Práctica 4: Algoritmos de exploración de grafos

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

Índice

Ejercicio 1.Diseño del algoritmo

```
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
```

Ejercicio 1.Funcionamiento con pequeña instancia

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:

Ejercicio 1.Funcionamiento con pequeña instancia

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).

Ejercicio 1.Funcionamiento con pequeña instancia

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.

Ejercicio 1.Funcionamiento con pequeña instancia

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

Ejercicio 2. Conveniencia máxima

Solución parcial: Tupla de valores x= {x1, x2, x3, ..., xn} donde xi representa un comensal en una posición de la mesa, siendo 3 <= n. Habrá tantos valores de xi como comensales.

Restricciones explícitas:

- Siempre se va a cumplir que x1 = 0.
- xi ∈ {1, n}
- xi ≠ xj

Restricciones implícitas:

- Se debe mantener un vector de usados para no insertar el mismo comensal en dos posiciones diferentes.
- Las componentes del vector usados modifican su valor cuando el comensal se encuentre en asignación.

Ejercicio 2. Funcionamiento

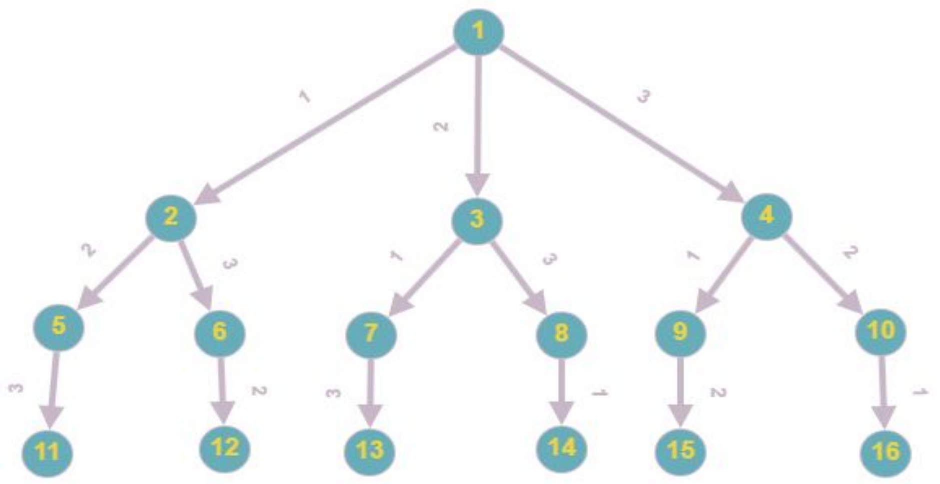
Exploración en profundidad.

El comensal 0 estará siempre sentado en primera posición por simplicidad.

Ej: Pequeña instancia para 4 comensales.

```
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
```

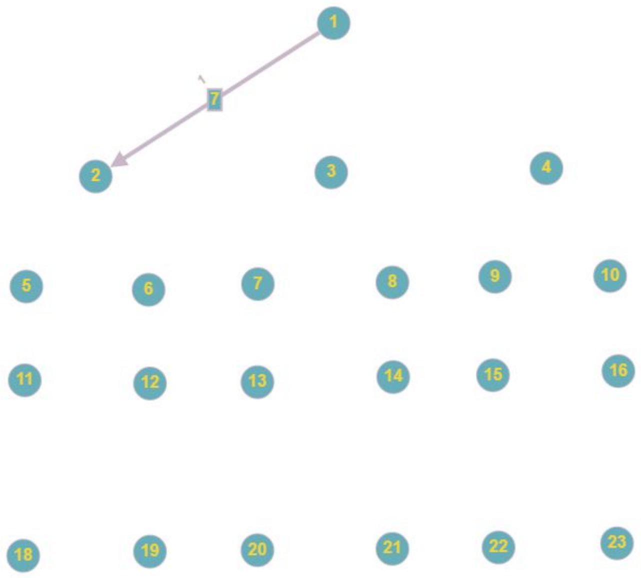
Ejercicio 2. Conveniencia máxima



Ejercicio 2. Funcionamiento con pequeña instancia

Comensales:	*0*	*1*	*2*	*3*
Comensal 0	0	**5**	4	6
Comensal 1	**2**	0	7	9
Comensal 2	6	2	0	3
Comensal 3	6	1	9	0

