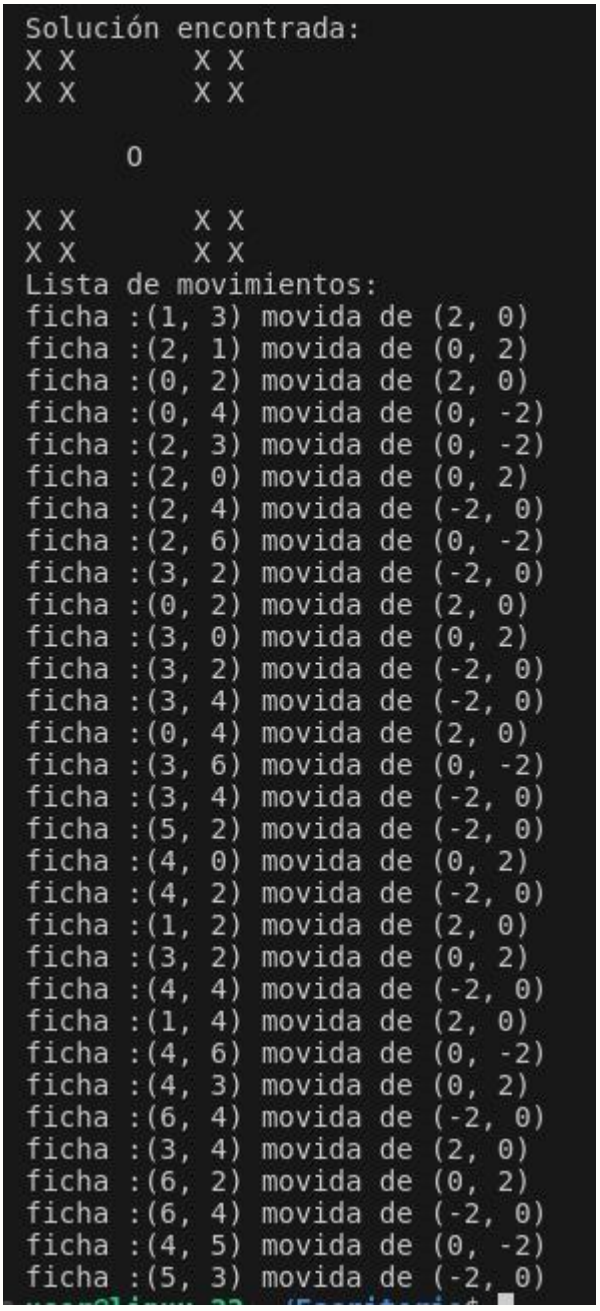
6. Finalización:

● Cuando encontramos una solución devolvemos true, por lo tanto cuando eso ocurre la ejecución de la función termina.

● Devolución de la lista de movimientos: cada vez que aplicamos un movimiento, añadimos a una cola ese movimiento (dirección movimiento de filas y columnas), y cuando deshacemos un movimiento lo eliminamos de la cola, por lo que cuando encontramos la solución vamos a devolver todos los movimientos realizados hasta llegar a ese punto, (posición de la ficha inicial y dirección del movimiento).

2. Ejemplo paso a paso

-Recorremos la matriz desde (0,0) hasta (7,7)
-Empezamos en (0,0) hasta que no encontramos una 'O' seguimos
    La primera va a ser en la posición (0,2)
-Comprobamos los movimientos posibles
    No hay movimientos posibles
-Seguimos hasta encontrar una 'O' con movimientos posibles
    La primera va a ser en la posición (1,3) y el movimiento para abajo (+2,0)
-Utilizamos recursividad
-Hacemos lo mismo siempre hasta que quede una 'O' en el centro
    Si la 'O' no queda en el centro deshacemos el movimiento realizado
    Si había otros movimientos posibles los aplicamos
    Si no deshacemos otra vez el movimiento y así hasta encontrar la solución
Al final el primer recorrido correcto que va a encontrar va a ser este:



Esta no es la única solución posible, pero es la primera que encuentra.

