Programación Declarativa

PROYECTO # 1

Beatriz Pérez Vera **Roman A. Pozas Peñate** **C-412**

b.vera@estudiantes.matcom.uh.cu r.pozas@estudiantes.matcom.uh.cu

Los jarros de agua:

Se dispone de dos jarros, uno con una capacidad de 3 litros, y otro con una capacidad de 4 litros. Ninguno de los dos tiene marcas que permitan medir cantidades que no sean las de sus propias capacidades.

Existe una pila de agua que permite llenar los jarros con agua y un sumidero donde se puede vaciar los mismos. El problema consiste en encontrar cuál es la secuencia de movimientos de llenado, vaciado y trasvase que permitan obtener exactamente dos litros de agua en el jarro de 4 litros.

Los jarros de agua:

Representación de los Estados:

Los estados se van representar por pares (X,Y) con ***X*** en {0,1,2,3,4} y ***Y*** en {0,1,2,3}.

Número de Estados:

20

Estado Inicial:

(0,0)

Estados Finales:

Todos los estados de la forma ***(2,_).***

```
objetivo([_,2]). %estado final
```

Operadores:

- Llenar la jarra de 4 litros con la pila
- Llenar la jarra de 3 litros con la pila
- Vaciar la jarra de 4 litros en el suelo
- Vaciar la jarra de 3 litros en el suelo
- Llenar la jarra de 4 litros con la jarra de 3 litros
- Llenar la jarra de 3 litros con la jarra de 4 litros
- Vaciar la jarra de 4 litros en la jarra de 3 litros
- Vaciar la jarra de 3 litros en la jarra de 4 litros

Reglas que se pueden aplicar:

1. Llenar la jarra de 4 litros: Si (X,Y) and X<4 => (4,Y)
2. Llenar la jarra de 3 litros: Si (X,Y) and Y<3 => (X,3)
3. Vaciar la jarra de 4 litros: Si (X,Y) and X>0 => (0,Y)
4. Vaciar la jarra de 3 litros: Si (X,Y) and Y>0 => (X,0)

Los jarros de agua:

5. Pasar agua de la jarra de 4 litros a la jarra de 3 litros hasta llenarla:
   Si (X,Y) and X>0 and X+Y>=3 => (X-(3-Y),3)
6. Pasar agua de la jarra de 3 litros a la jarra de 4 litros hasta llenarla:
   Si (X,Y) and Y>0 and X+Y>=4 => (4,Y-(4-X))
7. Pasar toda el agua de la jarra de 4 litros a la jarra de 3 litros:
   Si (X,Y) and X>0 and X+Y<3 => (0,X+Y)
8. Pasar toda el agua de la jarra de 3 litros a la jarra de 4 litros:
   Si (X,Y) and Y>0 and X+Y<4 => (X+Y,0)

El programa debería encontrar una secuencia de estados para ir del estado (0,0) al estado (2,0). Puede existir más de una secuencia de estados hacia la solución, por ejemplo:

(0,0) => (0,3) => (3,0) => (3,3) => (4,2) => (0,2) => (2,0)

En el cual a partir del estado inicial se aplicaron las reglas 2, 8, 2, 6, 3 y 8.

(0,0) => (4,0) => (1,3) => (1,0) => (0,1) => (4,1) => (2,3) => (2,0)

En el cual se aplicaron las reglas 1, 5, 4, 7, 1, 5, y 4

```
estado(vaciar_J4, [J3,_], [J3,0]).
estado(vaciar_J3, [_,J4], [0,J4]).

estado(llenar_J4, [J3,_], [J3,4]).
estado(llenar_J3, [_,J4], [3,J4]).

estado(hechar_J4_en_J3, [J3,J4], [NJ3,0]):-
        Sobrante is 3-J3,
        Sobrante >= J4,
        NJ3 is J3+J4.

estado(hechar_J4_en_J3, [J3,J4], [3,NJ4]):-
         Sobrante is 3-J3,
         Sobrante < J4,
         NJ4 is J4-Sobrante.

estado(hechar_J3_en_J4, [J3,J4], [0,NJ4]):-
         Sobrante is 4-J4,
         Sobrante >= J3,
         NJ4 is J3+J4.

estado(hechar_J3_en_J4, [J3,J4], [NJ3,4]):-
         Sobrante is 4-J4,
         Sobrante < J3,
         NJ3 is J3-Sobrante.
```

