Block 1 – Knowledge Representation and Search

Chapter 3:
State-based representation through RBS

Block1: Chapter 3- Index

1. State-based representation
2. Design of state-based problems through RBS
3. Problem-solving by using RBS
4. The water jugs problem
5. The hanoi towers problem
6. The elevator problem

Bibliography

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

1. State-based representation

Problem-solving in AI is usually modelled as a **search** process in a **state space**. We consider a single problem-solving agent. Problems are solved by finding a sequences of states that leads to a solution.

Problem definition:

1. Represent the set of possible states in the problem
2. Specify the initial state
3. Specify the goal state or some goal test
4. Transition rules or move operators to move from one problem state to another (actions)

Action: transition that when applied in a state **s** returns a new state as a result of doing the action in **s**.

State space: Set of all states reachable from the initial state by any sequence of actions.

Goal: Find the sequence of operators or actions (solution) which being applied to the initial state leads to the goal state. This is done by searching in the state space.

Environment: *observable* (the agent knows the current state), *discrete* (finite number of actions), *known* (the agent knows which states are reached by each action), *deterministic* (each action has exactly one outcome)

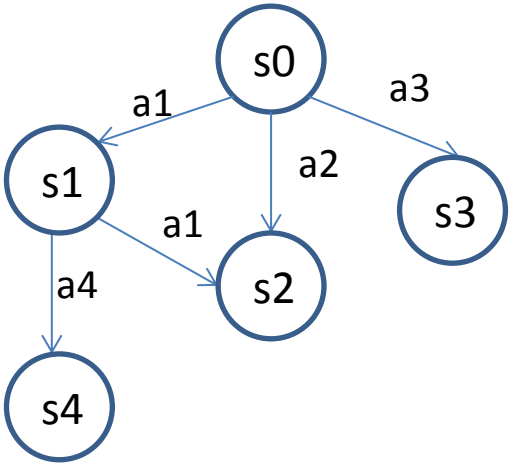
1. State-based representation: problem formulation

Initial state: initial state of the problem (s0)

Actions: actions available to the agent , actions that can be executed in the problem. For example: {a1,a2,a3,a4}

Actions(s) returns the set of actions applicable in s, i.e., the actions that can be executed in s.

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



Transition model: Result(s,a) returns the state that results from doing action a in state s.

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

The initial state, actions and transition model implicitly define the state space
The state space forms a directed network or graph
Nodes represent the problem states, edges represent the problem actions
A path in the state space is a sequence of states connected by a sequence of actions

1. State-based representation: problem formulation

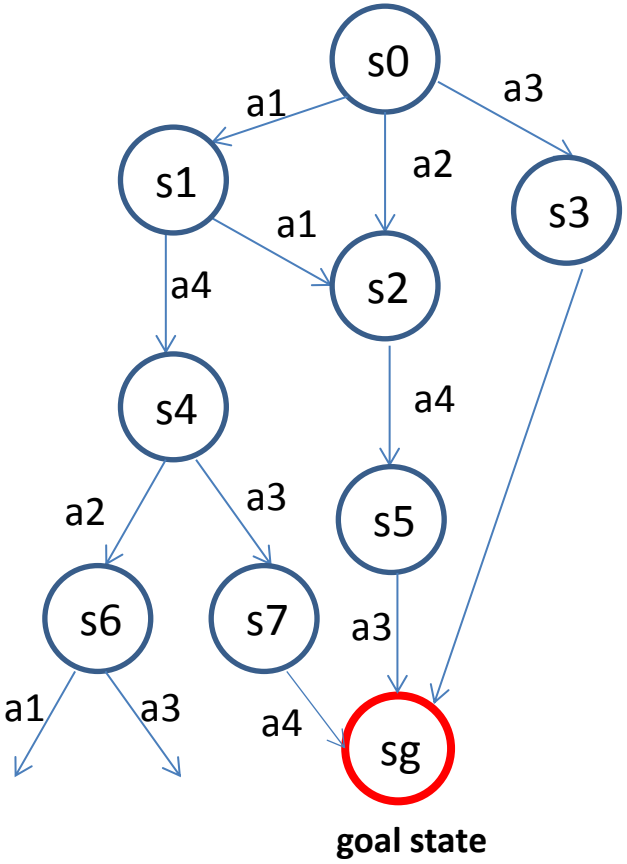
Goal test: determines whether a given state is a goal state or not

- define a particular state as the goal state or
- define an explicit set of goal states or
- define a test that checks whether a given state is a goal state

Path cost function that assigns a numeric cost to each path

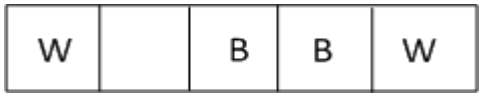
- the step cost of taking action **a** in state **s** to reach **s'** is denoted by $c(s,a,s')$; usually, the cost of an action is the same independently of the state in which the action is applied, i.e, $c(s0,a1,s1)=c(s1,a1,s2)$.

- the cost of a path is described as the sum of the costs of the individual actions along the path



2. Design of state-based problems through RBS

We want to solve the linear puzzle from chapter 2 as a **state-based search** process through a RBS.



Initial state

Final state

We first analyze the knowledge representation in slides 7 and 8 of chapter 2:

1. We use several facts to represent the puzzle board
2. Rules include a `retract` command in the RHS and eliminate the facts that match the patterns in the LHS. As a consequence of this elimination, the remaining rule instances of the Agenda are eliminated too and the search tree cannot be generated.

