Bloque 1 – Representación del conocimiento y búsqueda

Tema 3:
Diseño de problemas basados en estados mediante SBR

Indice

1. Representación basada en estados
2. Diseño de problemas basados en estados mediante SBR
3. Resolución de problemas con SBR
4. El problema de las jarras de agua
5. El problema de las torres de Hanoi
6. El problema del ascensor

Bibliografía

- Capítulo 3: *Sistemas Basados en Reglas*. Inteligencia Artificial. Técnicas, métodos y aplicaciones. McGraw Hill, 2008.
- CLIPS User's Guide.
- CLIPS Basic Programming Guide.

1. Representación basada en estados

La resolución de un problema de IA se modela, habitualmente, como un proceso de búsqueda en un espacio de estados (situaciones o estados del problema). Consideramos problemas de un solo agente. La resolución de un problema consiste en encontrar una secuencia de estados que lleva a una solución.

Definición del problema:

1. Identificar el conjunto de posibles estados del problema
2. Especificar el estado inicial
3. Especificar el estado final o una función objetivo
4. Definir las reglas de transición u operadores del problema que permiten 'pasar' de un estado del problema a otro (acciones del problema)

Acción: transición que cuando se aplica en un estado **s** devuelve un nuevo estado como resultado de ejecutar la acción en **s**.

Espacio de estados: Conjunto de todos los estados alcanzables desde el estado inicial mediante la aplicación de una secuencia de acciones

Objetivo: Encontrar la secuencia de operadores o acciones (solución), la cual aplicada al estado inicial alcanza el estado final u objetivo del problema. Esto se realiza mediante un proceso de búsqueda en un espacio de estados.

Entorno: *observable* (el agente conoce el estado actual), *discreto* (número finito de acciones), *conocido* (el agente conoce los estados que se alcanzan por la aplicación de una acción), *determinista* (existe un único resultado asociado a cada acción)

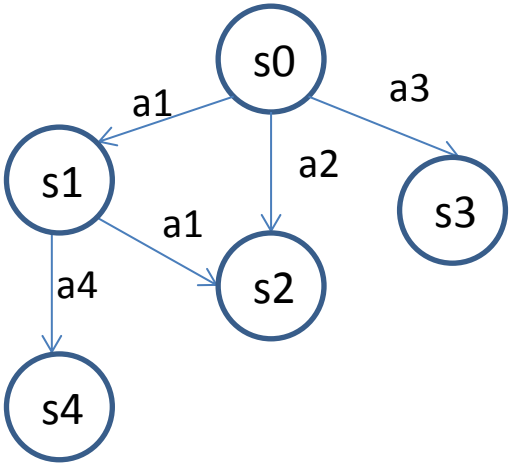
1. Representación basada en estados: formulación del problema

Estado inicial: estado inicial del problema (s0)

Acciones: acciones disponibles para el agente y que se pueden ejecutar en el problema. Por ejemplo: {a1,a2,a3,a4}

Actions(s) devuelve el conjunto de acciones aplicables/ejecutables en el estado **s**.

Actions(s0)={a1,a2,a3}, Actions(s1)={a1,a4}



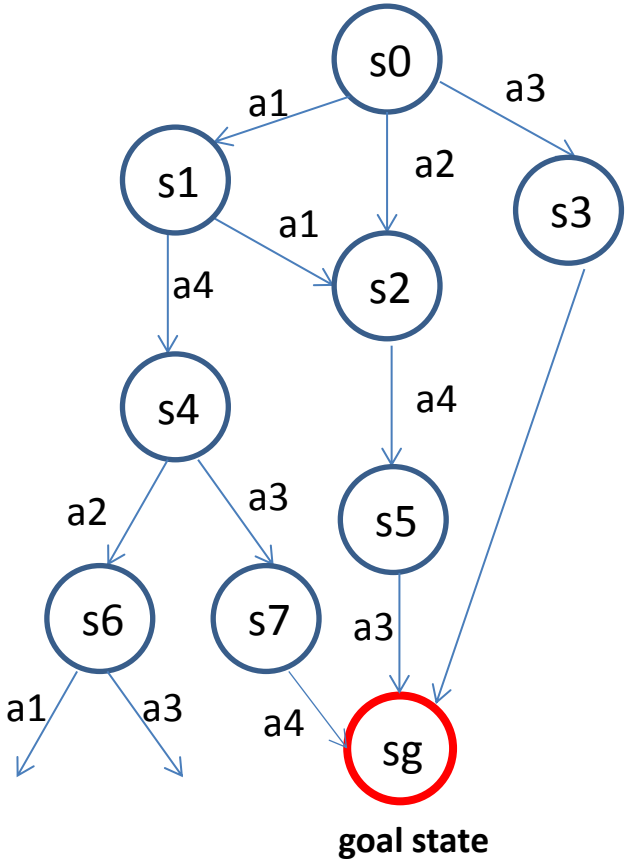
Modelo de transición: Result(s,a) devuelve el estado resultante de aplicar la acción **a** en el estado **s**.

