Rompecabezas de las Torres de Hanoi

El rompecabezas de las Torres de Hanoi consta de tres varillas verticales y un número de discos, que determinará la complejidad del problema, todos de distinto tamaño y apilados de mayor a menor radio en la primera varilla.

El objetivo del juego es pasar todos los discos de la primera a la última varilla, siguiendo tres simples reglas:

1. Se desplaza un disco cada vez.
2. Solo se pueden desplazar los discos de arriba de las varillas.
3. No se puede colocar un disco sobre otro más pequeño.

En esta primera parte de la práctica se mostrará cómo implementar el rompecabezas de las Torres de Hanoi como un problema de espacio de estados y se aplicarán distintos algoritmos de búsqueda para resolverlo.

Para implementar un problema de espacio de estados se pueden hacer uso de las clases de objetos proporcionadas por el módulo `problema_espacio_estados`.

In [2]:
```
import problema_espacio_estados as probee
```

Los algoritmos de búsqueda están implementados en el módulo `busqueda_espacio_estados`.

In [3]:
```
import busqueda_espacio_estados as busqee
```

El siguiente módulo será de utilidad para copiar un estado en otro estado igual, pero completamente nuevo e independiente.

In [4]:
```
import copy
```

El primer paso es decidir cómo se van a implementar los estados. Para el rompecabezas de las Torres de Hanoi una opción es hacerlo mediante una lista que guarde para cada varilla el conjunto de los discos que hay en ella.

In [5]:
```
estado1 = [{2}, set(), {1}]
estado2 = [{1}, set(), {2}]
```

A continuación hay que implementar las acciones como instancias de la clase `Accion`, proporcionando un nombre, una función de aplicabilidad y una función de aplicación para cada acción. Por ejemplo, la acción `De 1 a 3` que mueve un disco de la primera a la tercera varilla se puede implementar de la siguiente manera:

In [6]:
```
def esta_vacia(estado, varilla):
    return not estado[varilla - 1]

def disco_superior(estado, varilla):
    return min(estado[varilla - 1])

def aplicabilidad(estado):
    return (not esta_vacia(estado, 1) and
            (esta_vacia(estado, 3) or
             disco_superior(estado, 1) < disco_superior(estado, 3)))
```

```
def quitar_disco(estado, varilla):
    disco = disco_superior(estado, varilla)
    estado[varilla - 1].remove(disco)
    return disco

def poner_disco(estado, varilla, disco):
    estado[varilla - 1].add(disco)

def aplicacion(estado):
    nuevo_estado = copy.deepcopy(estado)
    disco = quitar_disco(nuevo_estado, 1)
    poner_disco(nuevo_estado, 3, disco)
    return nuevo_estado

a13 = probee.Accion('De 1 a 3', aplicabilidad, aplicacion)
```

In [7]:
```
a13.es_aplicable(estado1)
```

Out[7]:
```
False
```

In [8]:
```
a13.es_aplicable(estado2)
```

Out[8]:
```
True
```

In [9]:
```
a13.aplicar(estado2)
```

Out[9]:
```
[set(), set(), {1, 2}]
```

Normalmente las acciones se pueden agrupar en distintos tipos, cada uno de los cuales puede ser implementado de manera abstracta mediante una clase que herede de la clase `Accion`.

Para el rompecabezas de las Torres de Hanoi, todas las acciones son del tipo mover un disco de una varilla a otra. En este caso, consideramos que el coste de mover un disco es siempre 1, el valor por defecto. En caso de que fuera distinto, al crear una instancia de la clase `Accion` se puede proporcionar una función `coste`, o bien al heredar de la clase `Accion` se puede redefinir el método `coste_de_aplicar`.