Problema 4 con Backtracking

Para resolver el problema, el algoritmo Backtracking implementado, para cada posición de la matriz, comprueba si es solución.. Si no lo es, se agrega y se realiza una llamada recursiva para cada posición (x+1, x-1, y+1, y-1). El proceso continúa de manera recursiva hasta que se haya encontrado la salida del laberinto.

```cpp
void explorarLaberinto(int x, int y, int n, vector<vector<bool>> &laberinto, vector<vector<bool>> &visitadas, vector<vector<int>> &pasos, int numPaso, vector<vector<vector<int>>> &soluciones) {
    if (x == n - 1 && y == n - 1) {
        visitadas[x][y] = true;
        pasos[x][y] = numPaso; // Marcando la última posición en la solución actual.
        soluciones.push_back(pasos); // Guardar la solución encontrada.
        visitadas[x][y] = false;
        pasos[x][y] = 0; // Resetear el paso en la matriz de pasos.
        return;
    }

    if (sePuedeVisitar(x, y, n, laberinto, visitadas)) {
        visitadas[x][y] = true;
        pasos[x][y] = numPaso;

        // Explorar todas las direcciones posibles
        explorarLaberinto(x + 1, y, n, laberinto, visitadas, pasos, numPaso + 1, soluciones);
        explorarLaberinto(x, y + 1, n, laberinto, visitadas, pasos, numPaso + 1, soluciones);
        if (x > 0) explorarLaberinto(x - 1, y, n, laberinto, visitadas, pasos, numPaso + 1, soluciones);
        if (y > 0) explorarLaberinto(x, y - 1, n, laberinto, visitadas, pasos, numPaso + 1, soluciones);

        visitadas[x][y] = false; // Desmarcar para backtracking
        pasos[x][y] = 0; // Resetear el paso en la matriz de pasos.
    }
}

bool resolverLaberinto(vector<vector<bool>> &laberinto) {
    int n = laberinto.size();
    vector<vector<bool>> visitadas(n, vector<bool>(n, false));
    vector<vector<int>> pasos(n, vector<int>(n, 0));
    vector<vector<vector<int>>> soluciones;

    explorarLaberinto(0, 0, n, laberinto, visitadas, pasos, 1, soluciones);

    if (soluciones.empty()) {
        cout << "No se encontró ninguna solución." << endl;
        return false;
    }

    cout << "Caminos encontrados:" << endl;
    int numSoluciones = 1;
    for (auto sol : soluciones) {
        cout << "Solución " << numSoluciones++ << ":" << endl;
        for (auto fil : sol) {
            for (auto col : fil) {
                cout << col << "\t";
            }
            cout << endl;
        }
        cout << endl;
    }
    return true;
}
```

Restricciones explícitas:

- *El laberinto está definido por n x n celdas, por lo que no podemos acceder a ninguna celda fuera de este límite, es decir, 0<=x<n; 0<=y<n.*
- *Solo se puede generar camino por celdas transitables, es decir, las que están marcadas como 'true', las que están marcadas como 'false' son celdas no transitables.*
- *Una celda visitada no se puede revisar en un mismo camino, solo se pueden visitar las celdas marcadas como no visitadas*

Restricciones implícitas:

- *El punto de entrada del laberinto siempre es el (0,0) y el de salida siempre es (n-1, n-1)*

- *No puede haber ciclos.*
- *Cada movimiento debe ser a una celda adyacente, sin movimientos en diagonal.*

Funcionamiento del algoritmo:

```
int main() {
    vector<vector<bool>> laberinto = {
       {1, 1, 0},
       {1, 0, 1},
       {1, 1, 1}
    };

    if (!resolverLaberinto(laberinto)) {
        cout << "No existe un camino de inicio a fin." << endl;
    }

    return 0;
}
```

- *Empieza el problema llamado a resolverLaberinto para (laberinto).*

Llamada a resolverLaberinto:

- *Se crea una variable n con el tamaño del laberinto, y se crean las matrices visitadas y soluciones y a continuación comienza la exploración del laberinto con la llamada a la función explorarLaberinto(0, 0, n, laberinto, visitadas, pasos, 1, soluciones) con la posición de inicio(0, 0).*
- *Se comprueba si es solución, como la posición (0, 0) no es solución comprobamos si se puede visitar. Como se puede visitar se pone visitadas[x][y] = true y se pone pasos[x][y] a numPaso. Siguiente se realiza una llamada recursiva para todas las posiciones siguientes posibles, [x+1][y], [x-1][y], [x][y+1], [x][y-1], sin embargo, como (x == 0) y (y == 0), no se realiza la llamada para [x-1][y], [x][y-1].*

Primera Llamada Recursiva:

- *Se realiza la llamada a la funcion explorarLaberinto para la posición [1][0].*
- *Como la posición no es la salida, comprobamos si se puede visitar.*
- *Como consecuencia de si poder visitar la casilla ya que la funcione sePuedeVisitar devuelve true, actualizamos visitadas[1][0] y pasos[x][y] a 2.*

Segunda Llamada Recursiva:

- *Se realiza la llamada reursiva a la función explorarLaberinto siguiendo la primera llamada recursiva.*
- *Se realiza la llamada a la funcion explorarLaberinto para la posición [2][0].*
- *Como la posición no es la salida, comprobamos si se puede visitar.*

- *Como consecuencia de si poder visitar la casilla ya que la funcione sePuedeVisitar devuelve true, actualizamos visitadas[2][0] y pasos[2][0] a 3.*

<u>Tercera Llamada Recursiva:</u>

- *Se realiza la llamada reursiva a la función explorarLaberinto siguiendo la segunda llamada recursiva.*
- *Como no se puede realizar la llamada a la funcion explorarLaberinto para la posición [3][0] realiza la llamada a la funcion explorarLaberinto para la posición [2][1].*
- *Como la posición no es la salida, comprobamos si se puede visitar.*
- *Como consecuencia de si poder visitar la casilla ya que la funcione sePuedeVisitar devuelve true, actualizamos visitadas[2][1] y pasos[2][1] a 4.*

<u>Cuarta Llamada Recursiva:</u>

- *Se realiza la llamada reursiva a la función explorarLaberinto siguiendo la tercera llamada recursiva.*
- *Como no se puede realizar la llamada a la funcion explorarLaberinto para la posición [3][1] realiza la llamada a la funcion explorarLaberinto para la posición [2][2].*
- *Como la posición es la de salida, actualizamos visitadas[2][2] a true y pasos[2][2] al número de pasos que sería en este caso 5. A continuación guardamos en soluciones la matriz pasos y finalmente reseteamos pasos y actualizamos visitadas[2][2] a false.*
- *Se hace un return para acabar con la llamada recursiva.*

<u>Quinta Llamada Recursiva:</u>

- *Se realiza la llamada reursiva a la función explorarLaberinto siguiendo la tercera llamada recursiva.*
- *Se realiza la llamada a la funcion explorarLaberinto para la posición [1][1].*
- *Como la posición no se puede visitar se acaba la llamada recursiva.*

<u>Sexta Llamada Recursiva:</u>

- *Se realiza la llamada reursiva a la función explorarLaberinto siguiendo la tercera llamada recursiva.*
- *Se realiza la llamada a la funcion explorarLaberinto para la posición [2][0].*
- *Como la posición no se puede visitar porque visitadas[2][0] == true, nos salimos de la llamada recursiva.*
- *Saliendonos de esta llamada recursiva, se actualiza visitadas[2][1] a false y pasos[2][1] a 0.*

<u>Séptima Llamada Recursiva:</u>

- *Se realiza la llamada reursiva a la función explorarLaberinto siguiendo la segunda llamada recursiva.*
- *Se realiza la llamada a la funcion explorarLaberinto para la posición [1][0].*
- *Como la posición no se puede visitar porque visitadas[1][0] == true, nos salimos de la llamada recursiva.*
- *Saliendonos de esta llamada recursiva, se actualiza visitadas[1][0] a false y pasos[1][0] a 0.*

Octava Llamada Recursiva:

- *Se realiza la llamada reursiva a la función explorarLaberinto siguiendo la segunda llamada recursiva.*
- *Se realiza la llamada a la funcion explorarLaberinto para la posición [1][1].*
- *Como la posición no se puede visitar se acaba la llamada recursiva.*

Octava Llamada Recursiva:

- *Se realiza la llamada reursiva a la función explorarLaberinto siguiendo la segunda llamada recursiva.*
- *Como no se puede realizar la llamada a la funcion explorarLaberinto para la posición [0][0] realiza la llamada a la funcion explorarLaberinto para la posición [0][0].*
- *Como la posición no se puede visitar porque visitadas[0][0] == true, nos salimos de la llamada recursiva.*
- *Saliendonos de esta llamada recursiva, se actualiza visitadas[0][0] a false y pasos[0][0] a 0.*

Novena Llamada Recursiva:

- *Se realiza la llamada a la funcion explorarLaberinto para la posición [0][1].*
- *Como la posición no es la salida, comprobamos si se puede visitar.*
- *Como consecuencia de si poder visitar la casilla ya que la funcione sePuedeVisitar devuelve true, actualizamos visitadas[0][1] y pasos[0][1] a 2.*

Décima Llamada Recursiva:

- *Se realiza la llamada reursiva a la función explorarLaberinto siguiendo la novena llamada recursiva.*
- *Se realiza la llamada a la funcion explorarLaberinto para la posición [1][1], sin embargo al no poder visitarse se acaba la llamada.*

Onceava Llamada Recursiva:

- *Se realiza la llamada reursiva a la función explorarLaberinto siguiendo la novena llamada recursiva.*
- *Se realiza la llamada a la funcion explorarLaberinto para la posición [0][2], sin embargo al no poder visitarse se acaba la llamada y se actualiza visitadas[0][1] a false.*

Al no encontrar ninguna otra solución nos vamos al final del problema.

FINAL

Volviendo a la función resolverLaberinto se comprueba si soluciones está vacía, al no estarlo se hace un cout "Caminos encontrados:", se establece numSoluciones a 1 y se imprime cada posicion de soluciones imprimiendo así la matriz con el camino.

Problema 5 con Backtracking

Para encontrar la solución más óptima para recorrer un laberinto el algoritmo Backtracking implementado va comprobando si la posición es la solución. Si lo es y además se trata de un camino más corto que la actual mejor, actualiza la actualmente mejor. Si no lo es, pasa a generar los posibles hijos de dicha posición los cuales son, derecha, izquierda, arriba y abajo siempre que sean casillas transitables.

```
vector<pair<int,int>> mejor;
int n = 5;
int finFil = n-1;
int finCol = n-1;

struct nodo{
    int fil;
    int col;
    vector<pair<int,int>> secuenciaSeguida;
};

list<nodo> frontera2;
```

```
void backTracking3(vector<vector<bool>> matriz){

    if (!frontera2.empty()){
        nodo nodoAct = frontera2.front();
        frontera2.pop_front();

        if ( (nodoAct.fil == finFil && nodoAct.col == finCol) && (nodoAct.secuenciaSeguida.size() <= mejor.size() || mejor.empty() ) ){
            mejor = nodoAct.secuenciaSeguida;
        }
        else
        {
        //generamos los posibles caminos si los hay desde dicho nodo y pusheamos a la frontera.
        for (int i = -1 ; i <= 1 ; i++){
            if (i != 0){
                if (nodoAct.fil+i >= 0 && nodoAct.fil+i < n ){
                    if (matriz[nodoAct.fil+i][nodoAct.col] && (mejor.empty() || nodoAct.secuenciaSeguida.size() < mejor.size())){
                        nodo nodoHijo;
                        nodoHijo.fil = nodoAct.fil+i;
                        nodoHijo.col = nodoAct.col;
                        nodoHijo.secuenciaSeguida = nodoAct.secuenciaSeguida;
                        nodoHijo.secuenciaSeguida.push_back({nodoHijo.fil,nodoHijo.col});

                        frontera2.push_back(nodoHijo);
                    }
                }

                if (nodoAct.col+i >= 0 && nodoAct.col+i < n ){
                    if (matriz[nodoAct.fil][nodoAct.col+i] && (mejor.empty() || nodoAct.secuenciaSeguida.size() < mejor.size())){
                        nodo nodoHijo;
                        nodoHijo.fil = nodoAct.fil;
                        nodoHijo.col = nodoAct.col+i;
                        nodoHijo.secuenciaSeguida = nodoAct.secuenciaSeguida;
                        nodoHijo.secuenciaSeguida.push_back({nodoHijo.fil,nodoHijo.col});
                        frontera2.push_back(nodoHijo);
                    }
                }
            }
        }

            if (!frontera2.empty()){
                backTracking3(matriz);
            }
        }
    }
}
```

Restricciones explícitas:

● *El laberinto está definido por n x n celdas, por lo que no podemos acceder a ninguna celda fuera de este límite, es decir, 0<=x<n; 0<=y<n.*
● *Solo se puede generar camino por celdas transitables, es decir, las que están marcadas como 'true', las que están marcadas como 'false' son celdas no transitables.*

Restricciones implícitas:

● *El punto de entrada del laberinto siempre es el (0,0) y el de salida siempre es (n-1, n-1).*
● *Cada movimiento debe ser a una celda adyacente, sin movimientos en diagonal.*

Funcionamiento del algoritmo:

```
int main(){
    // Matriz del laberinto
    //vector<vector<bool>> laberinto = {
    //    {true, true, true, false,true},
    //    {false,false,true, false,true},
    //    {true, true, true, true, true},
    //    {true, false,false,false,false},
    //    {true, true, true, true, true}
    //};

     //// Matriz del laberinto
     //vector<vector<bool>> laberinto = {
     //    {true, true, true, true ,true},
     //    {false,false,false, false,true},
     //    {true, true, true, true, true},
     //    {true, false,false,false,false},
     //    {true, true, true, true, true}
     //};

     // Matriz del laberinto
     vector<vector<bool>> laberinto = {
         {true, true, true, true  ,true},
         {true,false,true, false, true},
         {true, true, true, true  ,true},
         {false, false,true,false ,true},
         {true, true, true, true  ,true},
     };

    nodo nodoOrig;
    nodoOrig.fil = 0;
    nodoOrig.col = 0;
    nodoOrig.secuenciaSeguida.push_back({0,0});

    frontera2.push_back(nodoOrig);
    backTracking3(laberinto);

    cout << "Laberinto:" << endl;
    for (int i = 0; i < n; ++i) {
        for (int j = 0; j < n; ++j) {
            cout << (laberinto[i][j] ? "1" : "0") << " ";
        }
        cout << endl;
    }
```

```
    vector<vector<int>> visualizaMatriz(n,vector<int>(n));

    for (int i = 0 ; i < n; i++){
        for (int j = 0 ; j < n ; j++){
            visualizaMatriz[i][j] = static_cast<int>(laberinto[i][j]);
        }
    }

    for (int i = 0 ; i < mejor.size() ; i++){
        visualizaMatriz[mejor[i].first][mejor[i].second] = 2;
    }

    cout << "Visualiza:" << endl;
    for (int i = 0; i < n; ++i) {
        for (int j = 0; j < n; ++j) {
            cout << visualizaMatriz[i][j] << " ";
        }
        cout << endl;
    }

    cout << "la mejor ruta es: " << endl;
    for (int i = 0 ; i < mejor.size() ; i++){
        cout << " (" << mejor[i].first << "," << mejor[i].second << ") ";
    }
}
```

- *Empieza el problema creando una matriz de tipo bool con la forma de un laberinto*
- *Inicializamos debidamente el nodo origen que por el enunciado sabemos que se trata del (0,0). Lo añadimos a la frontera*

  Llamamos al método que lo resolverá pòr backTracking:

- *Mientras que la frontera no esté vacía se va a estar ejecutando nuestro programa*
- *Sacamos el nodo que corresponde al frente de nuestra lista de nodos frontera*
- *Eliminamos dicho nodo de la lista*
- *Comprobamos si es solución y además si esta solución tiene mejor tamaño que la actual solución denominada mejor. En el caso de que cumpla estas dos condiciones se actualiza el valor de la secuencia mejor al de la secuencia que acabamos de encontrar.*
- *En el caso de que no se trate de una solución pasamos a generar los hijos del nodo actual.*
- *Con un bucle for que va desde -1 a 1 generamos el nodo adyacente al actual en los ejes x e y, como ya hemos especificado antes nuestro agente no podrá desplazarse diagonalmente.*
- *Si dicho hijo se encuentra en una posición correcta dentro de los límites se procede a comprobar si su secuencia aún tiene un tamaño válido. Este comportamiento de poda de nodos nos permite eliminar múltiples nodos de la cola puesto que no se añadirá aquel que, sin haber llegado aún a la solución, ya arrastra una secuencia más extensa que nuestra propia solución mejor actual. Es por esto que a dicho nodo no se le dará la oportunidad de ser estudiado y por tanto no pasará a la frontera.*

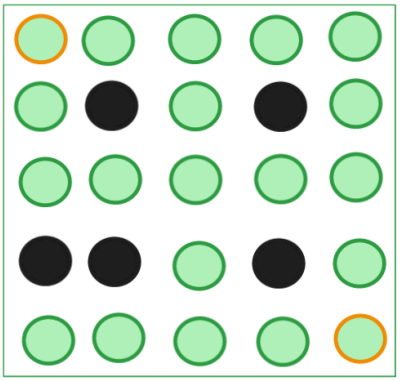
  Llamada Recursiva:
- *Como pretendemos obtener la mejor de las soluciones y no una solución cualquiera, se iniciará una llamada recursiva al método cada vez que termine de analizar un nodo y la frontera no esté vacía, es decir, queden nodos por analizar. Cuando ya se haya encontrado una solución óptima, los nodos de la*

frontera no generarán hijos al tener una longitud de secuencia inválida y se vaciará la cola rápidamente.

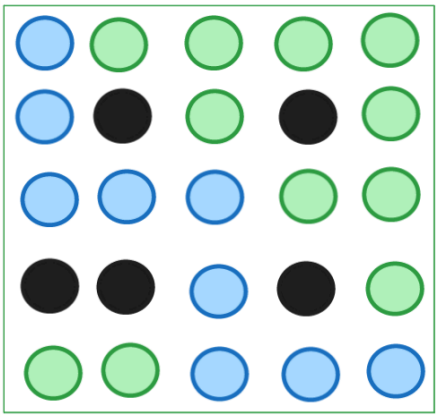
Tras la ejecución de los pasos explicados anteriormente se mostraría la siguiente salida:

```
jorge@smart-refrigerator:~/Escritorio/Algoritmica/ALGP4_2024$ ./problema5
Laberinto:
1 1 1 1 1
1 0 1 0 1
1 1 1 1 1
0 0 1 0 1
1 1 1 1 1
Visualiza:
2 1 1 1 1
2 0 1 0 1
2 2 2 1 1
0 0 2 0 1
1 1 2 2 2
la mejor ruta es:
 (0,0)  (1,0)  (2,0)  (2,1)  (2,2)  (3,2)  (4,2)  (4,3)  (4,4) jorge@smart-refri
4$
```

Para que sea más fácil de entender tenemos este laberinto: (con pocas restricciones para que se observe que toma el camino más eficiente pese a tener muchas posibilidades)



Y trás aplicar nuestro algoritmo nos presenta la siguiente solución: (hay varias soluciones óptimas, esta es una)



La secuencia más óptima es por tanto la siguiente:

```
la mejor ruta es:
 (0,0)  (1,0)  (2,0)  (2,1)  (2,2)  (3,2)  (4,2)  (4,3)  (4,4)
```