Result(s0,a1) =s1, Result(s1,a1)=s2 ….

1. El estado inicial, junto con las acciones y el modelo de transición definen ímplicitamente el espacio de estados
2. El espacio de estados forma un grafo dirigido
3. Los nodos representan los estados del problema, las aristas las acciones del problema
4. Un camino del espacio de estados es una secuencia de estados conectados a través de una secuencia de acciones

1. Representación basada en estados: formulación del problema

Prueba objetivo: determina si un estado es un estado objetivo
- definición explícita de un estado en particular como el estado objetivo ó
- definición explícita de un conjunto de estados finales ó
- definición de una función objetivo (test) que aplicada sobre un estado devuelve si éste es un estado objetivo



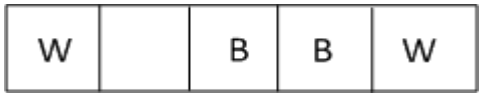
Coste del camino: función que asigna un coste numérico a cada camino

- El coste de aplicar la acción **a** en el estado **s** para alcanzar **s'** se denota como $c(s,a,s')$; generalmente, el coste de una acción es el mismo independientemente del estado en el que se aplique: $c(s0,a1,s1)=c(s1,a1,s2)=c(a1)$ (costes fijos de acciones)

- El coste de un camino se describe como la suma de los costes de las acciones individuales del camino

2. Diseño de problemas basado en estados mediante SBR

Queremos resolver el problema del puzzle lineal del tema 2 como un proceso de búsqueda en un espacio de estados mediante un SBR.



estado inicial

estado final

Vamos a analizar la representación del ejemplo 2 (transpas 7 y 8 del tema2):

1. Se utilizan varios hechos para representar los estados del problema.
2. Las reglas tienen un comando `retract` en la RHS por lo que al eliminar uno de los hechos que hace matching con un patrón de la regla, las restantes instancias de reglas de la Agenda se eliminan y eso impide generar el resto de nodos del árbol.

Base de Hechos

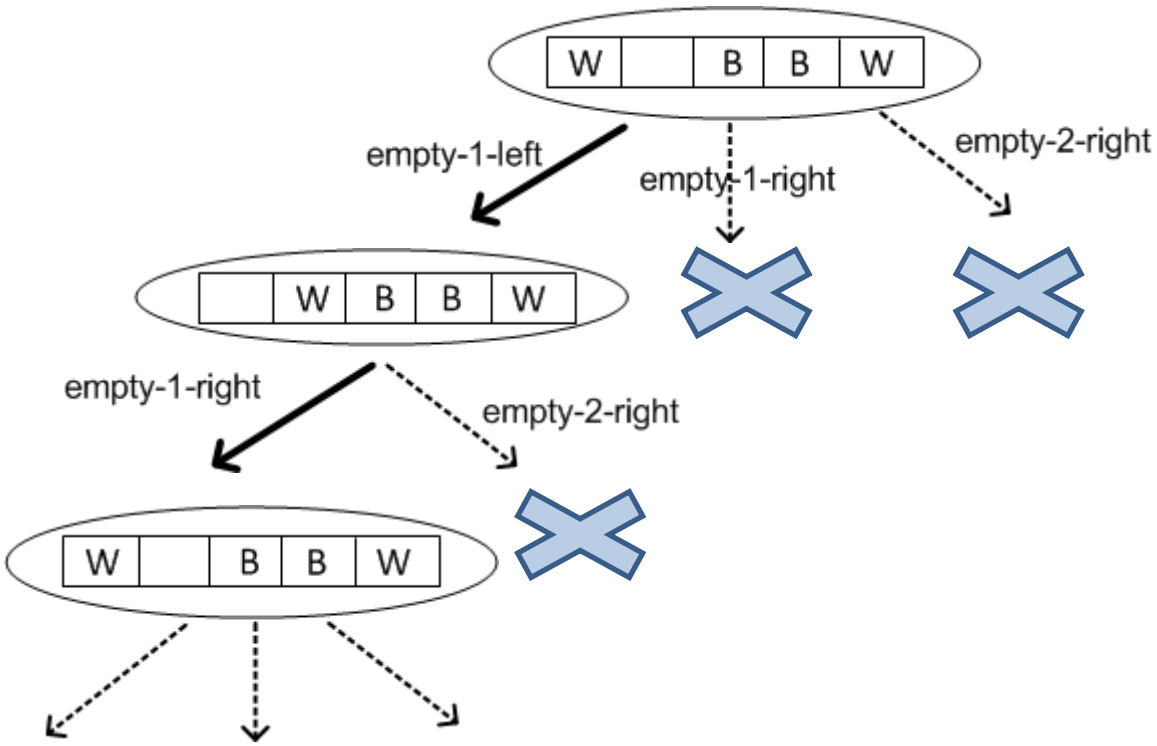
```
f-1: (cell 1 W)
f-2: (cell 2 E)
f-3: (cell 3 B)
f-4: (cell 4 B)
f-5: (cell 5 W)
```