In [10]:
```
class MoverDisco(probee.Accion):
    def __init__(self, i, j):
        nombre = 'De {} a {}'.format(i, j)
        super().__init__(nombre)
        self.varilla_de = i
        self.varilla_a = j

    def esta_vacia(self, estado, varilla):
        return not estado[varilla - 1]

    def disco_superior(self, estado, varilla):
        return min(estado[varilla - 1])

    def es_aplicable(self, estado):
        return (not self.esta_vacia(estado, self.varilla_de) and
                (self.esta_vacia(estado, self.varilla_a) or
                 self.disco_superior(estado, self.varilla_de) <
                 self.disco_superior(estado, self.varilla_a)))

    def quitar_disco(self, estado, varilla):
        disco = self.disco_superior(estado, varilla)
        estado[varilla - 1].remove(disco)
        return disco
```

```
    def poner_disco(self, estado, varilla, disco):
        estado[varilla - 1].add(disco)

    def aplicar(self, estado):
        nuevo_estado = copy.deepcopy(estado)
        disco = self.quitar_disco(nuevo_estado, self.varilla_de)
        self.poner_disco(nuevo_estado, self.varilla_a, disco)
        return nuevo_estado
```

Finalmente, un problema de espacio de estados se implementa como una instancia de la clase `ProblemaEspacioEstados`, proporcionando una lista de acciones, un estado inicial y una lista de estados finales.

In [11]:
```
acciones = [MoverDisco(i, j) for i in range(1, 4) for j in range(1, 4) if i != j]
estado_inicial = [{1, 2}, set(), set()]
estado_final = [set(), set(), {1, 2}]
Torres_Hanoi_2_discos = probee.ProblemaEspacioEstados(
    acciones, estado_inicial, [estado_final])
```

In [12]:
```
Torres_Hanoi_2_discos.es_estado_final(estado1)
```

Out[12]:
```
False
```

In [13]:
```
Torres_Hanoi_2_discos.es_estado_final(a13.aplicar(estado2))
```

Out[13]:
```
True
```

In [14]:
```
for accion in Torres_Hanoi_2_discos.acciones_aplicables(estado1):
    print(accion.nombre)
```

```
De 1 a 2
De 3 a 1
De 3 a 2
```

In [15]:
```
for accion in Torres_Hanoi_2_discos.acciones_aplicables(estado1):
    print(accion.aplicar(estado1))
```

```
[set(), {2}, {1}]
[{1, 2}, set(), set()]
[{2}, {1}, set()]
```

El procedimiento para realizar una búsqueda en un espacio de estados consiste en crear una instancia de una clase que implemente un algoritmo de búsqueda, proporcionando los argumentos necesarios, y aplicar el método buscar de esa instancia al problema de espacio de estados.

Las clases correspondientes a los algoritmos de búsqueda más comunes son las siguientes:

- `BúsquedaEnAnchura`
- `BúsquedaEnProfundidad`
- `BúsquedaPrimeroElMejor` : hay que proporcionar la función de evaluación heurística `h`.
- `BúsquedaÓptima`
- `BúsquedaAEstrella` : hay que proporcionar la función de estimación del coste `f = g + h`.

Adicionalmente, todas las clases anteriores admiten establecer el argumento `detallado` a `True`, para que al realizar una búsqueda se imprima por pantalla su traza.

In [16]:
```
b_anchura = busqee.BusquedaEnAnchura(detallado=True)
```

In [17]:
```
b_anchura.buscar(Torres_Hanoi_2_discos)
```

```
Nodo(1): Estado: [{1, 2}, set(), set()]; Prof: 0
  Nodo(2): Estado: [{2}, {1}, set()]; Prof: 1
  Nodo(3): Estado: [{2}, set(), {1}]; Prof: 1
    Nodo(4): Estado: [set(), {1}, {2}]; Prof: 2
    Nodo(5): Estado: [set(), {2}, {1}]; Prof: 2
      Nodo(6): Estado: [{1}, set(), {2}]; Prof: 3
      Nodo(7): Estado: [set(), set(), {1, 2}]; Prof: 3
```