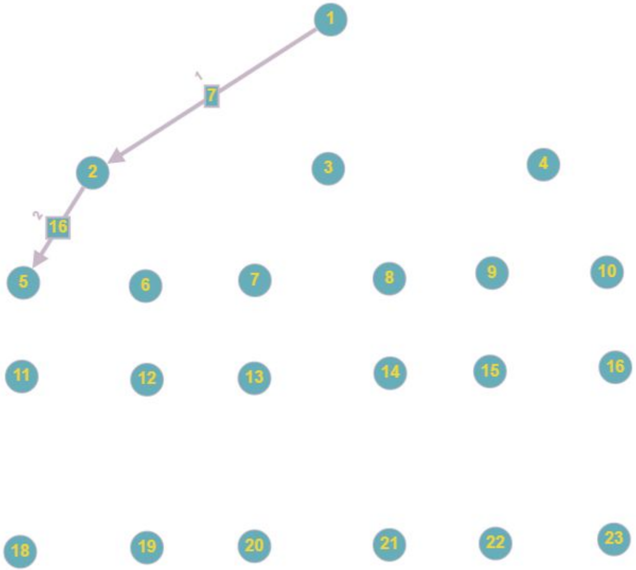
Asignación actual	***{0, 1}***
Conveniencia actual	***7***
Mejor conveniencia	*MIN*
Mejor asignación	*{}*



Ejercicio 2. Funcionamiento con pequeña instancia

Comensales:	*0*	*1*	*2*	*3*
Comensal 0	0	**5**	4	6
Comensal 1	**2**	0	**7**	9
Comensal 2	6	**2**	0	3
Comensal 3	6	1	9	0

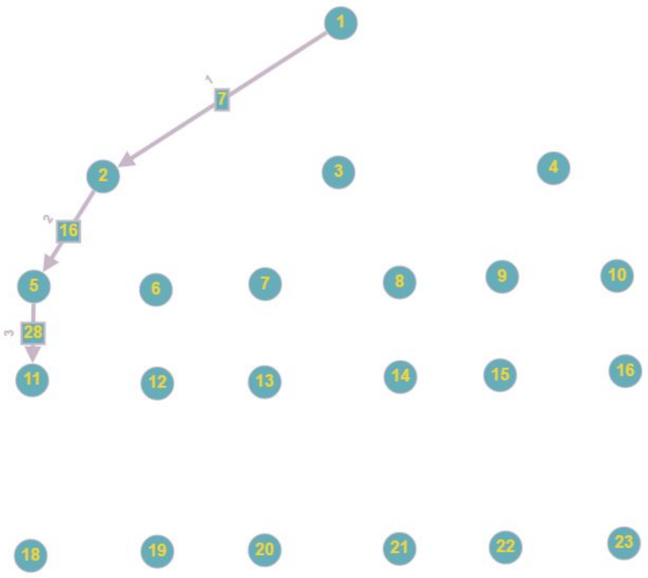
Asignación actual	***{0, 1, 2}***
Conveniencia actual	***16***
Mejor conveniencia	*MIN*
Mejor asignación	*{}*



Ejercicio 2. Funcionamiento con pequeña instancia

Comensales:	*0*	*1*	*2*	*3*
Comensal 0	0	**5**	4	6
Comensal 1	**2**	0	**7**	9
Comensal 2	6	**2**	0	**3**
Comensal 3	6	1	**9**	0

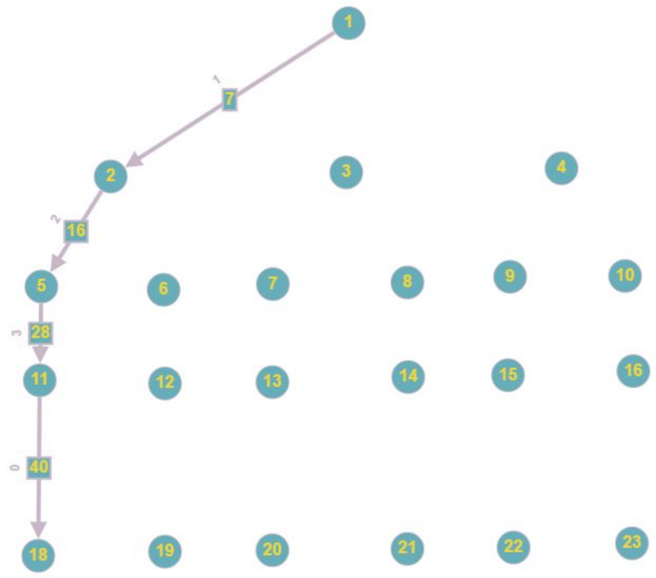
Asignación actual	***{0, 1, 2, 3}***
Conveniencia actual	***28***
Mejor conveniencia	*MIN*
Mejor asignación	*{}*