Base de Reglas

```
(defrule emtpy-1-left
    ?f1 <-  (cell ?x E)
     ?f2 <- (cell ?y ?cell)
     (test (= ?y (- ?x 1)))
  =>
     (retract ?f1 ?f2)
     (assert (cell ?y E))
     (assert (cell ?x ?cell)))
```

Gráficamente, lo que sucede es ….

2. Diseño de problemas basado en estados mediante SBR



Base de Hechos

f-1: (cell 1 W)
f-2: (cell 2 E)
f-3: (cell 3 B)
f-4: (cell 4 B)
f-5: (cell 5 W)

el hecho f-2: (cell 2 E) se borra y como consecuencia se eliminan las instancias de reglas de la Agenda

Base de Hechos

f-3: (cell 3 B)
f-4: (cell 4 B)
f-5: (cell 5 W)
f-6: (cell 1 E)
f-7: (cell 2 W)

Base de Hechos

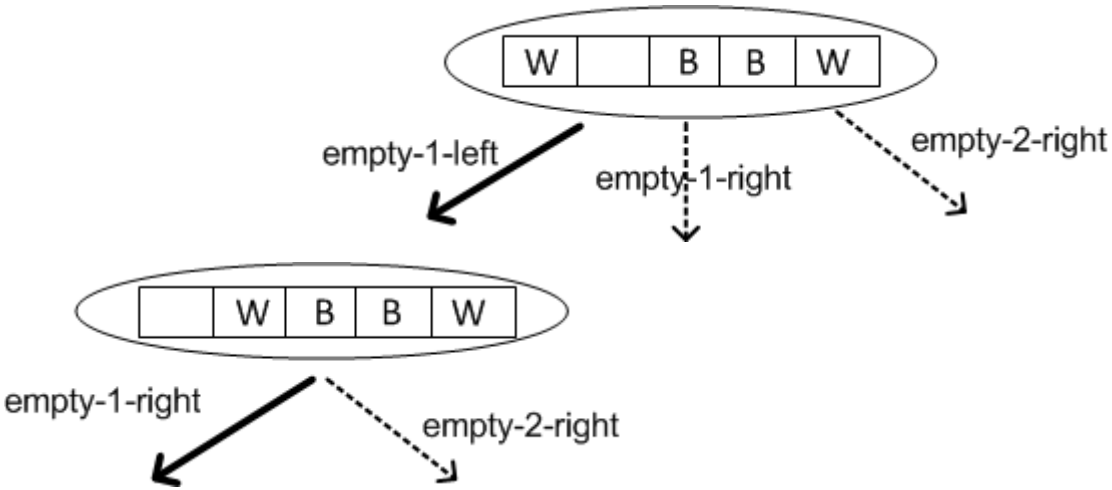
f-3: (cell 3 B)
f-4: (cell 4 B)
f-5: (cell 5 W)
f-8: (cell 2 E)
f-9: (cell 1 W)

OBJETIVO: generar el árbol de búsqueda del problema hasta encontrar un nodo que coincida con el estado final del problema ➔ las reglas no deben eliminar hechos que impidan la generación de estados del problema ➔ NO UTILIZAR COMANDO **retract**

IMPORTANTE: los nodos del árbol de búsqueda (estados del problema) deben estar representados consistentemente en la BH

2. Diseño de problemas basado en estados mediante SBR

Analizamos las consecuencias si eliminamos el comando `retract` de de las reglas.



Base de Hechos

f-1: (cell 1 W)
f-2: (cell 2 E)
f-3: (cell 3 B)
f-4: (cell 4 B)
f-5: (cell 5 W)

Base de Hechos

f-1: (cell 1 W)
f-2: (cell 2 E)
f-3: (cell 3 B)
f-4: (cell 4 B)
f-5: (cell 5 W)
f-6: (cell 1 E)
f-7: (cell 2 W)

estado inconsistente

La BH mantiene todos los hechos que insertan las reglas pero no es posible identificar los hechos que representan un estado u otro.