Working memory

```
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)
```

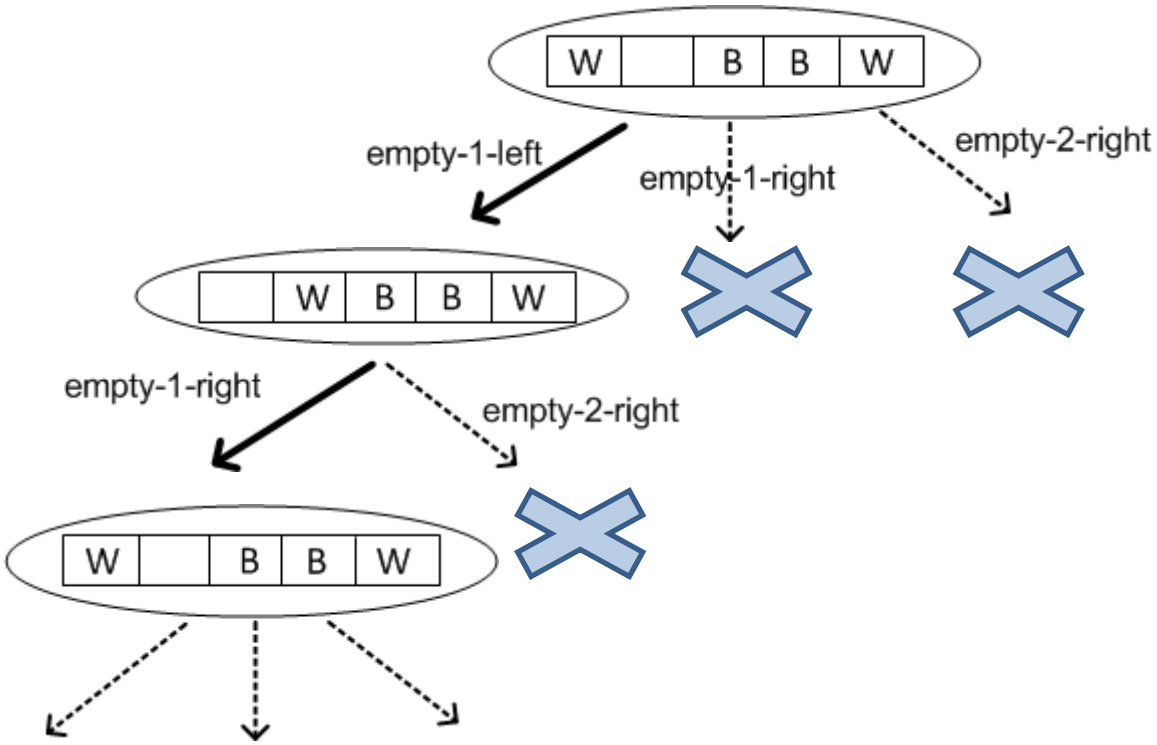
Rule Base

```
(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)))

(defrule emtpy-2-left
    ?f1 <-  (cell ?x E)
    ?f2 <- (cell ?y ?cell)
    (test (= ?y (- ?x 2)))
  =>
    (retract ?f1 ?f2)
    (assert (cell ?y E))
    (assert (cell ?x ?cell)))

. . . . .
```

2. Design of state-based problems through RBS



WM

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)

The fact f-2: (cell 2 E) is retracted and, consequently, the remaining instances are eliminated too.

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)

WM

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)

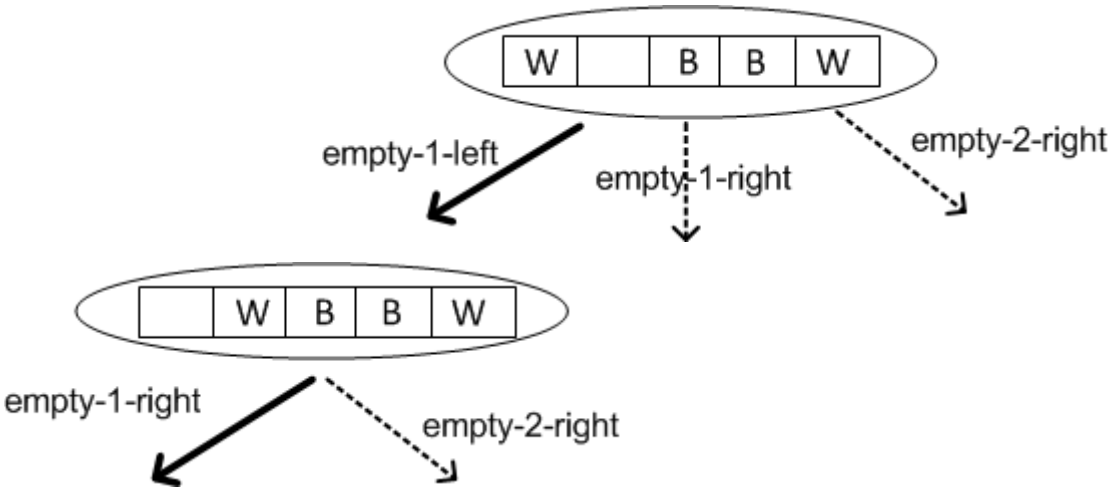
WM

OBJECTIVE: generate the search tree of the problem until finding a node that contains the final situation ➔ rules MUST NOT delete facts that prevent the search process from generating the tree ➔ DO NOT USE **retract**

IMPORTANT: the nodes of the search tree (problem states) must be consistently represented in the WM.

2. Design of state-based problems through RBS

Let's see what happens if we eliminate the retract command of the rules.