Ejercicio 2. Funcionamiento con pequeña instancia

Comensales:	0	1	2	3
Comensal 0	0	**5**	4	**6**
Comensal 1	**2**	0	**7**	9
Comensal 2	6	**2**	0	**3**
Comensal 3	**6**	1	**9**	0

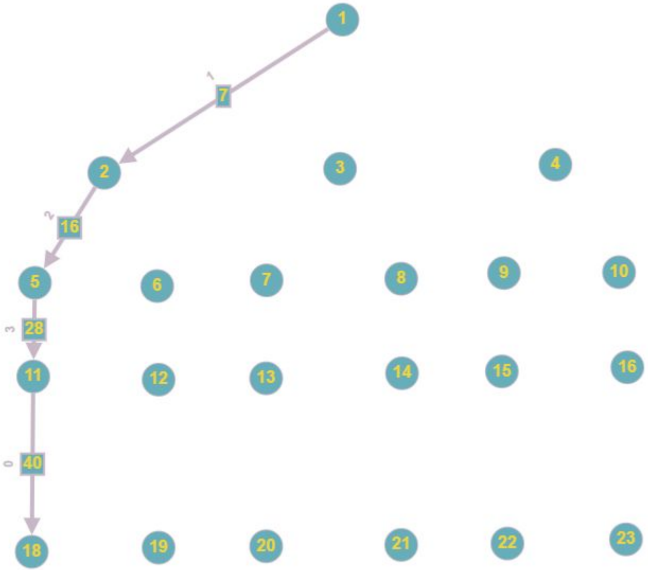
Asignación actual	*{0, 1, 2, 3}*
Conveniencia actual	***40***
Mejor conveniencia	*MIN*
Mejor asignación	*{}*



Ejercicio 2. Funcionamiento con pequeña instancia

Comensales:	0	1	2	3
Comensal 0	0	**5**	4	**6**
Comensal 1	**2**	0	**7**	9
Comensal 2	6	**2**	0	**3**
Comensal 3	**6**	1	**9**	0

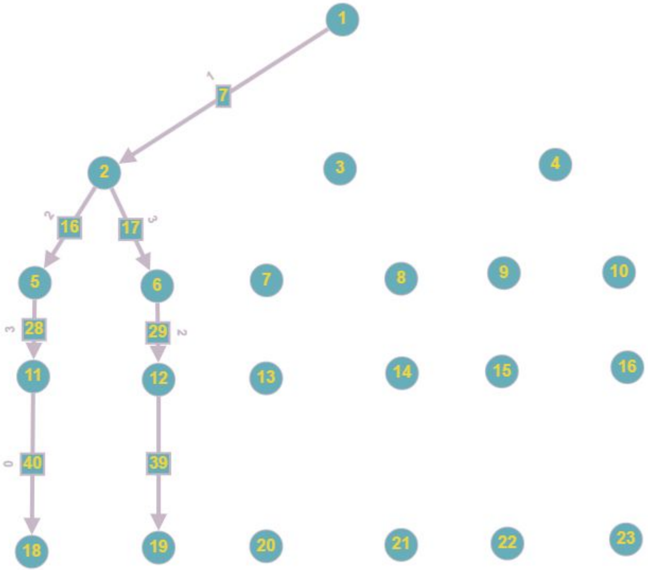
Asignación actual	*{0, 1, 2, 3}*
Conveniencia actual	*40*
Mejor conveniencia	***40***
Mejor asignación	***{0, 1, 2, 3}***



Ejercicio 2. Funcionamiento con pequeña instancia

Comensales:	0	1	2	3
Comensal 0	0	**5**	**4**	6
Comensal 1	**2**	0	7	**9**
Comensal 2	**6**	2	0	**3**
Comensal 3	6	**1**	**9**	0