Out[17]:
```
['De 1 a 2', 'De 1 a 3', 'De 2 a 3']
```

In [18]:
```
b_profundidad = busqee.BusquedaEnProfundidad(detallado=True)
```

In [19]:
```
b_profundidad.buscar(Torres_Hanoi_2_discos)
```

```
Nodo(1): Estado: [{1, 2}, set(), set()]; Prof: 0
  Nodo(2): Estado: [{2}, set(), {1}]; Prof: 1
    Nodo(3): Estado: [set(), {2}, {1}]; Prof: 2
      Nodo(4): Estado: [set(), {1, 2}, set()]; Prof: 3
      Nodo(5): Estado: [{1}, {2}, set()]; Prof: 3
        Nodo(6): Estado: [{1}, set(), {2}]; Prof: 4
          Nodo(7): Estado: [set(), set(), {1, 2}]; Prof: 5
```

Out[19]:
```
['De 1 a 3', 'De 1 a 2', 'De 3 a 1', 'De 2 a 3', 'De 1 a 3']
```

Podemos parametrizar la implementación del rompecabezas de las Torres de Hanoi para que dependa del número `n` de discos. Para ello basta implementar una clase que herede de la clase `ProblemaEspacioEstados`. Aprovechamos también para, en lugar de enumerar los estados finales, realizar una descripción declarativa de los mismos redefiniendo el método `es_estado_final`.

In [20]:
```
class TorresHanoi(probee.ProblemaEspacioEstados):
    def __init__(self, n):
        acciones = [MoverDisco(i, j) for i in range(1, 4) for j in range(1, 4) if i
        estado_inicial = [set(range(1, n + 1)), set(), set()]
        super().__init__(acciones, estado_inicial)
        self.n = n

    def es_estado_final(self, estado):
        return estado[2] == set(range(1, self.n + 1))
```

Con un número de discos igual a 8, el coste en tiempo de los algoritmos de búsqueda en anchura y profundidad comienza a no ser asumible, por lo que debemos pasar a realizar una búsqueda informada.

In [21]:
```
Torres_Hanoi_8_discos = TorresHanoi(8)
```

In [22]:
```
b_anchura = busqee.BusquedaEnAnchura()
```

In [23]:
```
from time import time
ti = time()
sol = b_anchura.buscar(Torres_Hanoi_8_discos)
print('Tiempo transcurrido: {} segundos.'.format(round(time() - ti,2)))
```

```
Tiempo transcurrido: 18.29 segundos.
```

La solución de menor tamaño para la instancia del problema con 8 discos tiene 255 pasos.

In [24]:
```
len(sol)
```

Out[24]:
```
255
```

In [25]:
```
# sol
```

Vamos a medir el tiempo de ejecución para distintas instancias del problema.

In [26]:
```
MAX_DISCOS = 8
b_anchura = busqee.BusquedaEnAnchura()

for i in range(1, MAX_DISCOS + 1):
    Torres_Hanoi_N_discos = TorresHanoi(i)
    ti = time()
    sol = b_anchura.buscar(Torres_Hanoi_N_discos)
    print('N discos: {} Tamaño solución: {} Tiempo transcurrido: {} segundos.'.forma
```

```
N discos: 1 Tamaño solución: 1 Tiempo transcurrido: 0.001 segundos.
N discos: 2 Tamaño solución: 3 Tiempo transcurrido: 0.001 segundos.
N discos: 3 Tamaño solución: 7 Tiempo transcurrido: 0.00499 segundos.
N discos: 4 Tamaño solución: 15 Tiempo transcurrido: 0.01384 segundos.
N discos: 5 Tamaño solución: 31 Tiempo transcurrido: 0.07402 segundos.
N discos: 6 Tamaño solución: 63 Tiempo transcurrido: 0.27717 segundos.
N discos: 7 Tamaño solución: 127 Tiempo transcurrido: 2.10687 segundos.
N discos: 8 Tamaño solución: 255 Tiempo transcurrido: 12.33921 segundos.
```