WM

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)

WM

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)

inconsistent states

The WM stores all of the facts asserted by the rules but we are not able to identify the facts which represent one state or another

Solution: represent a problem state with a single fact

2. Design of state-based problems through RBS

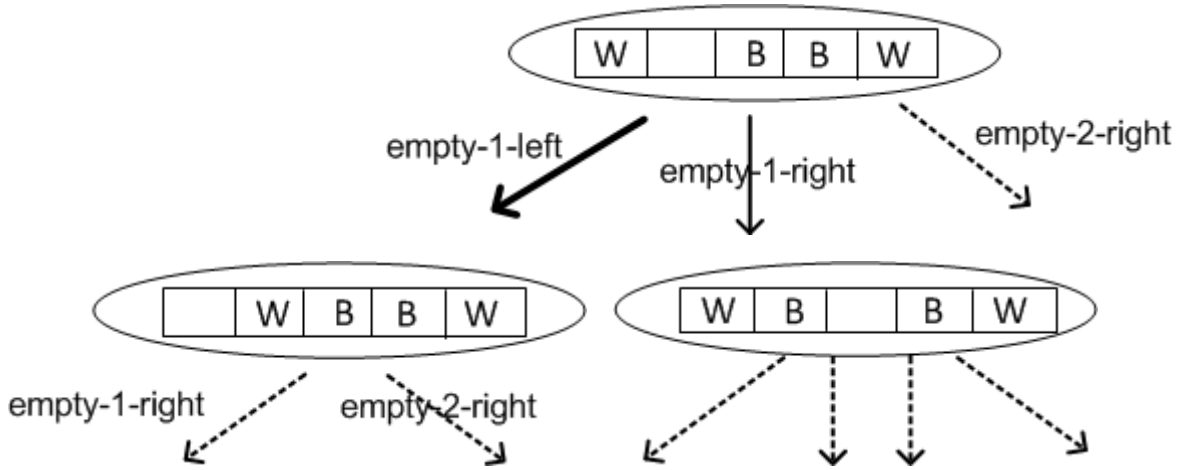
We use now the knowledge representation in slides 5 and 6 of chapter 2 but we eliminate the `retract` command of the rules.

Working memory

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

Rule Base

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



Working Memory

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. Problem solving by using RBS

State representation: patterns and facts

- Use a knowledge representation language that allows to represent patterns, facts and rules.
- Make a RBS design capable to represent the problem states.
- Represent only the information relevant to attain the goal.
- A good design should allow for an easy and clear identification of the key components of the problem states as well as facilitating the design of the LHS and RHS of the rules.

1. Dynamic information
   - Data that may change as a result of the actions execution.
   - Maintain dynamic information in a single fact such that the fact represents a particular problem situation (problem state).
   - Examples: (contents jug X 2 jug Y 1), (puzzle W E B B W).

2. Static information
   - Data that do not change along the problem-solving evolution.
   - Static information can be represented with as many facts as desired
   - Examples: (capacity jug X 4), (road cityA cityB)

2. Problem solving by using RBS

Problem actions representation: rules

- Rules denote the actions that can be performed in the problem to modify a problem state (problem operators).
- An action is represented with a rule.

Characteristics:

Completeness: the designed rules must guarantee to find a solution for a solvable problem. .

1. The rule base must contain all possible actions that are applicable in the problem
2. The LHS of the rules should only represent the conditions that are necessary to apply the rule
3. The RHS of the rules must represent all of the changes produced by the corresponding action.

Correctness: the designed rules must properly represent the changes in the problem states.

1. The LHS of the rules should express all of the conditions necessary to apply the rules
2. The RHS of the rules should only represent the changes produced by the corresponding action.

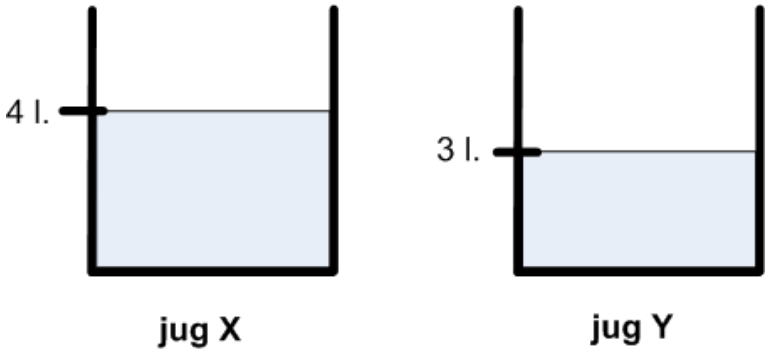
3. Problem solving by using RBS

Conflict resolution:

The choice of the conflict resolution strategy is a key issue for an efficient problem resolution

1. Choose an adequate conflict resolution strategy
2. Keep in mind that rules interact and the order of rule firing ***matters***.
3. It is important not only the design of the rules but also when the rules will be fired

4. The water jugs problem



- Maximum capacity of jug X is 4 l.
- Maximum capacity of jug Y is 3 l.
- *Initially, both jugs are empty*
- *Goal*: to have 2 l. in jug X
- No measure marks except maximum capacity

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

    (contents jug X 0 jug Y 0)
)

→ Static information, never changes

→ Initial state

Actions:

Fill up jug X
Fill up jug Y
Empty jug X
Empty jug Y
Fill X from Y
Fill Y from X
Empty Y into X
Empty X into Y