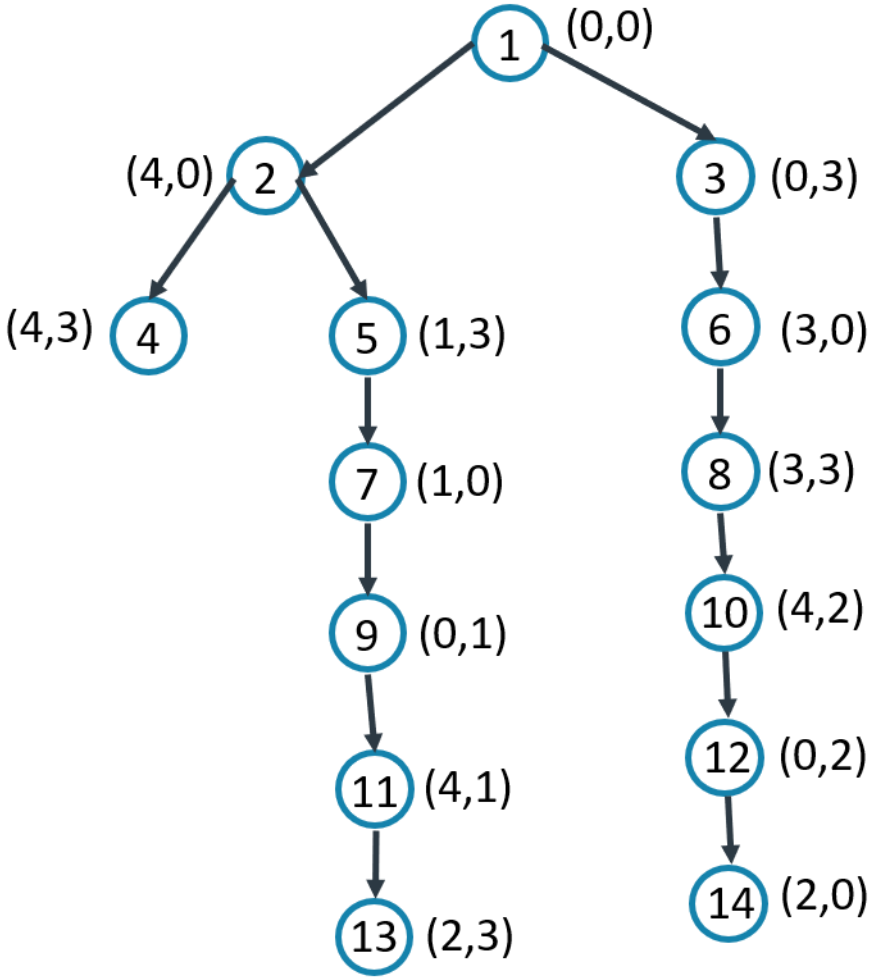
Solución

Como hemos visto anteriormente la solución del problema consiste en representar todos los posibles estados (esto es posible debido a que la cantidad de estados es finito) e ir moviéndonos en ellos hasta llegar al estado final.

```
solucion(Estado, [], Pasos):- Pasos >= 0, objetivo(Estado).
solucion(Estado, [X|Y], Pasos):-
         Pasos > 0,
         estado(X,Estado,NEstado),
         NPasos is Pasos-1,
         solucion(NEstado,Y,NPasos).
```