Ejercicio: El problema de las Jarras

- **Enunciado**:
  - Se tienen dos jarras, de 4 y 3 litros respectivamente.
  - Ninguna de ellas tiene marcas de medición.
  - Se tiene una bomba que permite llenar las jarras de agua.
  - Averiguar cómo se puede lograr tener exactamente 2 litrosde agua en la jarra de 4 litros de capacidad.
- **Estado inicial**: (0 0).
- **Estados finales**: todos los estados de la forma (2 y).
- **Acciones**:
  - Llenar la jarra de 4 litros con la bomba.
  - Llenar la jarra de 3 litros con la bomba.
  - Vaciar la jarra de 4 litros en el suelo.
  - Vaciar la jarra de 3 litros en el suelo.
  - Trasvasar de la jarra de 4 litros a la jarra de 3 litros. Si se llena la jarra de 3 litros, el sobrante permanece en la jarra de 4 litros.
  - Trasvasar de jarra de 3 litros a la jarra de 4 litros. Si se llena la jarra de 4 litros, el sobrante permanece en la jarra de 3 litros.

NOTA: Se ha usado una lista para representar el estado, al final del notebook se proporciona una solución al problema usando tuplas.

Llenar la jarra de 4 litros con la bomba.

In [27]:
```
# ESTADO: [x,y]
# La jarra de 4 contiene x litros y la jarra de 3 contiene y litros.

############################
### Opcion 1: Funciones ###
############################

def aplicabilidad(estado):
```

```
    return estado[0] < 4

def aplicacion(estado):
    nuevo_estado = copy.deepcopy(estado)
    nuevo_estado[0] = 4
    return nuevo_estado

llenar4 = probee.Accion('Llenar jarra de 4', aplicabilidad, aplicacion)

############################
### Opcion 2: Clases    ###
############################
# IMPORTANTE: No cambiar los nombres de las funciones es_aplicable y aplicar.
class LlenarJarraDe4(probee.Accion):
    def __init__(self):
        nombre = 'Llenar jarra de 4'
        super().__init__(nombre)

    def es_aplicable(self, estado):
        return estado[0] < 4

    def aplicar(self, estado):
        nuevo_estado = copy.deepcopy(estado)
        nuevo_estado[0] = 4
        return nuevo_estado

llenar4 = LlenarJarraDe4()
```

Llenar la jarra de 3 litros con la bomba.

In [28]:
```
class LlenarJarraDe3(probee.Accion):
    def __init__(self):
        nombre = 'Llenar jarra de 3'
        super().__init__(nombre)

    def es_aplicable(self, estado):
        return estado[1] < 3

    def aplicar(self, estado):
        nuevo_estado = copy.deepcopy(estado)
        nuevo_estado[1] = 3
        return nuevo_estado
```

Vaciar la jarra de 4 litros en el suelo.

In [29]:
```
class VaciarJarraDe4(probee.Accion):
    def __init__(self):
        nombre = 'Vaciar jarra de 4'
        super().__init__(nombre)

    def es_aplicable(self, estado):
        return estado[0] > 0

    def aplicar(self, estado):
        nuevo_estado = copy.deepcopy(estado)
        nuevo_estado[0] = 0
        return nuevo_estado
```

Vaciar la jarra de 3 litros en el suelo.

In [30]:
```
class VaciarJarraDe3(probee.Accion):
    def __init__(self):
        nombre = 'Vaciar jarra de 3'
```

```
        super().__init__(nombre)

    def es_aplicable(self, estado):
        return estado[1] > 0

    def aplicar(self, estado):
        nuevo_estado = copy.deepcopy(estado)
        nuevo_estado[1] = 0
        return nuevo_estado
```

Trasvasar de la jarra de 4 litros a la jarra de 3 litros.