Solución: representar los estados del problema con un único hecho

2. Diseño de problemas basado en estados mediante SBR

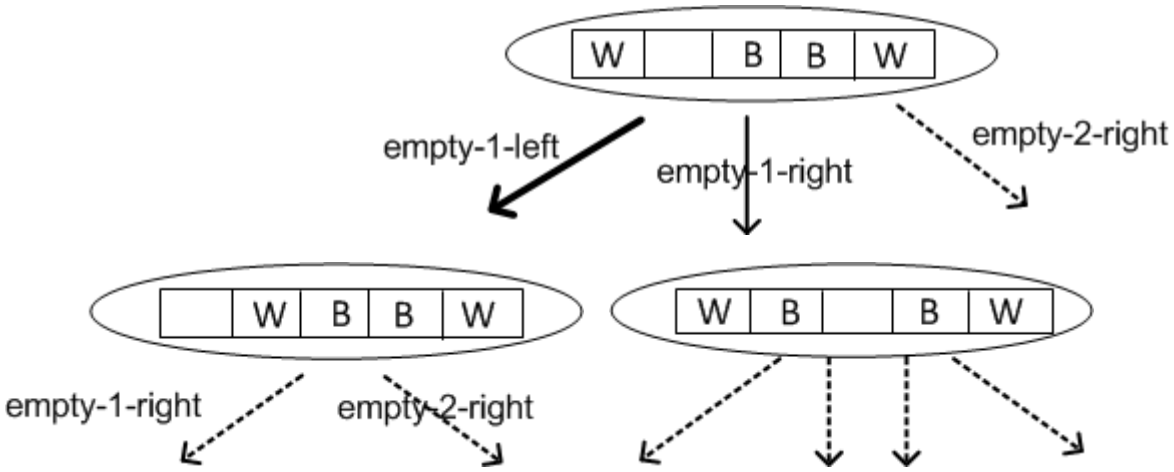
Utilizamos ahora la representación del puzzle lineal del ejemplo 1 del tema 2 (transpas 5 y 6) y analizamos el proceso inferencial habiendo eliminado el comando `retract` de las reglas.

Base de Hechos

f-1: (puzzle W E B B W)

Base de Reglas

```
(defrule emtpy-1-left
    ?f1 <-  (puzzle $?x ?y E $?z)
  =>
    (assert (puzzle $?x E ?y $?z)))
```



Base de Hechos

f-1: (puzzle W E B B W)

f-2: (puzzle E W B B W)

f-3: (puzzle W B E B W)

3. Resolución de problemas mediante SBR

Representación de los estados del problema: patrones y hechos

- Utilizar un lenguaje de representación de hechos (patrones).
- El diseño del patrón debe poder representar cada posible estado del problema.
- Debe representarse (solo) la información relevante para el objetivo a obtener.
- Un buen diseño debe permitir identificar fácil y claramente los componentes importantes de los estados, facilitando el diseño de la parte izquierda y parte derecha de la reglas.

1. Información dinámica
   - Datos que pueden cambiar como resultado de aplicación de las acciones
   - Mantener la información dinámica en un solo patrón/hecho que represente un estado/situación particular del problema
   - Ejemplos: (contents jug X 2 jug Y 1), (puzzle W E B B W)

2. Información estática
   - Datos que no cambian a lo largo de la evolución del problema
   - Se pueden utilizar tantos hechos como se desee para representar la información estática
   - Ejemplos: (capacity jug X 4), (road cityA cityB)

3. Resolución de problemas mediante SBR

Representación de las acciones del problema: reglas

- Las reglas representan las posibles acciones que se pueden realizar para modificar un estado del problema (operadores del problema).
- Una acción del problema se representa con una regla.

Características:

Completitud: el conjunto de reglas diseñadas debe garantizar que es posible encontrar una solución para un problema que es resoluble.

1. El conjunto de reglas deben representar todas las posibles acciones que se puedan hacer en el problema
2. La parte izquierda debe representar solo las condiciones que son necesarias para aplicar la regla.
3. La parte derecha debe representar todos los cambios correspondientes a la acción.

Correctitud: el conjunto de reglas diseñadas debe representar adecuadamente los cambios en los estados del problema.

1. La parte izquierda debe expresar todas las condiciones que se tienen que cumplir para aplicar las reglas.
2. La parte derecha debe representar solo los cambios correspondientes a la acción

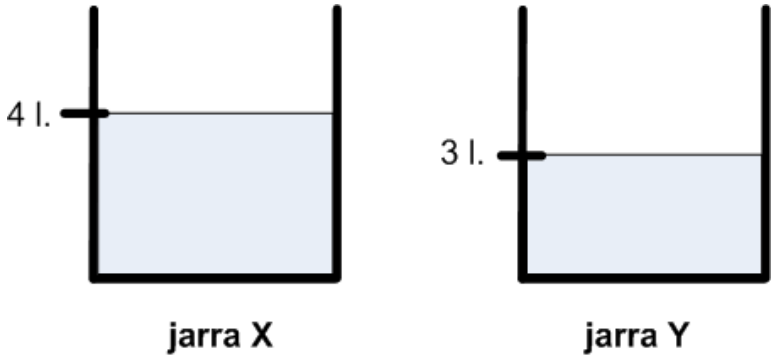
3. Resolución de problemas mediante SBR

Resolución de conflictos:

El criterio para la resolución de conflictos es fundamental para una eficiente resolución del problema

1. La elección de una estrategia de resolución de conflictos distinta puede conducir a una ejecución distinta del SBR
2. Las reglas interaccionan entre sí por lo que el orden en el que éstas se disparan es determinante en la ejecución del SBR (determina la generación del espacio de búsqueda)
3. Es importante no solo realizar un buen diseño de las reglas sino tener en cuenta cuando éstas se van a disparar y las relaciones que surgen entre las reglas.

4. El problema de las jarras de agua



- Máxima capacidad de la jarra X es 4 l.
- Máxima capacidad de la jarra Y es 3 l.
- *Inicialmente*, ambas jarras están vacías
- *Objetivo*: tener 2l. en la jarra X
- No hay marcas de medida excepto la de máx. capacidad

```
(deffacts water-jug-problem
    (capacity jug X 4)
    (capacity jug Y 3)

    (contents jug X 0 jug Y 0)
)
```

→ Hechos que representan información estática que no cambia nunca

→ estado inicial

Acciones:

Llenar jarra X
Llenar jarra Y
Vaciar jarra X
Vaciar jarra Y
Llenar X desde Y
Llenar Y desde X
Vaciar Y en X
Vaciar X en Y

4. El problema de las jarras de agua

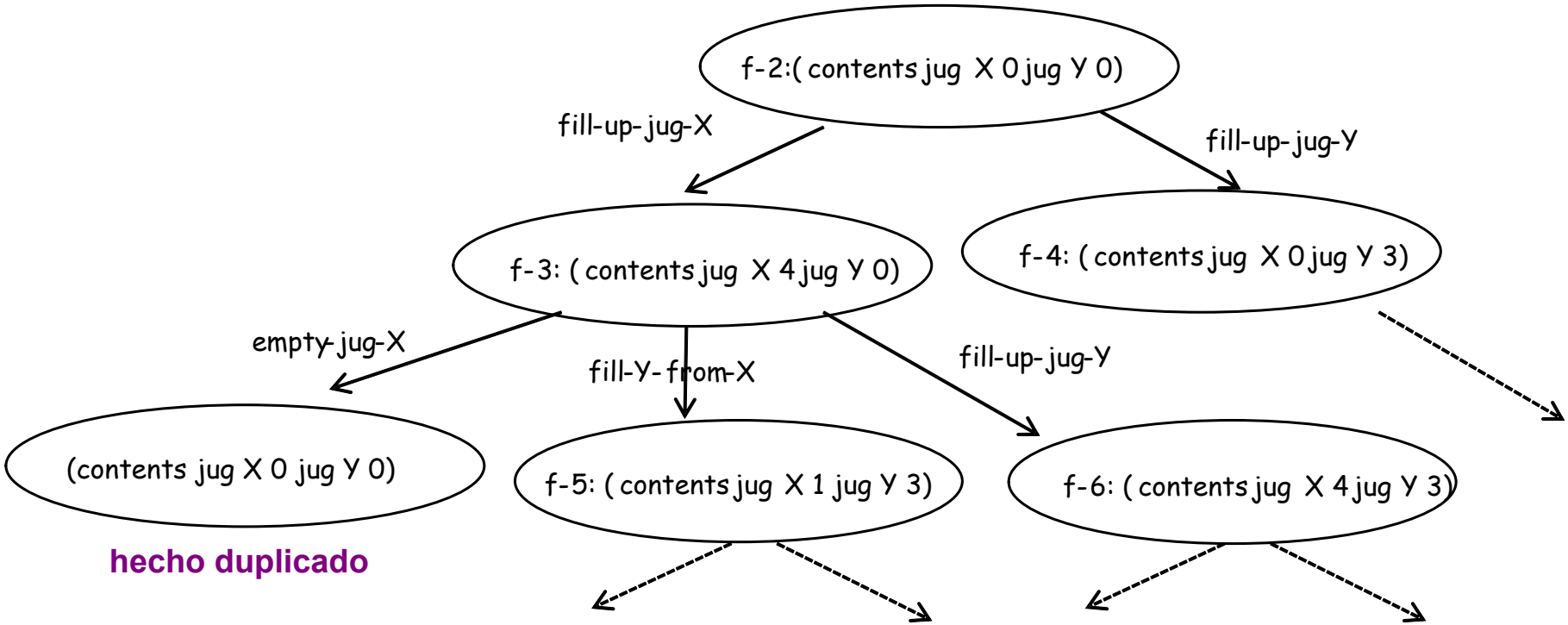
```
(defrule fill-up-jug-X
      (capacity jug X ?cap)
      (contents jug X ?x jug Y ?y)
      (test (< ?x ?cap))
 =>
      (assert (contents  jug X ?cap jug Y ?y)))
```

```
(defrule empty-jug-X
      (contents jug X ?x jug Y ?y)
      (test (> ?x 0))
 =>
      (assert (contents  jug X 0 jug Y ?y)))
```

```
(defrule fill-X-from-Y
      (contents jug X ?x jug Y ?y)
      (capacity jug X ?cap)
      (test (< ?x ?cap))
      (test (>= (+ ?x ?y) ?cap))
=>
      (assert
         (contents jug X ?cap jug Y (- ?y (- ?cap ?x)))))
```

```
(defrule pour-Y-into-X
      (contents jug X ?x jug Y ?y)
      (capacity jug X ?cap)
      (test (<= (+ ?x ?y) ?cap))
      (test (> ?y 0))
=>
      (assert
         (contents jug X (+ ?x ?y) jug Y 0)))
```

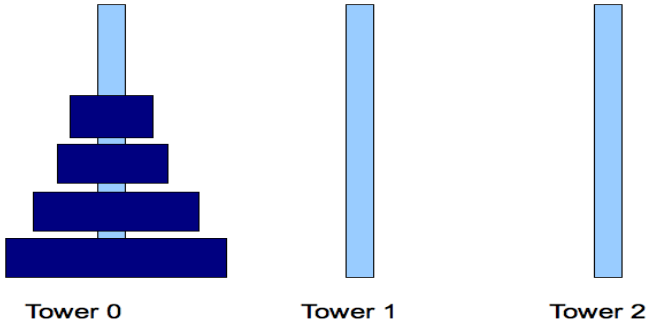
4. El problema de las jarras de agua



BH={f-0: (capacity jug X 4) f-1: (capacity jug Y 3) f-2: (contents jug X 0 Y 0) f-3 f-4 f-5 f-6 f-7 … }

5. El problema de las torres de Hanoi

Diseño para cuatro discos y tres torres



Identificación de objetos y relaciones entre objetos:

- Torres (T1, T2, T3)
- Discos (D1, D2, D3, D4) tal que Di > Dj si i > j. Los discos pueden representarse con símbolos (Di) ó directamente con un valor numérico que representa el tamaño del disco.
- Discos en una torre

5. El problemas de las torres de Hanoi

Patrón para representar los estados del problema:

(hanoi [tower tw^s dk^m]^m) ;; válido para cualquier número de torres y discos en cada torre

- El superíndice muestra el tipo de variable (s-single-valued, m-multi-valued)
- tw^s es el identificador de la torre, $tw^s \in$ {T1,T2,T3}
- dk^m: variable multivaluada que representa los discos de una torre, $dk^m \in 2^{\{1\text{-}4\}}$
- dk^m representa un conjunto de cero, uno, dos, tres o cuatro elementos del conjunto {1-4} ; los elementos que constituyen el valor de la variable tienen que satisfacer las restricciones del problema.
- Ejemplos: (1 3), (2 3 4), (), (1 4), etc.
- [tower tw^s dk^m] representa la información de la torre tw^s
- [tower tw^s dk^m]^m representa un número indeterminado de torres

Hechos:

Estado inicial del problema: BHinicial= {(hanoi tower T1 1 2 3 4 tower T2 tower T3)}

Estado final del problema: BHfinal= {(hanoi tower T1 tower T2 towerT3 1 2 3 4)}

Ejemplos del espacio de estados (estados del problema):

s1={(hanoi tower T1 3 4 tower T2 1 tower T3 2)}

s2={(hanoi tower T1 4 tower T2 2 tower T3 1 3)}

s3 ={(hanoi tower T1 1 tower T2 2 4 tower T3 3)}

5. El problema de las torres de Hanoi (SBR 1)

```
(defrule move-disk-from-T1-to-empty-tower-T2
    (hanoi  towerT1 ?d1 $?rest1 tower T2 tower T3 $?rest3)
 =>
    (assert (hanoi tower T1 $?rest1 tower T2 ?d1 tower T3 $?rest3)))

(defrule move-disk-from-T1-to-tower-with-disks-T2
     (hanoi  towerT1 ?d1 $?rest1 tower T2 ?d2 $?rest2 )
     (test (neq ?d2 tower))
     (test (< ?d1 ?d2))
 =>
    (assert (hanoi tower T1 $?rest1 tower T2 ?d1 ?d2 $?rest2)))

(defrule move-disk-from-T1-to-empty-tower-T3
   (hanoi  towerT1 ?d1 $?rest1 tower T3)
  =>
    (assert (hanoi tower T1 $?rest1 tower T3 ?d1)))
...
 (defrule final
    (declare (salience 100))
    (hanoi $? tower T3 $?t3)
    (test (= (length $?t3) 4))
 =>
     (halt)
     (printout t "Solution found " crlf))
```