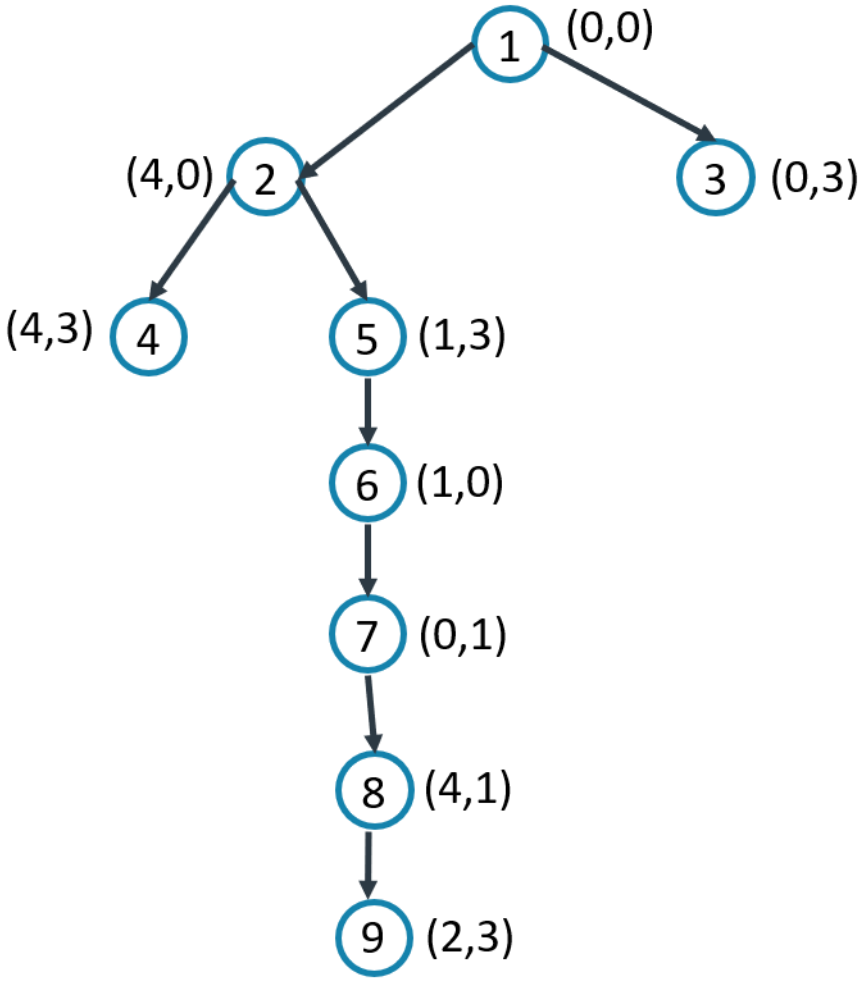
Los jarros de agua:

Grafo de búsqueda en anchura:



Los jarros de agua:

Grafo de búsqueda en profundidad:



Ejemplos:

1-)

```
1 ?- jarras(S,6).
S = [llenar_J4, hechar_J4_en_J3, vaciar_J3, hechar_J4_en_J3,
 llenar_J4, hechar_J4_en_J3] ■
```

Los jarros de agua:

2-)

```
1 ?- jarras(S,6).
S = [llenar_J4, hechar_J4_en_J3, vaciar_J3, hechar_J4_en_J3,
 llenar_J4, hechar_J4_en_J3] ;
S = [llenar_J3, hechar_J3_en_J4, llenar_J3, hechar_J3_en_J4,
 vaciar_J4, hechar_J3_en_J4] ;
false.
```

3-)

```
2 ?- jarras(S,8).
S = [vaciar_J4, vaciar_J4, llenar_J4, hechar_J4_en_J3,
vaciar_J3, hechar_J4_en_J3, llenar_J4, hechar_J4_en_J3]
 ;
S = [vaciar_J4, vaciar_J4, llenar_J3, hechar_J3_en_J4,
llenar_J3, hechar_J3_en_J4, vaciar_J4, hechar_J3_en_J4]
 ;
S = [vaciar_J4, vaciar_J3, llenar_J4, hechar_J4_en_J3,
vaciar_J3, hechar_J4_en_J3, llenar_J4, hechar_J4_en_J3]
```

Mundo de los bloques:

El mundo (ambiente) de los bloques está constituido por una superficie (mesa) y tres bloques (a, b y c) de igual tamaño.

En una configuración (estado) de este ambiente los bloques pueden aparecer encima de la mesa o colocados uno encima de otro.

Con este ambiente interacciona un robot que posee un brazo que es capaz de coger y mover un solo bloque en cada oportunidad.

La tarea del robot consiste en realizar una secuencia de acciones para transformar una configuración inicial en una configuración final u objetivo dada.

Este ambiente no es dinámico pues solamente cambia a partir de una configuración inicial dada bajo los efectos de las acciones del robot sobre el mundo.

Construir un programa que permita al robot obtener un plan para resolver cualquier caso particular del problema del mundo de los bloques.

Representación de Estados

Se utiliza una lista de listas, donde cada sub-lista representa un conjunto de bloques, ordenados de tal forma que el primer elemento de la sub-lista es el bloque más arriba del conjunto de bloques y las sub-listas vacías son espacios disponibles en la mesa.