In [31]:
```
class TrasvasarJarraDe4aJarraDe3(probee.Accion):
    def __init__(self):
        nombre = 'Trasvasar de jarra de 4 a jarra de 3'
        super().__init__(nombre)

    def es_aplicable(self, estado):
        return estado[0] > 0 and estado[1] < 3

    def aplicar(self, estado):
        nuevo_estado = copy.deepcopy(estado)
        if (estado[0] + estado[1]) > 3:
            nuevo_estado[0] = estado[0] - 3 + estado[1]
            nuevo_estado[1] = 3
        else:
            nuevo_estado[0] = 0
            nuevo_estado[1] = estado[0] + estado[1]
        return nuevo_estado
```

Trasvasar de la jarra de 3 litros a la jarra de 4 litros.

In [32]:
```
class TrasvasarJarraDe3aJarraDe4(probee.Accion):
    def __init__(self):
        nombre = 'Trasvasar de jarra de 3 a jarra de 4'
        super().__init__(nombre)

    def es_aplicable(self, estado):
        return estado[1] > 0 and estado[0] < 4

    def aplicar(self, estado):
        nuevo_estado = copy.deepcopy(estado)
        if (estado[0] + estado[1]) > 4:
            nuevo_estado[0] = 4
            nuevo_estado[1] = estado[1] - 4 + estado[0]
        else:
            nuevo_estado[0] = estado[0] + estado[1]
            nuevo_estado[1] = 0
        return nuevo_estado

######################################################
# IMPLEMENTACIONES ALTERNATIVAS DEL MÉTODO APLICAR #
######################################################
#      def aplicar(self, estado):
#         nuevo_estado = copy.deepcopy(estado)

#         nuevo_estado[1] = estado[0]+estado[1]
#         nuevo_estado[0] = 0
#         if nuevo_estado[1] > 3:
#             nuevo_estado[1] = 3
#             nuevo_estado[0] = estado[0] - (3 - estado[1])
#         return nuevo_estado
```

```
#     def aplicar(self, estado):
#         nuevo_estado = copy.deepcopy(estado)

#         for cantidad in range(estado[0]+1):
#             if estado[1]+cantidad == 3 or estado[0]-cantidad == 0:
#                 nuevo_estado[0] = estado[0] - cantidad
#                 nuevo_estado[1] = estado[1] + cantidad
#         return nuevo_estado
```

In [33]:
```
acciones = [LlenarJarraDe4(), LlenarJarraDe3(), VaciarJarraDe4(), VaciarJarraDe3(),
            TrasvasarJarraDe4aJarraDe3(), TrasvasarJarraDe3aJarraDe4()]
estado_inicial = [0, 0]
estados_finales = [[2, i] for i in range(4)]
Jarras_4_3 = probee.ProblemaEspacioEstados(acciones, estado_inicial, estados_finales
```

In [34]:
```
b_anchura = busqee.BusquedaEnAnchura(detallado=True)
b_anchura.buscar(Jarras_4_3)
```

```
Nodo(1): Estado: [0, 0]; Prof: 0
  Nodo(2): Estado: [4, 0]; Prof: 1
  Nodo(3): Estado: [0, 3]; Prof: 1
    Nodo(4): Estado: [4, 3]; Prof: 2
    Nodo(5): Estado: [1, 3]; Prof: 2
    Nodo(6): Estado: [3, 0]; Prof: 2
      Nodo(7): Estado: [1, 0]; Prof: 3
      Nodo(8): Estado: [3, 3]; Prof: 3
        Nodo(9): Estado: [0, 1]; Prof: 4
        Nodo(10): Estado: [4, 2]; Prof: 4
          Nodo(11): Estado: [4, 1]; Prof: 5
          Nodo(12): Estado: [0, 2]; Prof: 5
            Nodo(13): Estado: [2, 3]; Prof: 6
```

Out[34]:
```
['Llenar jarra de 4',
 'Trasvasar de jarra de 4 a jarra de 3',
 'Vaciar jarra de 3',
 'Trasvasar de jarra de 4 a jarra de 3',
 'Llenar jarra de 4',
 'Trasvasar de jarra de 4 a jarra de 3']
```