4. The water jugs problem

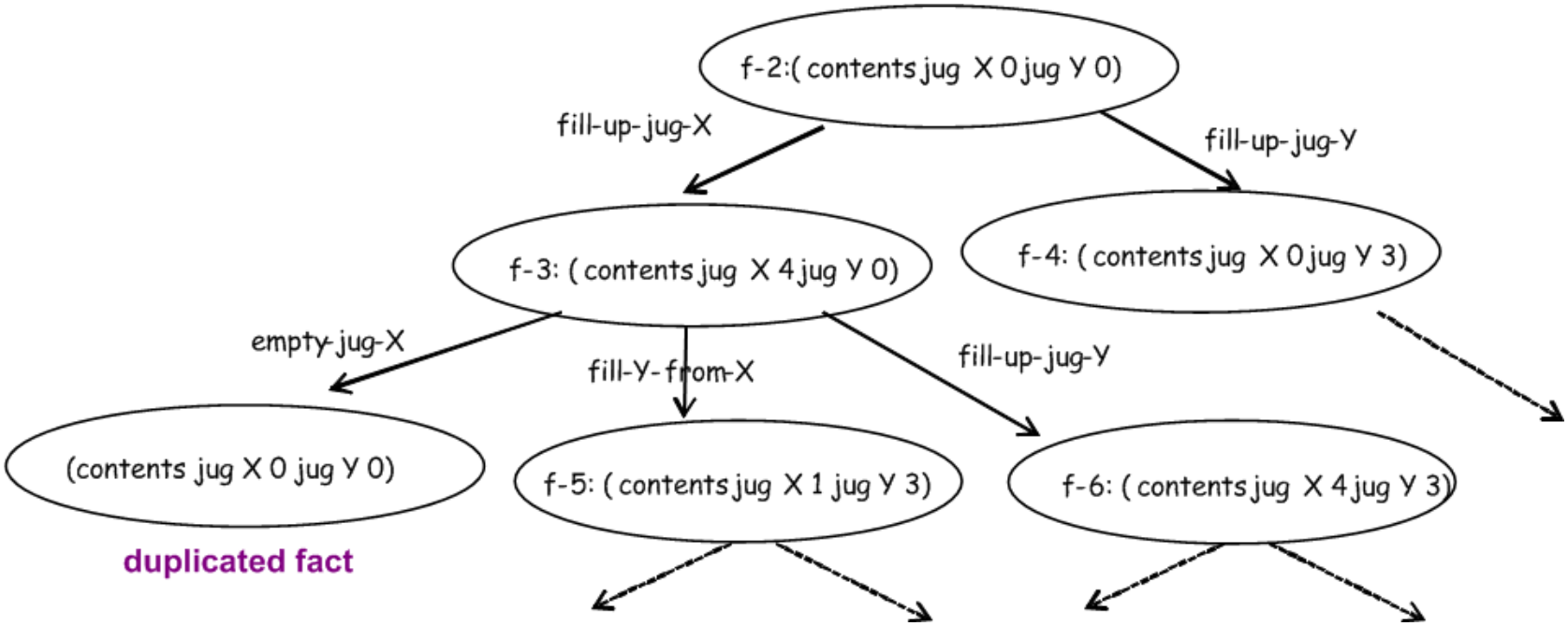
```
(defrule fill-up-jug-X
      (capacity jug X ?cap)
      (contents jug X ?x jug Y ?y)
      (test (< ?x ?cap))
 =>
      (assert (contents  jug X ?cap jug Y ?y)))
```

```
(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)))
```

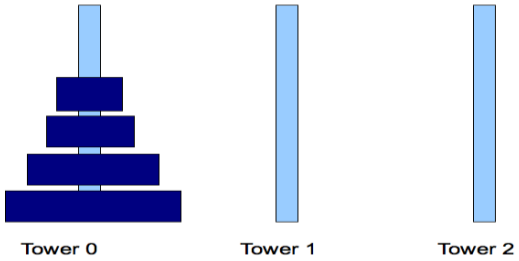
4. The water jugs problem



WM={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. The hanoi towers problem

Design for four disks and three towers



Identify objects and relationships between objects:

- Towers (T1, T2, T3)
- Disks (D1, D2, D3, D4) such that Di > Dj if i > j. Disks can be represented with symbols (Di) or with a numeric value that indicates its size.
- Disks in a tower

5. The hanoi towers problem

Pattern to represent the problem states:

(hanoi [tower tw^s dk^m]^m) ;; valid for any number of disks and towers

- superscript shows the type of variable (s-single-valued, m-multi-valued)
- tw^s is the tower identifier, $tw^s \in$ {T1,T2,T3}
- dk^m: multi-valued variable that represents the disks in a tower, $dk^m \in 2^{\{1\text{-}4\}}$.
- dk^m represents a set of zero, one, two, three or four items from the set {1-4} .
  Examples: (1 3), (2 3 4), (), (1 4)
- [tower tw^s dk^m] represents the information of tower tw^s
- [tower tw^s dk^m]^m represents any number of towers

Facts:

Initial state of the problem: WMinitial= {(hanoi towerT1 1 2 3 4 tower T2 towerT3)}

Final state of the problem: WMfinal= {(hanoi towerT1 towerT2 towerT3 1 2 3 4)}

Examples of the space state (problem states):

s1={(hanoi towerT1 3 4 towerT2 1 towerT3 2)}

s2={(hanoi towerT1 4 towerT2 2 tower T3 1 3)}

s3 ={(hanoi towerT1 1 towerT2 2 4 tower T3 3)}

5. The hanoi towers problem

```
(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 (member tower $?rest2))
     (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. The hanoi towers problem (another representation – RBS 2)

We can also choose a more uniform pattern (only valid for 4 disks):

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

Facts:

(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)

Rules: we can generalize the rule **move-disk** regardless whether the destination tower is empty or not

```
(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. The hanoi towers problem (RBS 2)

Because facts are ordered facts, i.e. elements in a fact are referenced positionally , we have to write 6 rules:

```
(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  …
```

It would be possible to define some general moves through the use of multi-field variables:

- We could write a single rule to move a disk from tower T1 to any of the two towers on its right.
- We could write a single rule to move a disk from tower T3 to any of the two towers on its left.

