Universidad de Granada

ALGORÍTMICA

Práctica 4

Práctica 4 - Algoritmos de exploración de grafos

INTEGRANTES
GRUPO B3

Miguel Rodríguez Ayllón
Alberto Cámara Ortiz
Cristóbal Pérez Simón
José Vera Castillo
Pablo Rejón Camacho

ÍNDICE

Preámbulo

Nuestro proyecto cuenta con un makefile para que resulte más fácil a la hora de ejecutar los códigos proporcionados.

Cada ejercicio cuenta con una regla específica que muestra la ejecución del algoritmo y se muestra el resultado de la ejecución.

- make ej1 *(./Ej1.bin)*
- maje ej2 *(./Ej2.bin)*
- make ej3 *(./Ej3.bin)*
- make ej4 *(./Ej4.bin)*
- make ej5 *(./Ej5.bin)*

Si es necesario probar con unos valores a elegir por el profesor, recomendamos modificar la regla de ese ejercicio, o cambiar el argumento de la ejecución a mano. De todas formas, también podría ejecutarse a mano con los argumentos necesarios ya que no eliminamos los archivos .bin.

Problema 1

1.1. Diseño del algoritmo

Solución parcial: Se define por el conjunto de parejas ya formadas hasta el momento. Estas parejas se almacenan en un vector de pares (vector< pair<int, int>>), donde cada par representa dos estudiantes emparejados.

Restricciones explícitas:

- **n** debe ser un número par, ya que todos los estudiantes deben ser emparejados.
- Los valores de la matriz “preferences” deben ser no negativos, ya que representan niveles de preferencia.
- Cada estudiante puede estar en una sola pareja a la vez.

Restricciones implícitas:

- El algoritmo debe buscar maximizar la suma global de los productos de las preferencias mutuas, no sólo las preferencias individuales.
- La matriz preferences es cuadrada de tamaño n x n.

Formalización:

Entrada: Una matriz p de tamaño n x n donde p[i][j] representa la preferencia del estudiante i hacia el estudiante j.

Salida: Un conjunto de parejas y la suma máxima de los valores de emparejamiento, donde el valor de emparejamiento entre dos estudiantes i y j es dado por p[i][j] * p[j][i].

```
function Emparejar(preferencias, n, emparejados, ParejasActuales,
sumaActual, indice, sumaMaxima)
    resul.sumaMaxima <- sumaMaxima

    if indice == n then
        if sumaActual > sumaMaxima then
            resul.sumaMaxima <- sumaActual
            resul.mejoresParejas <- copia de ParejasActuales
        end if
        return resul
    end if

    bestResul <- copia de resul

    if emparejados[indice] then
        return Emparejar(preferencias, n, emparejados,
ParejasActuales, sumaActual, indice + 1, sumaMaxima)
    end if
```

```
    for j desde indice + 1 hasta n - 1 do
        if not emparejados[j] then
            emparejados[indice] <- true
            emparejados[j] <- true
            ParejasActuales.agregar((indice, j))
            valorPareja <- preferencias[indice][j] *
preferencias[j][indice]

            currentResul <- Emparejar(preferencias, n, emparejados,
ParejasActuales, sumaActual + valorPareja, indice + 1,
bestResul.sumaMaxima)

            if currentResul.sumaMaxima > bestResul.sumaMaxima then
                bestResul <- currentResul
            end if

            emparejados[indice] <- false
            emparejados[j] <- false
            ParejasActuales.eliminarUltimo()
        end if
    end for

    return bestResul
end function
```

1.2. Funcionamiento para una instancia pequeña

Para la instancia dada con la matriz de preferencias:

{	*0*	*3*	*7*	*1*	*}*
{	*1*	*0*	*5*	*6*	*}*
{	*4*	*5*	*0*	*9*	*}*
{	*2*	*4*	*5*	*0*	*}*

Proceso paso a paso usando el algoritmo de backtracking que diseñamos:

1. **Inicio (Índice = 0):**
   El índice es 0 y el primer estudiante no está emparejado.

2. **Emparejar estudiante 0 con 1:**
   Se marcan como emparejados 0 y 1.
   Parejas actuales: (0, 1)
   Suma actual: 3 * 1 = 3
   Se llama a la recursión con el siguiente índice (1).

3. Paso recursivo (Índice = 1):
El estudiante 1 ya está emparejado, por lo que se pasa al siguiente índice (2).

4. Emparejar estudiante 2 con 3:
Se marcan como emparejados 2 y 3.
Parejas actuales: (0, 1), (2, 3)
Suma actual: 3 + (9 * 5) = 48
Se llama a la recursión con el siguiente índice (3).

5. Paso recursivo (Índice = 3):
El estudiante 3 ya está emparejado, se pasa al siguiente índice (4), que es el final.
Retornamos con la máxima suma alcanzada hasta el momento (48).