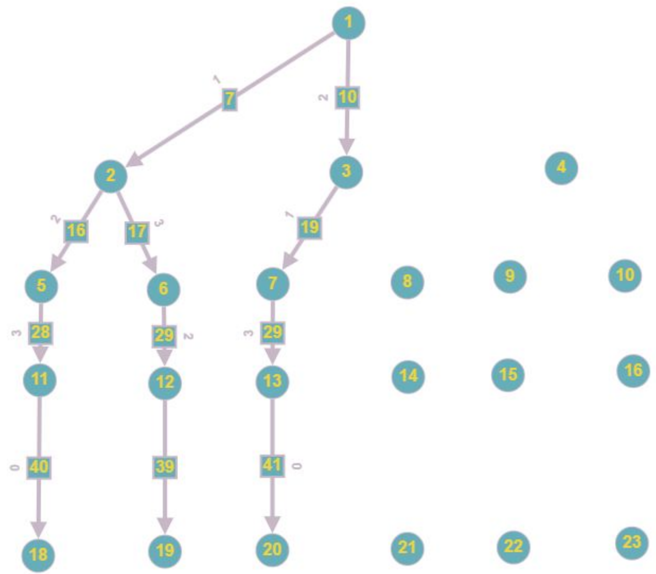
Asignación actual	***{0, 1, 3, 2}***
Conveniencia actual	***39***
Mejor conveniencia	*40*
Mejor asignación	*{0, 1, 2, 3}*



Ejercicio 2. Funcionamiento con pequeña instancia

Comensales:	0	1	2	3
Comensal 0	0	5	**4**	**6**
Comensal 1	2	0	**7**	**9**
Comensal 2	**6**	**2**	0	3
Comensal 3	**6**	**1**	9	0

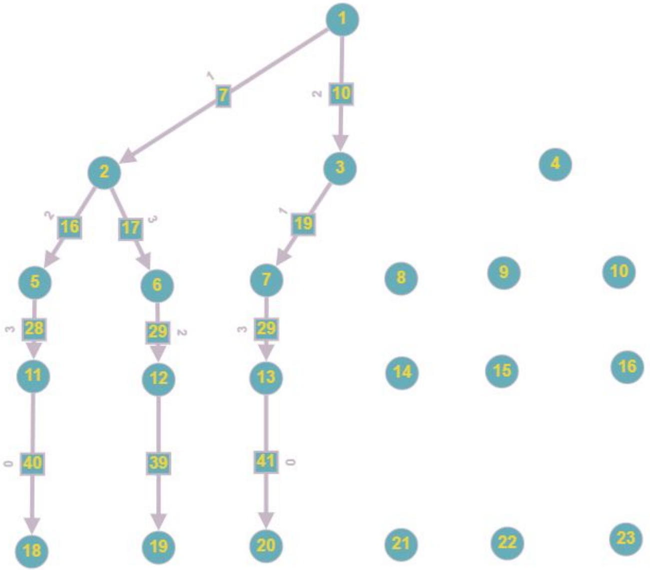
Asignación actual	***{0, 2, 1, 3}***
Conveniencia actual	***41***
Mejor conveniencia	*40*
Mejor asignación	*{0, 1, 2, 3}*



Ejercicio 2. Funcionamiento con pequeña instancia

Comensales:	0	1	2	3
Comensal 0	0	5	**4**	**6**
Comensal 1	2	0	**7**	**9**
Comensal 2	**6**	**2**	0	3
Comensal 3	**6**	**1**	9	0

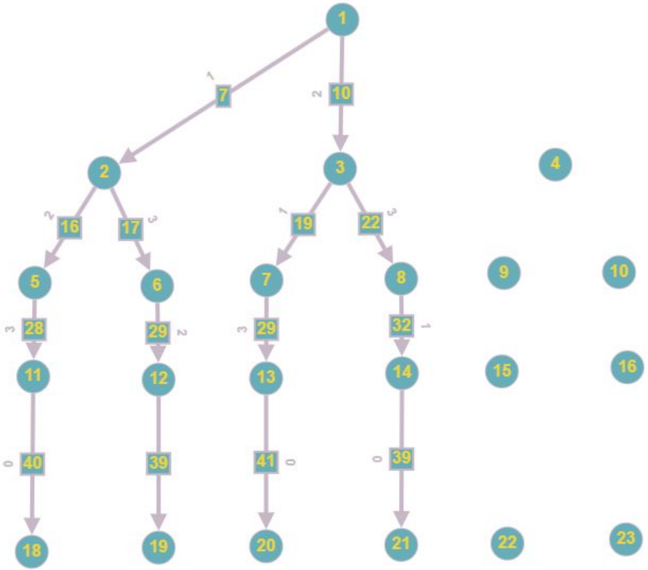
Asignación actual	*{0, 2, 1, 3}*
Conveniencia actual	*41*
Mejor conveniencia	***41***
Mejor asignación	***{0, 2, 1, 3}***



Ejercicio 2. Funcionamiento con pequeña instancia

Comensales:	0	1	2	3
Comensal 0	0	**5**	**4**	6
Comensal 1	**2**	0	7	**9**
Comensal 2	**6**	2	0	**3**
Comensal 3	6	**1**	**9**	0

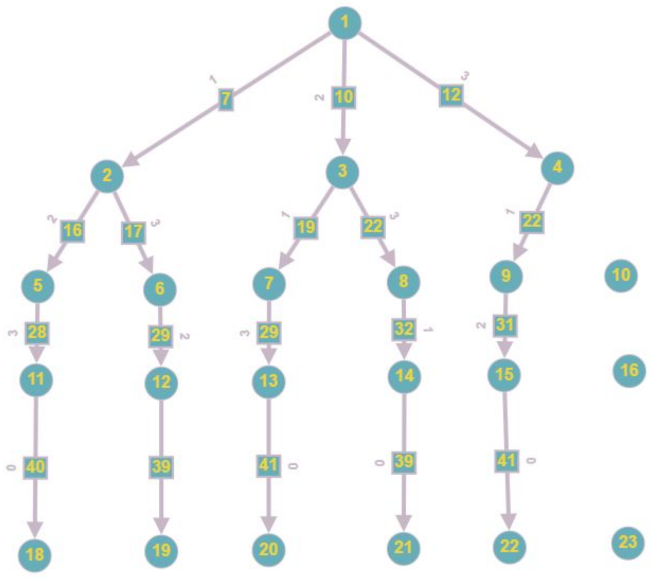
Asignación actual	***{0, 2, 3, 1}***
Conveniencia actual	***39***
Mejor conveniencia	*41*
Mejor asignación	*{0, 2, 1, 3}*



Ejercicio 2. Funcionamiento con pequeña instancia

Comensales:	0	1	2	3
Comensal 0	0	5	**4**	**6**
Comensal 1	2	0	**7**	**9**
Comensal 2	**6**	**2**	0	3
Comensal 3	**6**	**1**	9	0

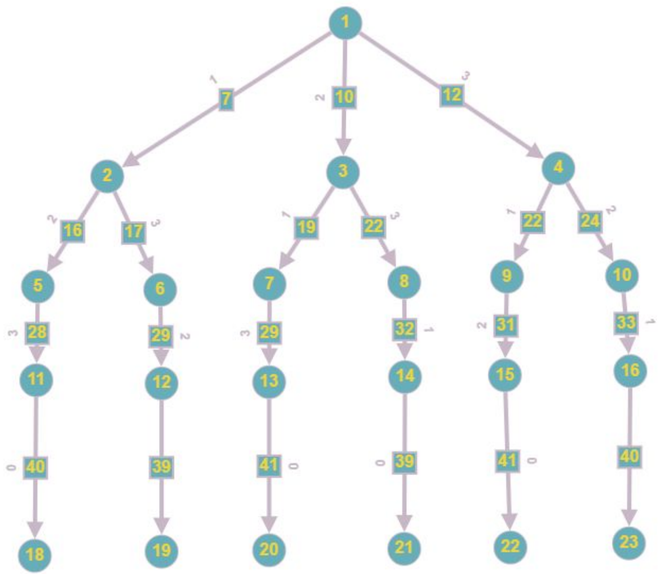
Asignación actual	***{0, 3, 1, 2}***
Conveniencia actual	***41***
Mejor conveniencia	*41*
Mejor asignación	*{0, 2, 1, 3}*