5. El problema de las torres de Hanoi (SBR 2)

Podemos escoger un patrón más uniforme (solo válido para cuatro discos):

(hanoi [tower tw^s $d1^s$ $d2^s$ $d3^s$ $d4^s$]^m) ;; $tw^s \in$ {T1,T2,T3} $di^s \in$ [0-4]

Hechos:

(hanoi tower T1 1 2 3 4 tower T2 0 0 0 0 tower T3 0 0 0 0)
(hanoi tower T1 2 3 4 0 tower T2 1 0 0 0 tower T3 0 0 0 0)
(hanoi tower T1 2 4 0 0 tower T2 0 0 0 0 tower T3 1 3 0 0)

Reglas: podemos generalizar la regla **move-disk** independientemente de que la torre destino esté vacía o no

```
(defrule move-disk-from-T1-to-T2
   (hanoi  tower T1 ?d1 $?rest1 tower T2 ?d2 $?rest2 0 tower T3 $?rest3)
   (test (and (<> ?d1 0) (or (= ?d2 0)(< ?d1 ?d2))))
=>
   (assert (hanoi tower T1 $?rest1 0 tower T2 ?d1 ?d2 $?rest2 tower T3 $?rest3)))
```

5. El problema de las torres de Hanoi (SBR 2)

Dado que los hechos son hechos ordenados, es decir, los elementos del hecho se referencian posicionalmente, tenemos que escribir 6 reglas:

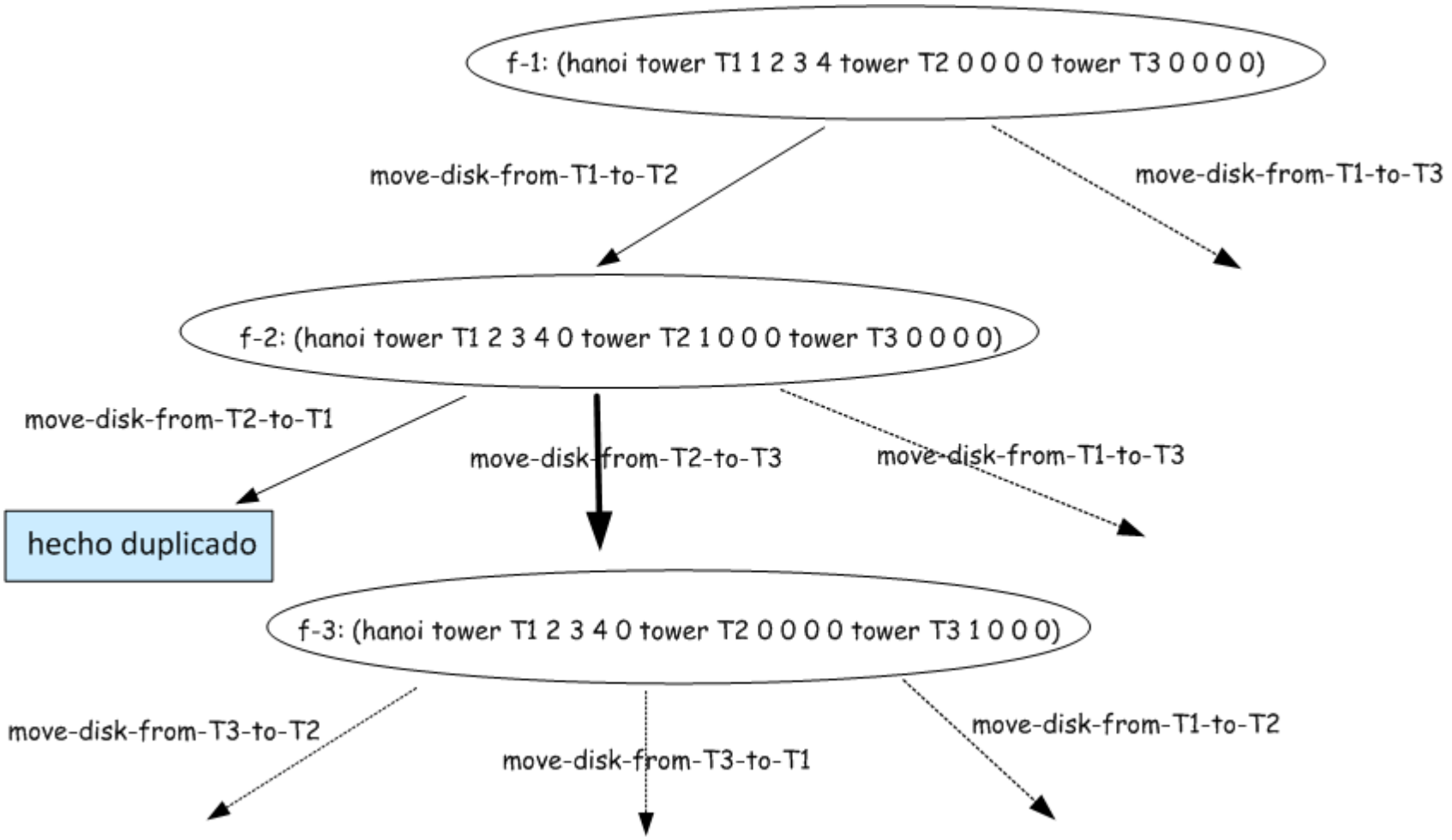
```
(defrule move-disk-from-T1-to-T2  ...
(defrule move-disk-from-T1-to-T3  ...
(defrule move-disk-from-T2-to-T1  ...
(defrule move-disk-from-T2-to-T3  ...
(defrule move-disk-from-T3-to-T1  ...
(defrule move-disk-from-T3-to-T2  ...
```

Sería posible generalizar alguno de estos movimientos mediante el uso de variables multi-valuadas:

- Se podría escribir una única regla para mover un disco de la torre T1 a cualquiera de las otras dos torres (a su derecha)
- Se podría escribir una única regla para mover un disco de la torre T3 a cualquiera de las otras dos torres (a su izquierda)

La regla 'final' sería la misma que la que se muestra en la transparencia 9.

5. El problema de las torres de Hanoi (SBR 2)



6. El problema del ascensor

Un ascensor se mueve por un edificio de 4 plantas. El ascensor puede moverse de una planta a otra planta cualquiera:

- Juan está en la planta 1 y quiere ir a la planta 3;
- María está en la planta 4 y quiere ir a la planta 1
- El ascensor está inicialmente en la planta 2.

Objetos del problema y relaciones entre objetos:

- plantas: F1, F2, F3, F4
- Ascensor: E
- Personas: María (M) y Juan (J)
- El ascensor, E, está siempre en una planta
- Una persona puede estar dentro del ascensor (E) o en una planta (F1, F2, F3, F4)

Es importante identificar la **información estática** del problema; por ejemplo:

- Plantas a las que puede moverse el ascensor
- Destino de las personas

6. El problema del ascensor

Patrones:

(can_move E fi^s fj^s) fi^s $fj^s \in \{F1,F2,F3,F4\}$, $fi^s \neq fj^s$ *(información estática)*

(destination per^s f^s) $per^s \in \{M,J\}$, $f^s \in \{F1,F2,F3,F4\}$ *(información estática)*

(elevator E $site^s$ [person per^s $sitp^s$]^m) ;; patrón que representa el estado del problema

$site^s \in \{F1,F2,F3,F4\}$, $per^s \in \{M,J\}$, $sitp^s \in \{F1,F2,F3,F4,E\}$

Las plantas también se pueden representar con números

Hechos:

(can_move E F1 F2) (can_move E F1 F3)(can_move E F1 F4)
(can_move E F2 F3)(can_move E F2 F4)(can_move E F3 F4)
(destination J F3)(destination M F1)

(elevator E F2 person M F4 person J F1) ;; el ascensor está en la planta F2, María está en F4 y Juan está en F1

```
(deffacts datos
    (elevator E F2 person M F4 person J F1)
    (can_move E F1 F2) (can_move E F1 F3) (can_move E F1 F4) (can_move E F2 F3) (can_move E F2 F4)
    (can_move E F3 F4) (destination M F1) (destination J F3))
```

```
(defrule move-elevator
    (elevator E ?sit $?rest)
    (or (can_move E ?sit ?dest)(can_move E ?dest ?sit))
  =>
    (assert (elevator E ?dest $?rest)))
```

```
(defrule board-person
    (elevator E ?sit $?x person ?pers ?sit $?y)
  =>
    (assert (elevator E ?sit $?x person ?pers E $?y)))
```

Regla general para subir a una persona en un ascensor.
Diseño más inteligente: evitar subir a una persona en un ascensor cuando la persona está en su destino

```
(defrule debark-person
    (elevator E ?sit $?x person ?pers E $?y)
  =>
    (assert (elevator E ?sit $?x person ?pers ?sit $?y)))
```

Regla general para bajar una persona en una planta cualquiera.
Diseño más inteligente: bajar a una persona solo cuando el ascensor está en la misma planta que su destino

```
(defrule final
    (declare (salience 100))
    (elevator E ? person ?p1 ?d1 person ?p2 ?d2)
    (destination ?p1 ?d1)
    (destination ?p2 ?d2)
  =>
    (printout t " Solution found " crlf)
    (halt))
```