Nótese que como la en la mesa siempre existirán espacios disponible, en caso de ser necesario, se adicionará al conjunto de listas, las listas vacías que sean necesarias para poder representar todas las maneras de colocar los bloques mediante la siguiente función:

```
adcionando_espacios_en_la_mesa(Cantidad_De_Bloques, Estado, Estado) :-
    length(Estado, Length),
    Length = Cantidad_De_Bloques,
    !.

adcionando_espacios_en_la_mesa(1, [], [[]]) :- !.

adcionando_espacios_en_la_mesa(Cantidad_De_Bloques, [Estado|Y], [Estado|R]) :-
    M is Cantidad_De_Bloques-1,
    adcionando_espacios_en_la_mesa(M, Y, R),
    !.

adcionando_espacios_en_la_mesa(Cantidad_De_Bloques, [], [[]|R]) :-
    M is Cantidad_De_Bloques-1,
    adcionando_espacios_en_la_mesa(M, [], R).
```

De esa forma representamos cada uno de los posibles “Estados” o “Estados de Mesa”.

Posibles Movimientos

Como el robot tiene un solo brazo un posible movimiento o cambio de estado es:

- tomar el bloque del tope de una de las pilas y ponerlo al inicio de otra.

```
mueve(MesaActual, NuevaMesa) :-
    select([Tope|Pila1], MesaActual,Resto),
    select(Pila2, Resto, RestoPilas),
    NuevaMesa=[Pila1,[Tope|Pila2]|RestoPilas].
```

Estados Finales

Después de un conjunto de movimientos o cambios de estados (conjunto que puede ser vacío), si se obtiene la configuración final definida por el usuario, se considera un triunfo.

```
objetivo([Pila_de_bloques], MesaActual) :-
    member(Pila_de_bloques, MesaActual).

objetivo([Pila|Resto_de_Mesa], MesaActual) :-
    member(Pila, MesaActual),
    objetivo(Resto_de_Mesa, MesaActual).
```

Ideas de Implementación

El algoritmo va a mantener una lista con todos los caminos activos.
En cada iteración del algoritmo consiste en tres pasos:

1. Quitar el primer camino de la lista de caminos
2. Generar un nuevo camino por cada posible siguiente estado etiquetando el ultimo nodo del camino seleccionado
3. Agregar el nuevo camino generado al final de la lista de caminos

Mundo de los bloques:

Solución:

Según la bibliografía consultada este tipo de problemas se pueden abordar de dos formas diferentes, como en el ejercicio anterior, es posible definir Nodos de un árbol o grafo donde cada nodo es un posible estado y la raíz o estado inicial es el definido por el usuario. Utilizando la naturaleza recursiva de Prolog una manera de movernos por el grafo es usando uno de los algoritmos de búsqueda DFS (búsqueda en profundidad) o BFS (búsqueda por niveles). Implementar DFS en Prolog es muy fácil, ya que el mismo Prolog lo usa para su funcionamiento interno, pero, debido a la naturaleza del problema se decido tomar el algoritmo de Búsqueda por niveles (BFS) por los beneficios que nos brindaba.

Ejemplos:

1-)

```
4 ?- mundo_de_los_bloques(3,[[a,b],[c]],[[a,b,c]],S).
S = [[[], [a, b], [c]], [[c], [b], [a]], [[b, c], [a],
[]], [[a, b, c], [], []]] 
```

2-)

```
2 ?- mundo_de_los_bloques(3,[[a,b],[c]],[[a,b,c]],
[[[a,b],[c]],[[c],[b],[a]],[[b,c],[a]],[[a,b,c]]])
.
true 
```

3-)

```
3 ?- mundo_de_los_bloques(3,[[a,b,c]],[[c,a],[b]],
[[[a,b,c]],[[b,c],[a]],[[a],[c],[b]],[[c,a],[b]]])
.
true 
```

Mundo de los bloques:

4-)

```
4 ?- mundo_de_los_bloques(5,[[f,b,a],[c],[d]],[[d,a,c],[b,f]],P)
.
P = [[[f, b, a], [c], [d]], [[f], [b, a], [c], [d]], [[c], [a],
[d], [b, f]], [[a, c], [d], [b, f]], [[d, a, c], [b, f]]]
```

Bibliografía:

1. Ulle Endriss. Logisch Programmeren en Zoektechnieken. Institute for Logic, Language and Computation University of Amsterdam.\
2. Juan Cruz Jiménez, A (2013), el problema de las jarras de agua con CLIPS-búsqueda primero en profundidad.