Ejercicio 2. Funcionamiento con pequeña instancia

Comensales:	0	1	2	3
Comensal 0	0	**5**	4	**6**
Comensal 1	**2**	0	**7**	9
Comensal 2	6	**2**	0	**3**
Comensal 3	**6**	1	**9**	0

Asignación actual	***{0, 3, 2, 1}***
Conveniencia actual	***40***
Mejor conveniencia	*41*
Mejor asignación	*{0, 2, 1, 3}*



Ejercicio 2. 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;
}
```

```cpp
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;
        }
    }
}
```

Ejercicio 3. Diseño

Solución parcial: Viene dada en una tupla de valores (x1, x2, ..., xm) donde xi representa un movimiento(posición inicial, final, dirección), 1 m 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:

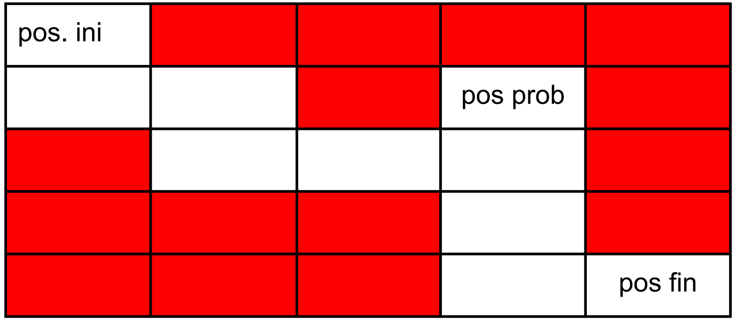
- Queda 1 sola bola en el centro del tablero o se puede hacer un movimiento válido.

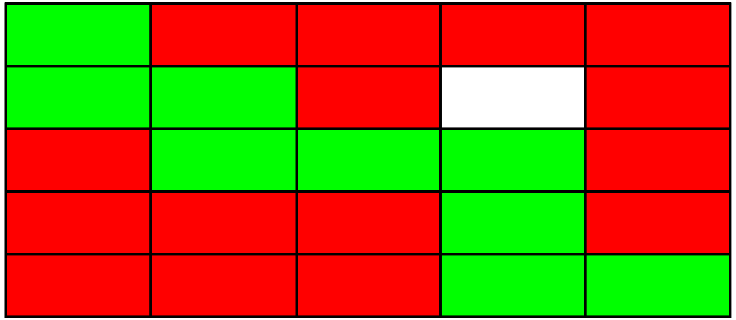
```
function senku(tablero)
      S <- ∅
      sol <- false
      mov <- {izquierda, arriba, derecha, abajo}

      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
```

Ejercicio 4 - Diseño del algoritmo

```
function CasillaValida(L, x, y)
        return (x >= 0 && x < N && y >= 0 && y < N && laberinto[x][y] = 1)


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
```

Ejercicio 4 - Funcionamiento para una instancia pequeña





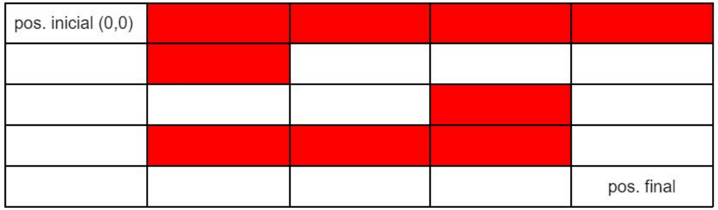
Ejercicio 4 - Implementación

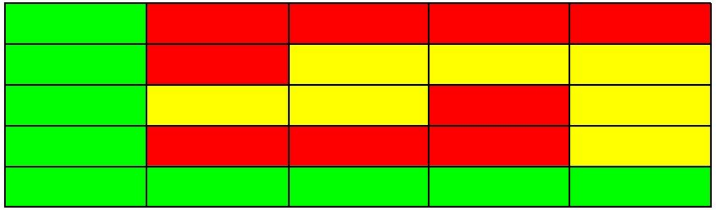
```cpp
const int N = 5; // Tamaño del laberinto

// 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;
}
```

Ejercicio 5 - Diseño del algoritmo

```
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
```

Ejercicio 5 - Funcionamiento para una instancia pequeña





Ejercicio 5 - 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){
        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
}
```