In [35]:
```
b_profundidad = busqee.BusquedaEnProfundidad(detallado=True)
b_profundidad.buscar(Jarras_4_3)
```

```
Nodo(1): Estado: [0, 0]; Prof: 0
  Nodo(2): Estado: [0, 3]; Prof: 1
    Nodo(3): Estado: [3, 0]; Prof: 2
      Nodo(4): Estado: [3, 3]; Prof: 3
        Nodo(5): Estado: [4, 2]; Prof: 4
          Nodo(6): Estado: [0, 2]; Prof: 5
            Nodo(7): Estado: [2, 0]; Prof: 6
```

Out[35]:
```
['Llenar jarra de 3',
 'Trasvasar de jarra de 3 a jarra de 4',
 'Llenar jarra de 3',
 'Trasvasar de jarra de 3 a jarra de 4',
 'Vaciar jarra de 4',
 'Trasvasar de jarra de 3 a jarra de 4']
```

- ¿Cómo podríamos generalizar la implementación para, por un lado reducir el número de acciones, y por otro poder trabajar con jarras de capacidades diferentes a 4 y 3?

ANEXO: El problema de las Jarras con tuplas

- **Enunciado**:
  - Se tienen dos jarras, de 4 y 3 litros respectivamente.
  - Ninguna de ellas tiene marcas de medición.

    - Se tiene una bomba que permite llenar las jarras de agua.
    - Averiguar cómo se puede lograr tener exactamente 2 litrosde agua en la jarra de 4 litros de capacidad.
- **Estado inicial**: (0 0).
- **Estados finales**: todos los estados de la forma (2 y).
- **Acciones**:
    - Llenar la jarra de 4 litros con la bomba.
    - Llenar la jarra de 3 litros con la bomba.
    - Vaciar la jarra de 4 litros en el suelo.
    - Vaciar la jarra de 3 litros en el suelo.
    - Trasvasar de la jarra de 4 litros a la jarra de 3 litros. Si se llena la jarra de 3 litros, el sobrante permanece en la jarra de 4 litros.
    - Trasvasar de jarra de 3 litros a la jarra de 4 litros. Si se llena la jarra de 4 litros, el sobrante permanece en la jarra de 3 litros.

Llenar la jarra de 4 litros con la bomba.

In [36]:
```
class LlenarJarraDe4(probee.Accion):
    def __init__(self):
        nombre = 'Llenar jarra de 4'
        super().__init__(nombre)

    def es_aplicable(self, estado):
        return estado[0] < 4

    def aplicar(self, estado):
        return (4, estado[1])
```

Llenar la jarra de 3 litros con la bomba.

In [37]:
```
class LlenarJarraDe3(probee.Accion):
    def __init__(self):
        nombre = 'Llenar jarra de 3'
        super().__init__(nombre)

    def es_aplicable(self, estado):
        return estado[1] < 3

    def aplicar(self, estado):
        return (estado[0], 3)
```

Vaciar la jarra de 4 litros en el suelo.

In [38]:
```
class VaciarJarraDe4(probee.Accion):
    def __init__(self):
        nombre = 'Vaciar jarra de 4'
        super().__init__(nombre)

    def es_aplicable(self, estado):
        return estado[0] > 0

    def aplicar(self, estado):
        return (0, estado[1])
```

Vaciar la jarra de 3 litros en el suelo.

In [39]:
```
class VaciarJarraDe3(probee.Accion):
    def __init__(self):
```

```
        nombre = 'Vaciar jarra de 3'
        super().__init__(nombre)

    def es_aplicable(self, estado):
        return estado[1] > 0

    def aplicar(self, estado):
        return (estado[0], 0)
```

Trasvasar de la jarra de 4 litros a la jarra de 3 litros.