The 'final' rule would be the same as the one shown in slide 9.

5. The hanoi towers problem (RBS 2)

The execution of the RBS solves a state-based problem generating internally a **search tree**.

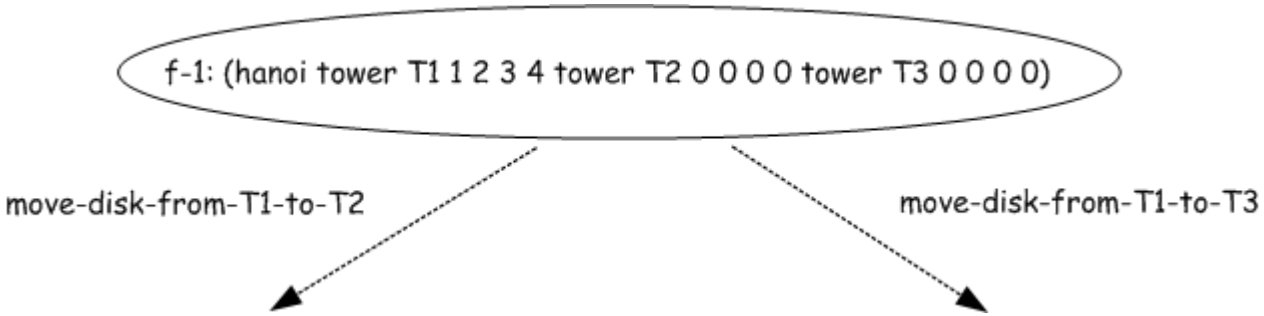
Trace of Hanoi towers RBS 2 assuming a **depth-first strategy** of the agenda.

RB= {move-disk-from-T1-to-T2, move-disk-from-T1-to-T3, move-disk-from-T2-to-T1, …}

WM_initial= {f-1: (hanoi tower T1 1 2 3 4 tower T2 0 0 0 0 tower T3 0 0 0 0)}

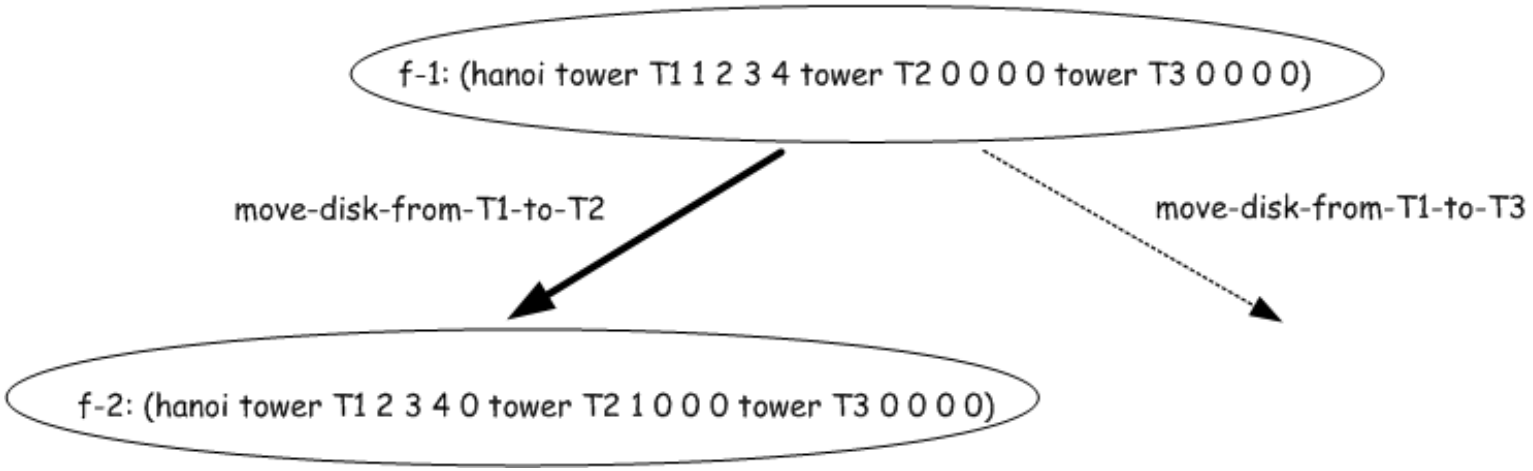
NOTE: rules do not have a (retract …) command in the RHS because otherwise we would not be able to generate the search tree.

1. Matching



5. The hanoi towers problem (RBS 2)

2-3. Selection and Execution



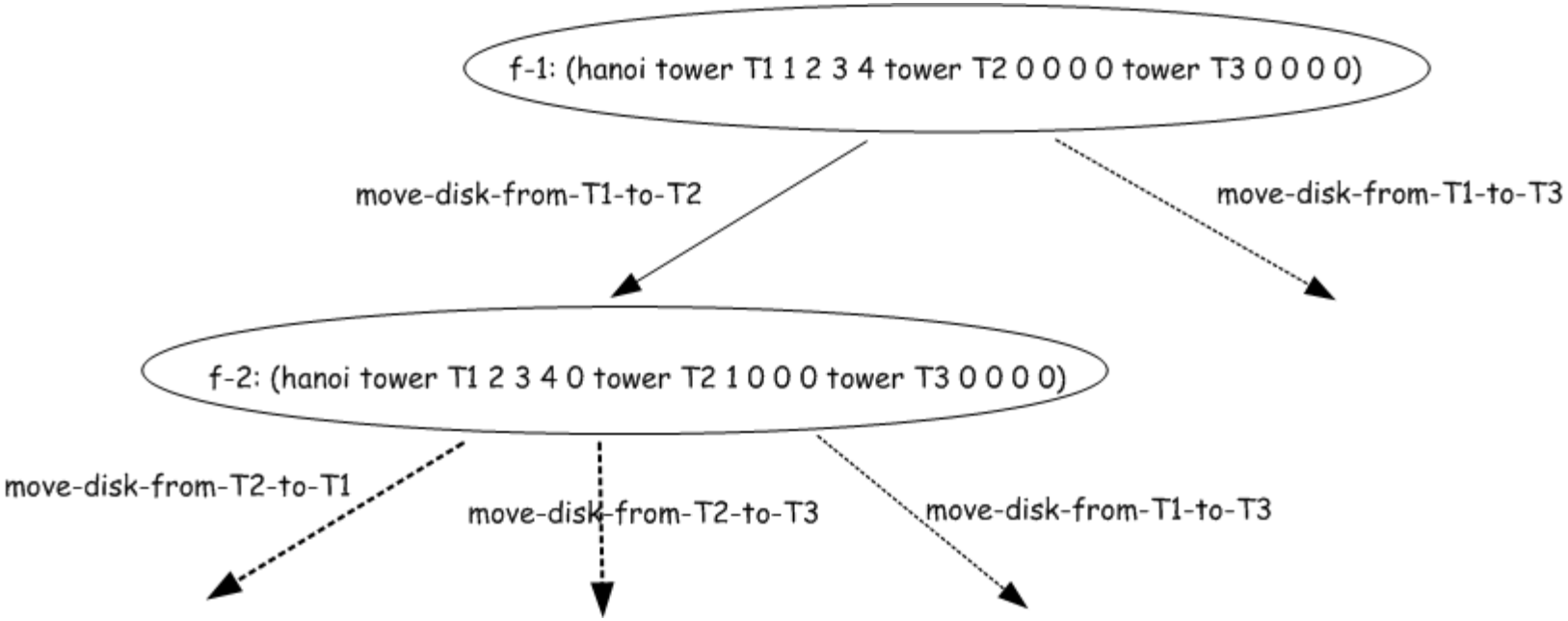
WM= {f-1: (hanoi tower T1 1 2 3 4 tower T2 0 0 0 0 tower T3 0 0 0 0)
f-2: (hanoi tower T1 2 3 4 0 tower T2 1 0 0 0 tower T3 0 0 0 0)}

- The rule instance **move-disk-from-T1-to-T3** remains pending in the Agenda or Conflict Set.

- If the rule instance **move-disk-from-T1-to-T2** would retract the fact f-1 when executed then the rule instance **move-disk-from-T1-to-T3** would be eliminated from the Agenda, thus preventing the corresponding node from being generated.

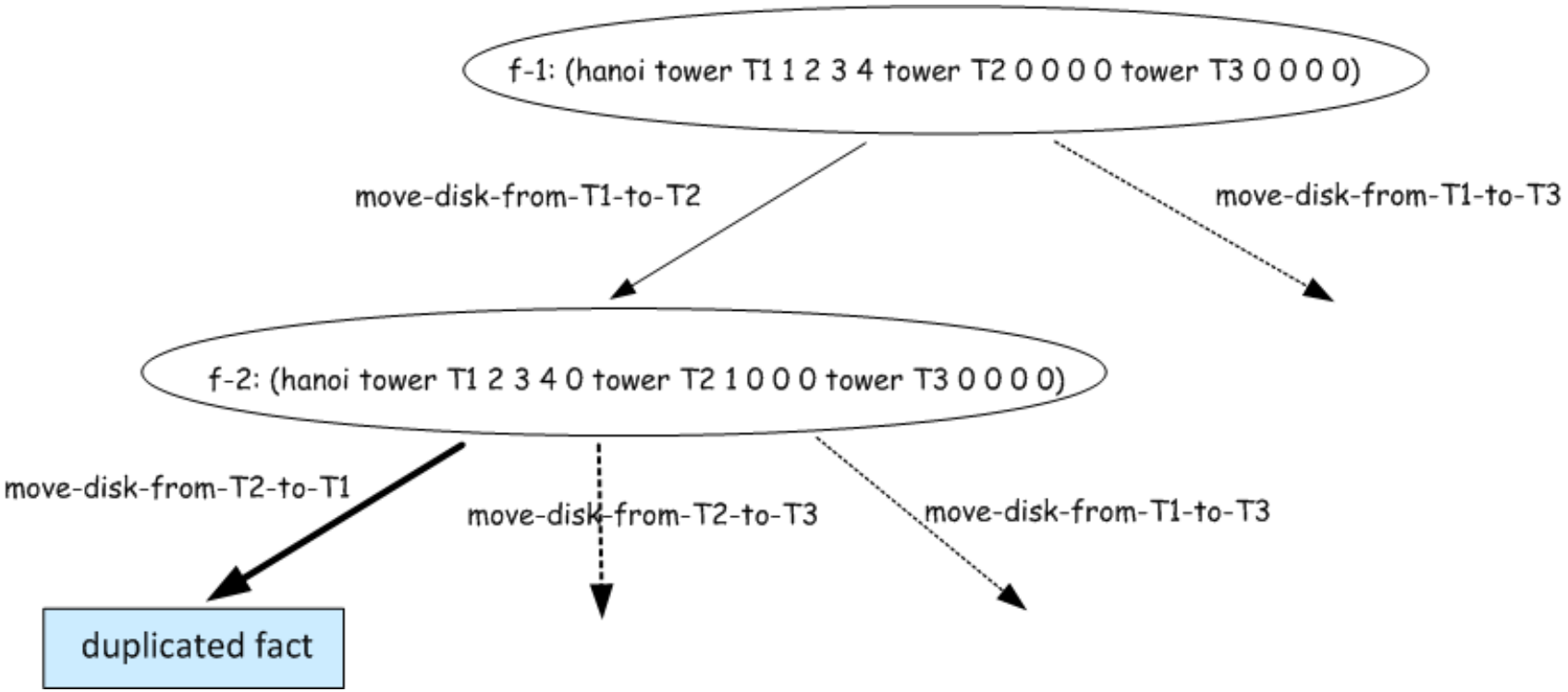
5. The hanoi towers problem (RBS 2)

1. Matching



5. The hanoi towers problem (RBS 2)

2-3. Selection and Execution

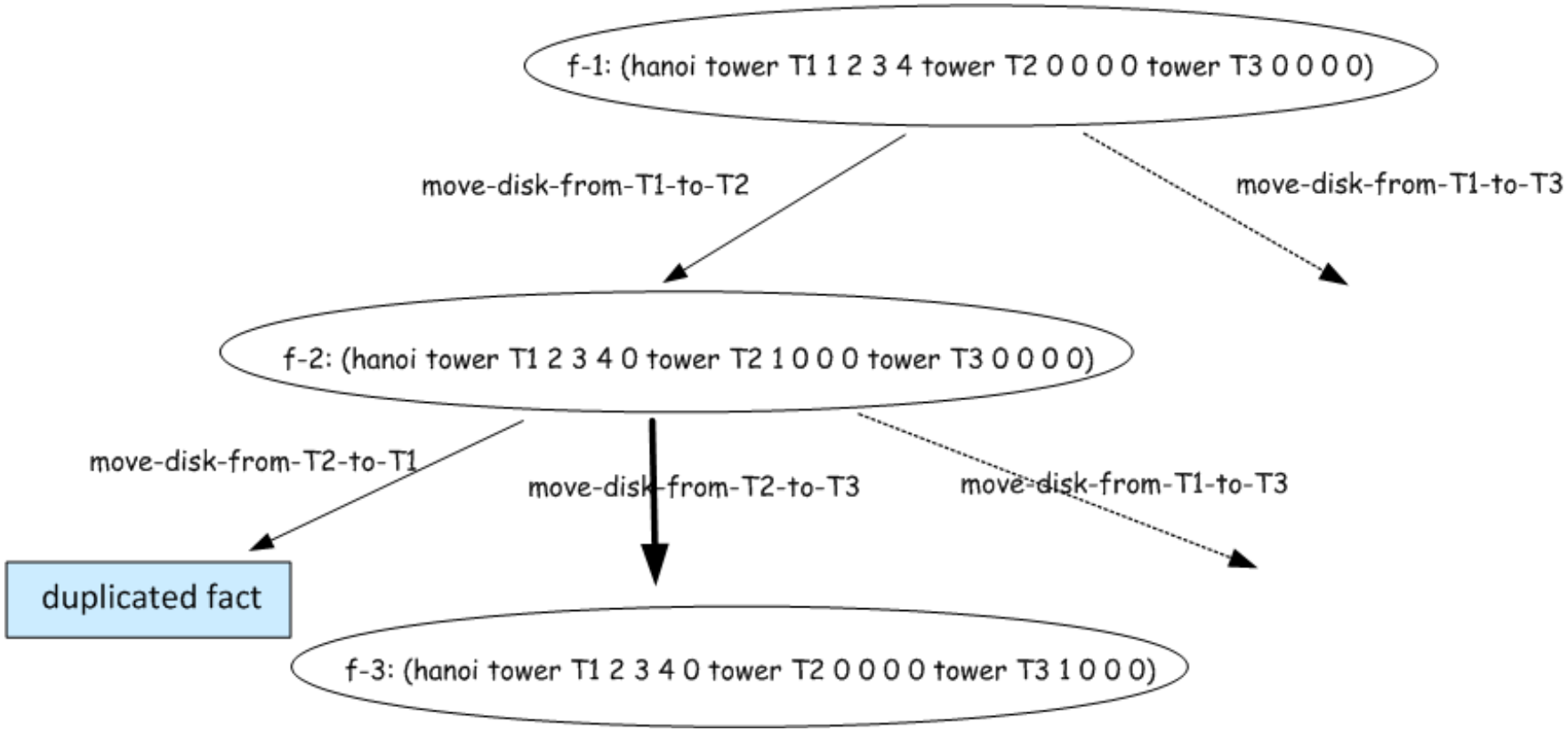


BH= {f-1: (hanoi tower T1 1 2 3 4 tower T2 0 0 0 0 tower T3 0 0 0 0)
f-2: (hanoi tower T1 2 3 4 0 tower T2 1 0 0 0 tower T3 0 0 0 0)}

5. The hanoi towers problem (RBS 2)

1. Matching: no new rule instances because there are no new facts in the WM

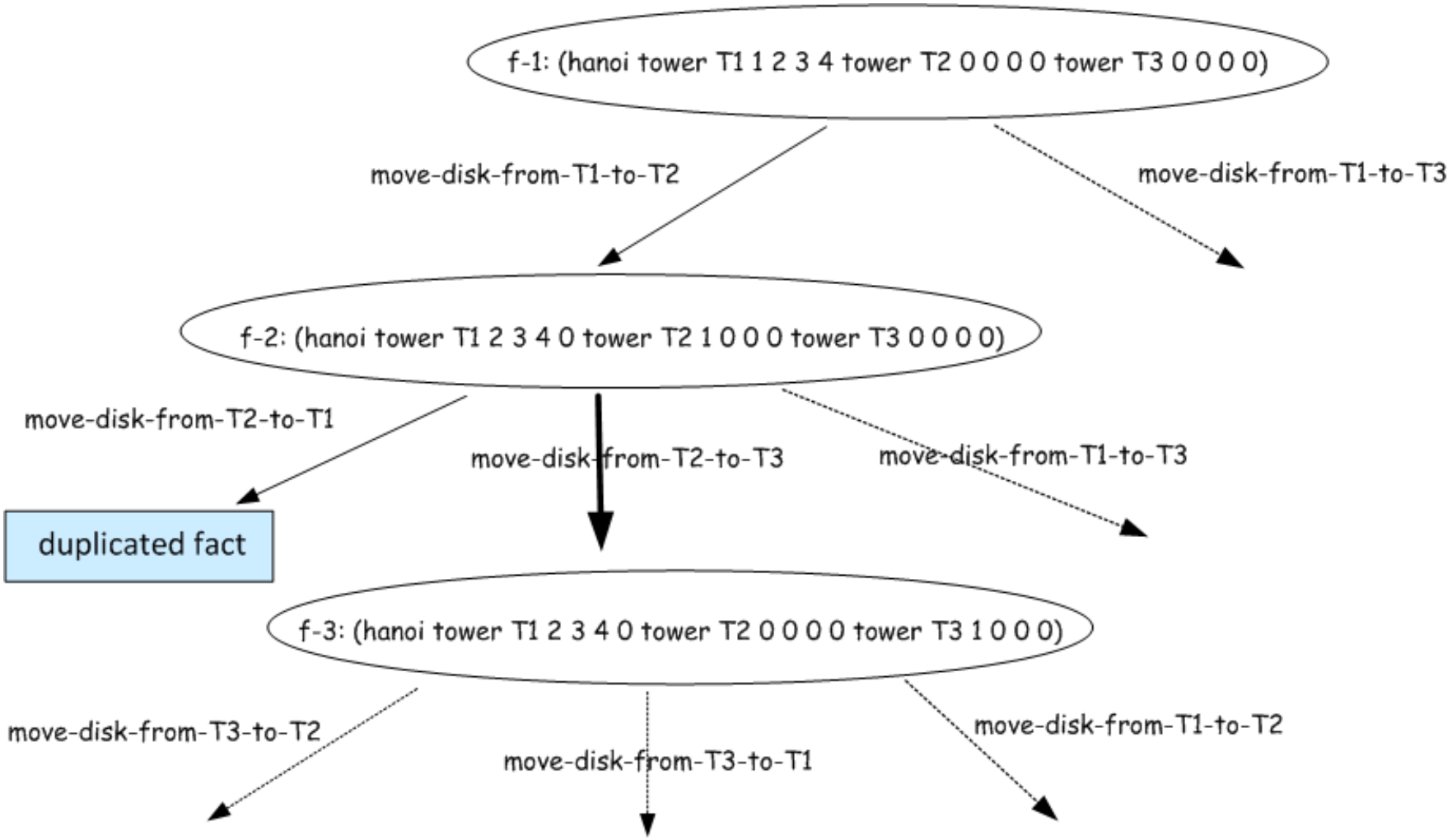
2-3. Selection and Execution



BH= {f-1: (hanoi tower T1 1 2 3 4 tower T2 0 0 0 0 tower T3 0 0 0 0)
f-2: (hanoi tower T1 2 3 4 0 tower T2 1 0 0 0 tower T3 0 0 0 0)
f-3: (hanoit tower T1 2 3 4 0 tower T2 0 0 0 0 tower T3 1 0 0 0)}

5. The hanoi towers problem (RBS 2)

1. Matching



6. The elevator problem

An elevator moves along a 4-story building. The elevator moves between any two floors of the building.

- John is at floor 1 (F1) and wants to go to floor 3 (F3)
- Mary is at floor 4 (F4) and wants to go to floor 1 (F1)
- The elevator is initially at floor2 (F2)

Problem objects and relationships among them:

- Floors: F1, F2, F3, F4
- Elevator: E
- Persons: Mary (M) and John (J)
- The elevator, E, is always at a floor
- A person can be inside the elevator (E) o at a floor (F1, F2, F3, F4)

It is important to define the **static information of the problem**; for instance:

- Floors to which the elevator can move
- People destination floor

6. The elevator problem

Patterns:

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

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

(elevator E $site^s$ [person per^s $sitp^s$]^m) ;; pattern to represent the problem state

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

Floors can also be represented via numbers

Facts:

(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) ;; the elevator is at F2, Mary is at F4 and John is at 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)))
```

General rule to board a person in the elevator.
More intelligent design: avoid boarding a person when is already at destination

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

General rule to debark a person at any floor.
More intelligent design: only debark the person when is at destination

```
(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))
```