6. Deshacer emparejamiento de 2 con 3:
Se deshacen las parejas (2, 3) y se vuelve a desmarcar como emparejados 2 y 3.

7. Emparejar estudiante 0 con 2:
Se marcan como emparejados 0 y 2.
Parejas actuales: (0, 2)
Suma actual: 7 * 4 = 28
Se llama a la recursión con el siguiente índice (1).

8. Paso recursivo (Índice = 1):
Se empareja 1 con 3.
Parejas actuales: (0, 2), (1, 3)
Suma actual: 28 + (5 * 6) = 58
Retornamos con una suma máxima nueva de 58.

9. Deshacer emparejamientos:
Se deshacen las parejas (0, 2) y (1, 3).

10. Emparejar estudiante 0 con 3:
Se marcan como emparejados 0 y 3.
Parejas actuales: (0, 3)
Suma actual: 1 * 2 = 2
Se llama a la recursión con el siguiente índice (1).

11. Paso recursivo (Índice = 1):
Se empareja 1 con 2.
Parejas actuales: (0, 3), (1, 2)
Suma actual: 2 + (5 * 5) = 27
Retornamos con una suma máxima de 58 (previa).

Resultados Finales:

Las mejores parejas formadas son (0, 2) y (1, 3) con una suma máxima de 58.

1.3 Implementación

```cpp
struct Resultado {
    int sumaMaxima;
    vector< pair<int, int> > mejoresParejas;
};


// Función para intentar emparejar los estudiantes
Resultado Emparejar( vector< vector<int> > preferencias ,int n, vector<bool>& emparejados, vector< pair<int, int>>& ParejasActuales, int sumaActual, int indice, int sumaMaxima) {
    Resultado resul;
    resul.sumaMaxima = sumaMaxima;
    if (indice == n) {
        if (sumaActual > sumaMaxima) {
            resul.sumaMaxima = sumaActual;
            resul.mejoresParejas = ParejasActuales;
        }
        return  resul;
    }

    Resultado bestResul = resul;

    if (emparejados[indice]) {
        return Emparejar(preferencias,n, emparejados, ParejasActuales, sumaActual, indice + 1,sumaMaxima);

    }

    for (int j = indice + 1; j < n; j++) {
        if (!emparejados[j]) {
            emparejados[indice] = true;
            emparejados[j] = true;
            ParejasActuales.emplace_back(indice, j);
            int valorPareja = preferencias[indice][j] * preferencias[j][indice];

            Resultado currentResul = Emparejar(preferencias,n, emparejados, ParejasActuales, sumaActual + valorPareja, indice + 1, bestResul.sumaMaxima);

            if ( currentResul.sumaMaxima > bestResul.sumaMaxima )
                bestResul = currentResul;

            // Deshacemos cambios para la próxima iteración
            emparejados[indice] = false;
            emparejados[j] = false;
            ParejasActuales.pop_back();
        }
    }

    return bestResul;
}
```

Problema 2

2.1. Diseño del algoritmo

Solución parcial: Tupla de valores $x = (x_1, x_2, x_3, \dots, x_n)$ donde x_i representa un comensal en un posición de la mesa, siendo $3 <= n$. Habrá tantos valores de x_i como comensales, lo único que podría variar entre soluciones es el orden de los comensales, cuya solución se le dará en la *'función de poda'*.

Restricciones explícitas:

- Siempre se va a cumplir que $x_1 = 0$.
- $x_i \neq x_j$.

Restricciones implícitas:

- Se debe mantener un vector de usados para no poder insertar el mismo comensal en dos posiciones diferentes, que modificará su valor a true cuando el comensal se encuentre en la asignación.

Función de poda:
No hemos implementado ninguna función de poda. *¿Por qué?* Debido a que la implementación del algoritmo se centra en anular todas las soluciones equivalentes. Al no modificar el valor $x_1 = 0$, no te permite evaluar soluciones que darán el mismo resultado.

Ej: $\{0, 1, 2, 3\} = 40$ || $\{2, 3, 0, 1\} = 40$.

Formalización:
C → Matriz de conveniencia.
A → Vector con los valores de la asignación actual.
S → Vector en el que se almacena la mejor asignación.
T → Entero que almacena el mejor valor de conveniencia.

```
function calcularConveniencia(A, C)
     total:= 0;
     n:= longitud(A)
     for i=0 until n-1, do
          izquierda:= A[(i-1+n)%n]
          derecha:= A[(i+1)%n]
          total += C[i][izquierda]
          total += C[i][derecha]
     end for
     return total
end function
```

```
function backtracking(A, C, usado, n, T, S)

    if longitud(A) = n entonces
        conveniencia_actual:= calcularConveniencia(A, C)
        if conveniencia_actual > T then
            T:= conveniencia_actual
            S:= A
            return
        end if
    end if

    for i=1 until n-1, do
        if !usado[i] then
            usado[i]:= true
            A.push(i);

            backtracking(A, C, usado, n, T, S)

            A.pop()
            usado[i]:= false;
        end if
    end for

end function
```

2.2. Funcionamiento para una instancia pequeña.

Vamos a coger una matriz de conveniencia de 4x4 con la cuál mostrar el funcionamiento de una pequeña instancia:

{ 0 5 4 6 }
{ 2 0 7 9 }
{ 6 2 0 3 }
{ 6 1 9 0 }

Nuestro algoritmo, probará todas las posibilidades posibles comprobando en cada una de las opciones si el nivel de conveniencia es mayor que el anteriormente calculado como "mejor valor de conveniencia". Esto lo consigue mediante llamadas recursivas dentro de un bucle for con tantas iteraciones como comensales haya. Usaremos un recorrido en profundidad del árbol. Vamos a explicar el funcionamiento de este algoritmo con esta instancia:

El algoritmo inicia con el valor 0 añadido dentro del vector de asignación actual, debido a que todos los comensales van a estar sentados en un sitio específico, sea el que sea.

El algoritmo en primer lugar añadirá todos los comensales en orden creciente $\{0, 1, 2, 3\}$, con las llamadas recursivas necesarias a backtracking y, en cada una de ellas, no cumpliéndose la condición del primer if entran en el bucle for añadiendo en cada iteración el valor correspondiente debido a que el vector usado se encuentra inicialmente con todas las

componentes a false. Una vez añadidos todos los valores a “asignación_actual”, se cumplirá la condición del primer if, que en una primera ejecución modificará el valor de mejor conveniencia, ya que se encuentra inicializado al mínimo entero. A partir de aquí, siempre que sea mayor el valor de “convenciencia_actual” que el almacenado en la variable entera “mejor_conveniencia”, actualizará el valor de esta variable al valor de conveniencia_actual. En este caso, en el que ya tenemos todos los valores en el vector con el orden {0,1,2,3}, se almacenará el valor **40** [(5+6) + (2+7) + (2+3) + (9+6)] en la variable “mejor_conveniencia”. Este valor se consigue cuando nos encontramos en la llamada a backtracking con un valor de $i = 3$ y se realizaría un return de la llamada. A continuación, se eliminaría el valor 3 del vector asignacion_actual y se pondría a false su valor en el vector de usados, dando por finalizada esta ejecución de la función backtracking ya que se cumple la condición del bucle for (i<n) y pasando el flujo a la finalización de la llamada a backtracking.

El flujo seguiría en el bucle for e iteración $i = 2$, en la que se eliminaría este valor del vector “asignacion_actual” y se modificaría a false su valor en el vector usado. El bucle pasaría a la iteración $i = 3$, que sería añadida a la asignación actual y se volvería hacer una llamada recursiva a backtracking, que añadiría el valor entero 2, quedando la asignación actual como {0,1,3,2}. En este lugar, el valor de la conveniencia sería de 39, por lo que no se modificaría el valor de mejor conveniencia. Se sacaría el entero 2 del vector, ya que al aumentar de iteración detectaría al 3 como usado y finalizará esa llamada a backtracking. Por consiguiente, se eliminaría el 3 y se finalizará esta llamada a backtracking. En este instante, tendríamos un vector asignacion_actual con {0,1}.

El flujo seguiría en el bucle for e iteración $i = 1$, ya con el return de la llamada a backtracking, es decir, ya se habrían analizado todas las posibilidades de sentar a los comensales que inician por {0,1}, por lo que se eliminaría el valor entero 1 de la asignación actual y aumentaría la iteración del bucle for. En este caso, se añadiría el entero con valor 2, ya que no está siendo usado y se volvería a llamar a backtracking, se volverían a analizar todas las posibilidades de sentar a los comensales con {0,2} y, en caso de que alguna posibilidad supere la mejor conveniencia almacenada, en este caso 40, modificaría su valor y el vector de mejor asignación. Este caso nos lo podemos encontrar con la asignación {0, 2, 1, 3}, cuyo valor de conveniencia es **41**, por lo que este valor se guardaría en “mejor_conveniencia”.

Tras analizar todas las posibilidades, se eliminaría el valor 2 y se aumentaría la iteración, que añadiría el valor a la asignación actual {0,3} y, de nuevo, se probarían todas las soluciones posibles y, si en alguna de ellas se consigue un valor de conveniencia mayor al que tenemos almacenado, se modificaría el valor y el valor de mejor asignación. Al ser un caso con una instancia pequeña, los únicos dos casos posibles serían {0,3,1,2} y {0,3,2,1}. Tras analizar ambos casos, **ninguno de los valores son mayores que 41**, por lo que no modificarán ni la mejor conveniencia ni la mejor asignación. Finalmente, cuando se haga un return a esa llamada a backtracking, el algoritmo finaliza ya que se encuentra en la última iteración del bucle for ($i = 3$)

Por lo tanto, el programa devolverá como mejor valor de conveniencia **41** y la mejor asignación como **{0, 2, 1, 3}**.

2.3 Implementación

```cpp
int calcularConvenienciaTotal(const vector<int>& asignacion, const vector<vector<int>>& conveniencia) {
    int total = 0;
    int n = asignacion.size();
    for (int i = 0; i < n; ++i) {
        int izquierda = asignacion[(i - 1 + n) % n];
        int derecha = asignacion[(i + 1) % n];
        total += conveniencia[asignacion[i]][izquierda];
        total += conveniencia[asignacion[i]][derecha];
    }
    return total;
}

void backtracking(vector<int>& asignacion_actual, vector<vector<int>>& conveniencia,
                  vector<bool>& usado, int n, int& mejor_conveniencia, vector<int>& mejor_asignacion) {

    if (asignacion_actual.size() == n) {
        int conveniencia_actual = calcularConvenienciaTotal(asignacion_actual, conveniencia);
        if (conveniencia_actual > mejor_conveniencia) {
            mejor_conveniencia = conveniencia_actual;
            mejor_asignacion = asignacion_actual;
        }
        return;
    }

    for (int i = 1; i < n; ++i) {
        if (!usado[i]) {
            usado[i] = true;
            asignacion_actual.push_back(i);

            backtracking(asignacion_actual, conveniencia, usado, n, mejor_conveniencia, mejor_asignacion);

            asignacion_actual.pop_back();
            usado[i] = false;
        }
    }
}
```

Problema 3

3.1. Diseño del algoritmo

Solución parcial: Viene dada en una tupla de valores $(x_1, x_2, ..., x_m)$ donde x_i representa un movimiento(posición inicial, final, dirección), $1 \leq m \leq 31$.

Restricciones explícitas:

- Las posiciones deben estar dentro del tablero.
- Se debe comer 1 ficha que esté en una casilla adyacente en cada movimiento.

Restricciones implícitas:

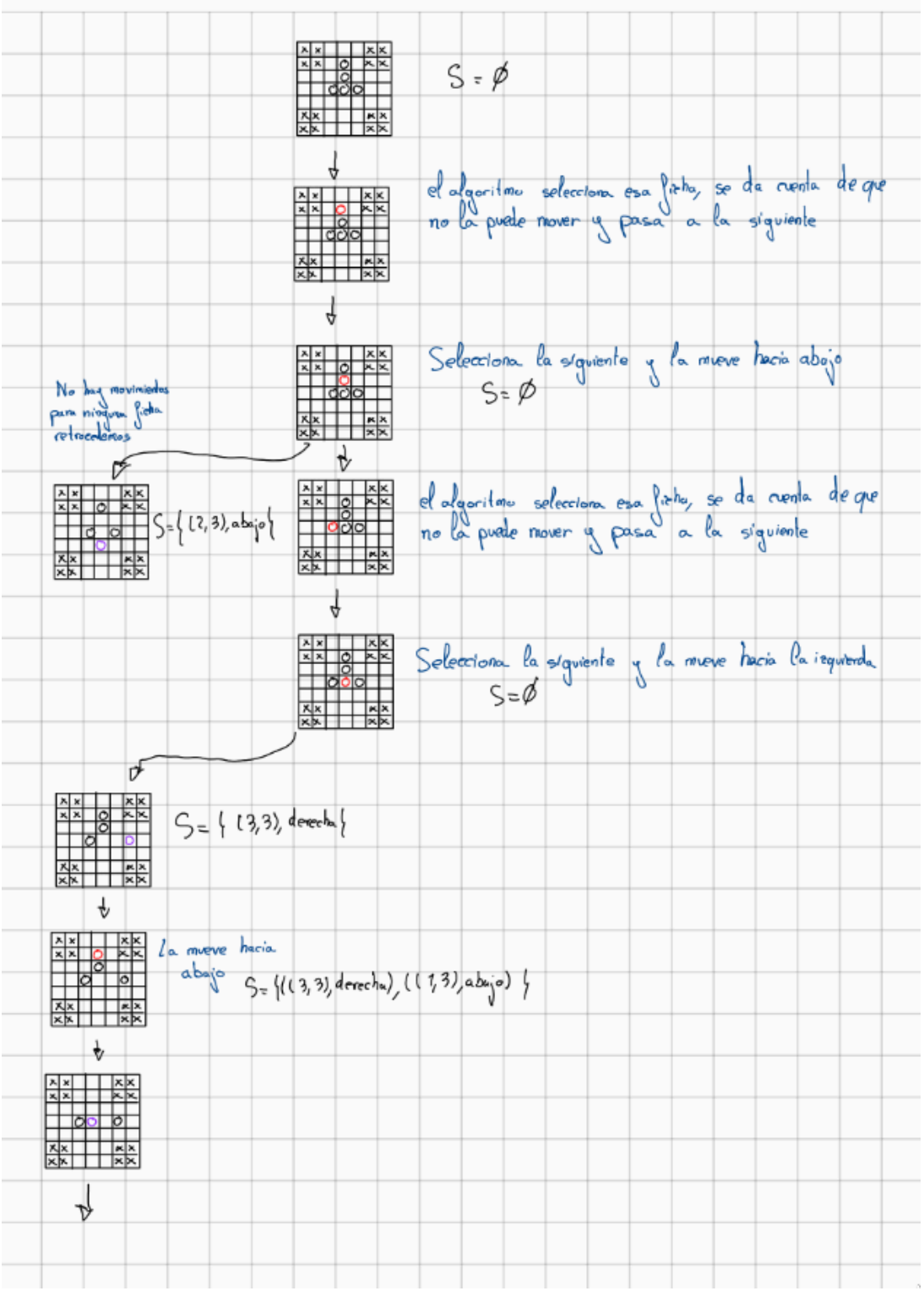
- Queda 1 sola bola en el centro del tablero o se puede hacer un movimiento válido.

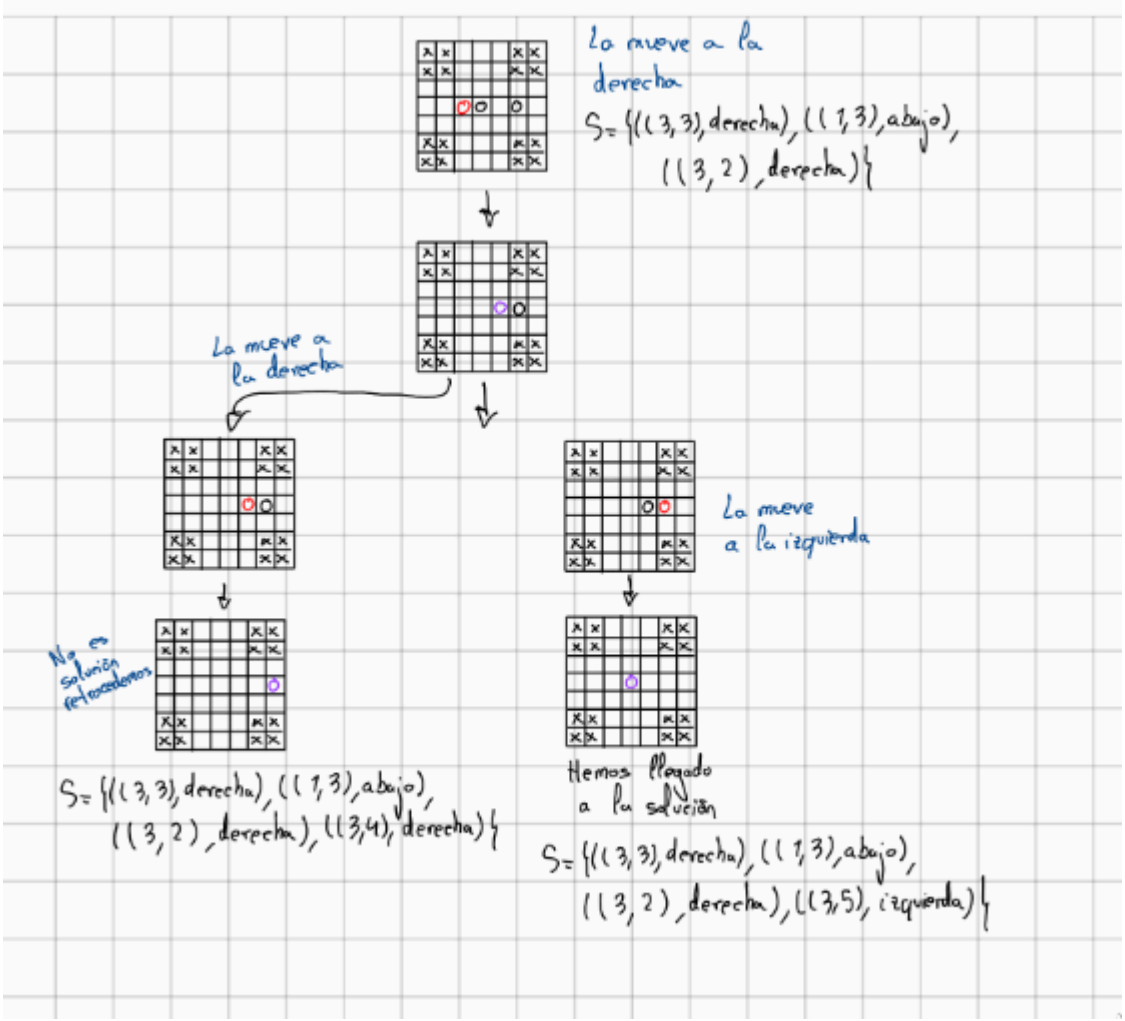
Formalización:
n → Tamaño del tablero.
estado → (libre, ocupada, no válida).
S → Solución.

```
function senku(tablero)
      S <- ∅
      sol <- false
      mov <- {izquierda, arriba, derecha, abajo}

      for i to n do
            for j to n do
                  for k to 4
                        if valido(i,j,mov[k],tablero)
                              otro = NuevaTabla(i,j, mov[k], tablero)
                              if solucion(otro) hacer
                                    sol = true
                                    S <- S U (i,j, mov[k])
                              end if
                                    else
                                    mini <- senku(otro)
                                    if |mini| != 0
                                          sol = true
                                          S <- S U (i,j, mov[k])
                                                S <- S U mini
                                    end if
                              end else
                        end if
                  end for
            end for
      end for

      if !S
            devolver ∅
      end if
      devolver S
end senku
```

3.2. Funcionamiento para una instancia pequeña

En rojo representamos la ficha que el algoritmo selecciona para intentar mover en las 4 direcciones. En morado como queda dicha ficha después de aplicar el movimiento.


S = ∅
el algoritmo selecciona esa ficha, se da cuenta de que no la puede mover y pasa a la siguiente
Selecciona la siguiente y la mueve hacia abajo
S= ∅
No hay movimientos para ninguna ficha retrocedemos
S={(2,3), abajo}
el algoritmo selecciona esa ficha, se da cuenta de que no la puede mover y pasa a la siguiente
Selecciona la siguiente y la mueve hacia la izquierda
S=∅
S= {(3,3), derecha}
La mueve hacia abajo
S= {((3,3), derecha), ((1,3), abajo)}


La mueve a la derecha
S= {((3,3), derecha), ((1,3), abajo), ((3,2), derecha)}
La mueve a la derecha
La mueve a la izquierda
No es solución retrocedemos
S= {((3,3), derecha), ((1,3), abajo), ((3,2), derecha), ((3,4), derecha)}
Hemos llegado a la solución
S= {((3,3), derecha), ((1,3), abajo), ((3,2), derecha), ((3,5), izquierda)}

3.3 Implementación

```
vector<accion> senku2(const vector<vector<estado>> &tablero){
    bool hemosMovido = false;
    bool encontradoSol = false;
    vector< accion > plan;

    for (int i = 0; i < TAM && !encontradoSol; i++)
      for (int j = 0; j < TAM && ! encontradoSol; j++)
        for (int k = 1; k <= 4; k++)
          if (esValido({i,j}, k, tablero) && !hemosMovido){
              vector<vector<estado>> otro = cambiaTablero({i,j}, k, tablero);
              if (esSolucion(otro)) {
                  hemosMovido = true;
                  plan.push_back({{i,j}, k});
                  encontradoSol = true;
              }
              else{
                  vector< accion > mini = senku2(otro);
                  if (!mini.empty()){
                      hemosMovido = true;
                      plan.push_back({{i,j}, k});
                      plan.insert(plan.end(), mini.begin(), mini.end());
                  }
              }
        }

    if (!hemosMovido){
        return vector< accion >();
    }

    return plan;
}
```

Problema 4

4.1. Diseño del algoritmo

Para realizar el diseño del algoritmo, tenemos que tener en cuenta una serie de restricciones, tanto explícitas como implícitas, pero antes vamos a tener en cuenta ciertos parámetros:

- **N:** Tamaño de la matriz
- **x_i:** Posición válida dentro del tablero
- **L:** Laberinto (de tipo booleano donde "1" indica que la casilla es transitable y "0" indica que no lo es)
- **S:** Matriz donde se almacenarán las casillas que componen la solución

Restricciones explícitas:

- El laberinto debe tener unas dimensiones finitas (matriz NxN).
- x_i será válida si y sólo si se cumple: $x_i \in \{L;\ 0 \leq i,j < N\ y\ L[i][j] = 1\}$
- Los únicos movimientos permitidos en el tablero son arriba, abajo, izquierda y derecha.
- Las casillas de entrada y salida del laberinto son (0,0) y (n-1, n-1) respectivamente.

Restricciones implícitas:

- Se debe mantener un registro de las celdas visitadas para evitar un ciclo infinito.
- Se establecen condiciones de parada tales como encontrar la salida del laberinto o haber explorado todas las posibles rutas.
- El algoritmo debe ser capaz de retroceder a estados anteriores cuando no hay más movimientos posibles desde un estado actual.

Solución parcial: Vendría dada en una tupla de valores (c_1, c_2, …., c_m) donde c_i representa una casilla de la solución ($0 \leq i \leq N$).

A continuación, describimos el diseño del algoritmo:

```
function CasillaValida(L, x, y)
     Devuelve si la casilla es válida o no (restr. explícitas)


function SolBacktracking(L, x, y, S)
     if(x = N-1 && y=N-1) then // Caso base (salida)
          return true
     end if

     if(CasillaValida(L, x, y)) then
          S[x][y] = 1     // La incluimos en la solución
          L[x][y] = 3     // Casilla ya transitada

          if(SolBacktracking(L, x + 1, y, S)||
             SolBacktracking(L, x, y + 1, S) ||
             SolBacktracking(L, x - 1, y, S) ||
             SolBacktracking(L, x, y - 1, S))
               return true
          S[x][y] = 0 // Solución no válida (vuelta atrás)
     end if
     return false
end fuction
```

4.2. Funcionamiento para una instancia pequeña

Definimos una instancia pequeña para la resolución del algoritmo;

1. La matriz será de 5x5 y tendrá la siguiente forma:

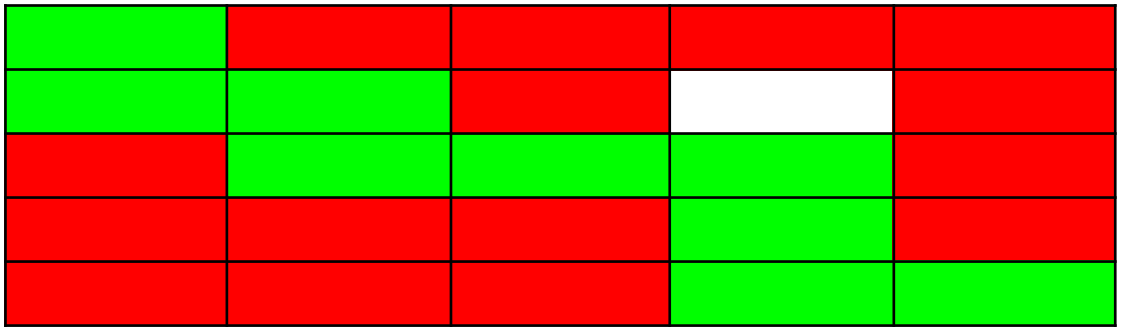
pos. inicial (0,0)				
			pos. prob	
				pos. final

donde las casillas rojas indican que no son transitables y las blancas sí.

Comienza la ejecución del programa. Se le pasa la posición 0,0 a la función de solución;

1. Comprobamos que la casilla actual es válida (lo es porque es transitable y está dentro de la matriz)
2. Añadimos dicha casilla a nuestra solución

3. Tratamos de movernos en alguna de las 4 posibles posiciones (la única válida es la de abajo), esto implica llamar recursivamente a la función añadiendo en cada caso las posiciones a nuestra solución
4. Si nos detenemos en el caso de haber llegado a la posición problemática (pos. prob), nos damos cuenta de que tenemos que retroceder porque no hay ninguna salida posible, por lo que esa casilla no forma parte de nuestra solución y tenemos que seguir buscando
5. Una vez encontrada la solución devolveremos el valor true. Como vemos, nuestra solución está formada por las siguiente casillas etiquetadas en verde:



Como vemos en este caso, el algoritmo da una solución, que no tiene por qué ser la mejor, pero en este caso es la única posible.

4.3 Implementación

```cpp
const int N = 5; // Tamaño del laberinto

// Función para comprobar si una celda es válida
bool CasillaValida(vector<vector<int>>& laberinto, int x, int y) {
    return (x >= 0 && x < N && y >= 0 && y < N && laberinto[x][y] == 1);
}

// Función solución usando Backtracking
bool SolBacktracking(vector<vector<int>>& laberinto, int x, int y, vector<vector<int>>& sol) {


    // CASO BASE: Si llegamos a la salida
    if (x == N - 1 && y == N - 1) {
        sol[x][y] = 1; // Marcamos la salida
        return true;
    }

    // Comprobamos que la celda actual es válida
    if (CasillaValida(laberinto, x, y)) {
        sol[x][y] = 1; // Dicha celda forma parte de la solución

        laberinto[x][y] = 3; // Marcamos la celda como visitada

        // Nos movemos en las 4 direcciones posibles
        if (SolBacktracking(laberinto, x + 1, y, sol) || SolBacktracking(laberinto, x, y + 1, sol) || SolBacktracking(laberinto, x - 1, y, sol) || SolBacktracking(laberinto, x, y - 1, sol))
            return true;

        sol[x][y] = 0; // Volvemos atrás si ninguna dirección es válida

    }

    return false;
}

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

    vector<vector<int>> solucion(N, vector<int>(N, 0)); // Inicializamos la matriz solución

    if (SolBacktracking(laberinto, 0, 0, solucion)) {
        cout << "Solucion encontrada:\n";
        for (int i = 0; i < N; ++i) {
            for (int j = 0; j < N; ++j)
                cout << solucion[i][j] << " ";
            cout << "\n";
        }
    }
    else {
        cout << "No hay camino posible.\n";
    }

    return 0;
}
```

Problema 5

5.1. Diseño del algoritmo

Para realizar el diseño del algoritmo, tenemos que tener en cuenta una serie de restricciones, tanto explícitas como implícitas, pero antes vamos a tener en cuenta ciertos parámetros:

- **N:** Tamaño de la matriz
- **x_i:** Posición válida dentro del tablero
- **L:** Laberinto (de tipo booleano donde "1" indica que la casilla es transitable y "0" indica que no lo es)

Restricciones explícitas

- El laberinto debe tener unas dimensiones finitas (matriz NxN).
- x_i será válida si y sólo si se cumple: $x_i \in \{L[i][j];\ 0 \leq i,j < N\ y\ L[i][j] = 1\}$
- Los únicos movimientos permitidos en el tablero son arriba, abajo, izquierda y derecha.
- Las casillas de entrada y salida del laberinto son (0,0) y (n-1, n-1).

Restricciones implícitas

- Se debe mantener un registro de las celdas visitadas para evitar un ciclo infinito
- Se establecen condiciones de parada tales como encontrar la salida del laberinto o haber explorado todas las posibles rutas
- El algoritmo debe ser capaz de retroceder a estados anteriores cuando no hay más movimientos posibles desde un estados actual

Solución parcial: vendría dada en una tupla de valores (c_1, c_2, …., c_m) donde c_i representa una casilla de la solución ($0 \leq i \leq N$)

```
Función SolBackTracking(laberinto, visitado, solucion, casilla actual,
min_dist, dist)
    Si casilla actual es la salida del laberinto(N,N) entonces
        Marcar la casilla como parte de la solución
      Si la distancia actual a la salida es menor que la distancia mínima
      hasta el momento entonces
            Actualizar la distancia mínima
            Actualizar la solución con la ruta actual

      Fin Si
    Fin Si

   Si la distancia no es mayor a la distancia mínima(Si no, se poda)
      Seguir explorando
```

```
            Marcar la casilla actual como visitada

            (Para las 4 direcciones)
            Si es posible moverse en esa direccion entonces
              Llamar recursivamente a SolBackTracking para la casilla a la que
              nos estemos moviendo
            Fin Si

            Desmarcar la casilla actual como visitada (Retroceder)
        Fin Si
Fin SolBactracking
```

5.2 Funcionamiento del algoritmo para una instancia pequeña

Definimos una instancia pequeña para la resolución del algoritmo;

pos. inicial (0,0)				
				pos. final

donde las casillas rojas indican que no son transitables y las blancas sí.

1. Desde la posición inicial nos desplazamos hacia abajo ya que es la única opción posible.
2. Una vez en (1, 0) volvemos a ir hacia abajo porque no hay otra opción.
3. En (2,0) se plantean dos opciones, ir hacia la derecha o seguir bajando
4. El algoritmo se lanza recursivamente en ambas direcciones y en ambos casos se halla un camino que lleva a la salida.
5. Las distancias se van actualizando para ambos caminos y se llega a la conclusión de que la distancia mínima y por ende, la más corta de los dos es la del camino de abajo.
6. Por tanto, se toma ese camino y se descarta el otro, llegando así a la mejor solución.

Como vemos en este caso, el algoritmo da la solución más corta(verde) y se ha descartado una solución más larga(amarillo).

5.3 Implementación

```cpp
// Función solución usando Backtracking
void SolBackTracking(vector<vector<int>> &laberinto, vector<vector<bool>> &visitado, vector<vector<int>> &solucion, int x, int y, int &min_dist, int dist)
{
    //Si se llega a la salida, actualizar la distancia mínima
    if (x == N - 1 && y == N - 1){
        solucion[x][y] = 1; // Marcamos la salida
        if (dist < min_dist) {
            min_dist = dist;
            // Actualizar la solucion
            for (int x = 0; x < N; ++x) {
                for (int y = 0; y < N; ++y) {
                    solucion[x][y] = visitado[x][y] ? 1 : 0;
                }
            }
        }
    }

    if(dist < min_dist){ //Poda
        visitado[x][y] = true;

    // Nos movemos en las 4 direcciones posibles
    if (CasillaValida(laberinto, visitado, x + 1, y)) {
        SolBackTracking(laberinto, visitado, solucion, x + 1, y, min_dist, dist + 1);
    }

    if (CasillaValida(laberinto, visitado, x, y + 1)) {
        SolBackTracking(laberinto, visitado, solucion, x, y + 1, min_dist, dist + 1);
    }

    if (CasillaValida(laberinto, visitado, x - 1, y)) {
        SolBackTracking(laberinto, visitado, solucion, x - 1, y, min_dist, dist + 1);
    }

    if (CasillaValida(laberinto, visitado, x, y - 1)) {
        SolBackTracking(laberinto, visitado, solucion, x, y - 1, min_dist, dist + 1);
    }

    // Retroceder
    visitado[x][y] = false;
    }//Si no, no se sigue explorando
}
```