In [40]:
```
class TrasvasarJarraDe4aJarraDe3(probee.Accion):
    def __init__(self):
        nombre = 'Trasvasar de jarra de 4 a jarra de 3'
        super().__init__(nombre)

    def es_aplicable(self, estado):
        return estado[0] > 0 and estado[1] < 3

    def aplicar(self, estado):
        if (estado[0] + estado[1]) > 3:
            nuevo_estado = (estado[0] - 3 + estado[1], 3)
        else:
            nuevo_estado = (0, estado[0] + estado[1])
        return nuevo_estado
```

Trasvasar de la jarra de 3 litros a la jarra de 4 litros.

In [41]:
```
class TrasvasarJarraDe3aJarraDe4(probee.Accion):
    def __init__(self):
        nombre = 'Trasvasar de jarra de 3 a jarra de 4'
        super().__init__(nombre)

    def es_aplicable(self, estado):
        return estado[1] > 0 and estado[0] < 4

    def aplicar(self, estado):
        if (estado[0] + estado[1]) > 4:
            nuevo_estado = (4, estado[1] - 4 + estado[0])
        else:
            nuevo_estado = (estado[0] + estado[1], 0)
        return nuevo_estado
```

In [42]:
```
acciones = [LlenarJarraDe4(), LlenarJarraDe3(), VaciarJarraDe4(), VaciarJarraDe3(),
            TrasvasarJarraDe4aJarraDe3(), TrasvasarJarraDe3aJarraDe4()]
estado_inicial = (0, 0)
estados_finales = [(2, i) for i in range(4)]
Jarras_4_3 = probee.ProblemaEspacioEstados(acciones, estado_inicial, estados_finales
```

In [43]:
```
b_anchura = busqee.BusquedaEnAnchura(detallado=True)
b_anchura.buscar(Jarras_4_3)
```

```
Nodo(1): Estado: (0, 0); Prof: 0
  Nodo(2): Estado: (4, 0); Prof: 1
  Nodo(3): Estado: (0, 3); Prof: 1
    Nodo(4): Estado: (4, 3); Prof: 2
    Nodo(5): Estado: (1, 3); Prof: 2
    Nodo(6): Estado: (3, 0); Prof: 2
      Nodo(7): Estado: (1, 0); Prof: 3
      Nodo(8): Estado: (3, 3); Prof: 3
        Nodo(9): Estado: (0, 1); Prof: 4
        Nodo(10): Estado: (4, 2); Prof: 4
          Nodo(11): Estado: (4, 1); Prof: 5
```

```
          Nodo(12): Estado: (0, 2); Prof: 5
            Nodo(13): Estado: (2, 3); Prof: 6
```

Out[43]:
```
['Llenar jarra de 4',
 'Trasvasar de jarra de 4 a jarra de 3',
 'Vaciar jarra de 3',
 'Trasvasar de jarra de 4 a jarra de 3',
 'Llenar jarra de 4',
 'Trasvasar de jarra de 4 a jarra de 3']
```

In [44]:
```
b_profundidad = busqee.BusquedaEnProfundidad(detallado=True)
b_profundidad.buscar(Jarras_4_3)
```

```
Nodo(1): Estado: (0, 0); Prof: 0
  Nodo(2): Estado: (0, 3); Prof: 1
    Nodo(3): Estado: (3, 0); Prof: 2
      Nodo(4): Estado: (3, 3); Prof: 3
        Nodo(5): Estado: (4, 2); Prof: 4
          Nodo(6): Estado: (0, 2); Prof: 5
            Nodo(7): Estado: (2, 0); Prof: 6
```

Out[44]:
```
['Llenar jarra de 3',
 'Trasvasar de jarra de 3 a jarra de 4',
 'Llenar jarra de 3',
 'Trasvasar de jarra de 3 a jarra de 4',
 'Vaciar jarra de 4',
 'Trasvasar de jarra de 3 a jarra de 4']
```

In [ ]: