

Chapter 6

Control and implementation of state space search

(状态空间搜索的控制和实现)



6 Control and implementation of state space search

6.0 Introduction

6.1 Recursion-based search

6.1.1 Recursion search

6.1.2 A Recursive Search Example: Pattern - Driven Reasoning

6.2 Production systems

6.2.1 Definition and History

6.2.2 Examples of Production System

6.2.3 Control of Search in Production System

6.2.4 Advantages of Production System for AI

6.3 The blackboard architecture for problem solving



6.0 Introduction

To summarize, part II has:

1. Represented **a problem solution** as **a path** in a graph from a **start** state to a **goal**.
2. Used search to test **systematically** (系统化) **alternative paths** to goals.
3. Employed **backtracking**, or some other **mechanism**, to allow algorithms to **recover** (恢复) from paths that failed to find a goal.



4. Used **lists** to keep **records** of states :

- OPEN list : **untried states**
- CLOSED list : **tried states**, for **loop detection** (检测) and avoiding fruitless paths.

5. Implemented the OPEN list :

- as a **stack** for **depth-first** search,
- as a **queue** for **breadth-first** search,
- As a **priority queue** (优先队列) for **best-first** search.



6.1 Recursion-Based Search

6.1.1 Recursion search

- A recursive (递归的) procedure consists of:
 1. A **recursive step** : the procedure **calls itself** to repeat **a sequence of actions**.
 2. A **terminating condition** : that **stops** the procedure **from** recurring (递归) endlessly .



- Definition

propositional calculus **sentence**

- Every **propositional symbol** and **truth symbol** is a sentence. (true, P, Q ...)
- The **negation** (非, 否定) of a sentence is a sentence. ($\neg P$, $\neg \text{true}$)
- The **conjunction** (与, 合取) of two sentences is a sentence. ($P \wedge \neg P$)
- The **disjunction** (或, 析取) of two sentences is a sentence. ($P \vee \neg P$)



- The **implication** (蕴含) of **one sentence from another sentence** is a sentence. ($P \rightarrow Q$)
- The **equivalence** (等价) of two sentences is a sentence. ($P \wedge Q \equiv R$)
- ◆ **Legal sentences** are also called **well-formed formulas** (合适公式) or **WFFs**.



Function **unify**(E1, E2)

begin

case

both E1 and E2 are **constants** or **the empty list**:

//recursion stops

if E1 = E2 then return { };

else return fail;

E1 is a variable:

if E1 **occurs** in E2 then return fail

else return { E2 / E1 } ;

E2 is a variable:

if E2 **occurs** in E1 then return fail

else return { E1 / E2 } ;

either E1 or E2 are **empty** then return fail

//the list are of different size

otherwise:

//both E1 and E2 are list

begin

HE1:=**first** element of E1;

HE2:=**first** element of E2;

S1:=**unify**(HE1, HE2);

if S1:=fail then return fail;

TE1:=apply(S1, **rest** of E1);

TE2:=apply(S1, **rest** of E2);

S2:=**unify**(TE1, TE2);

if S2 = fail then return fail;

else return composition(S1, S2)

end

end

end



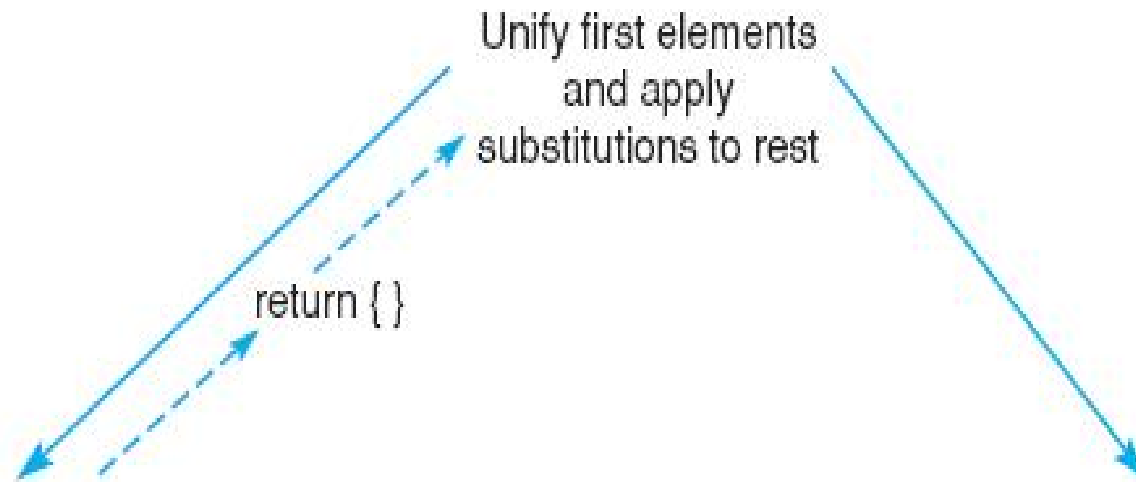
2.3.3 A Unification Example

- Example:

➤ Unify ((parents **X** (father X) (mother bill)) ,
(parents **bill** (father bill) **Y**))

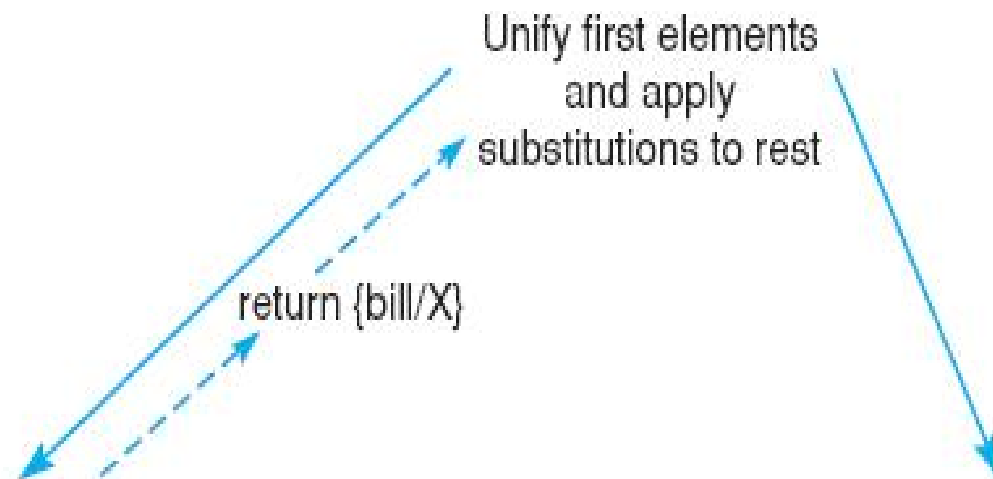


1. unify((parents X (father X) (mother bill)), (parents bill (father bill) Y))



2. unify(parents, parents)

3. unify((X (father X) (mother bill)), (bill (father bill) Y))



4. unify(X, bill)

5. unify(((father bill) (mother bill)), ((father bill) Y))

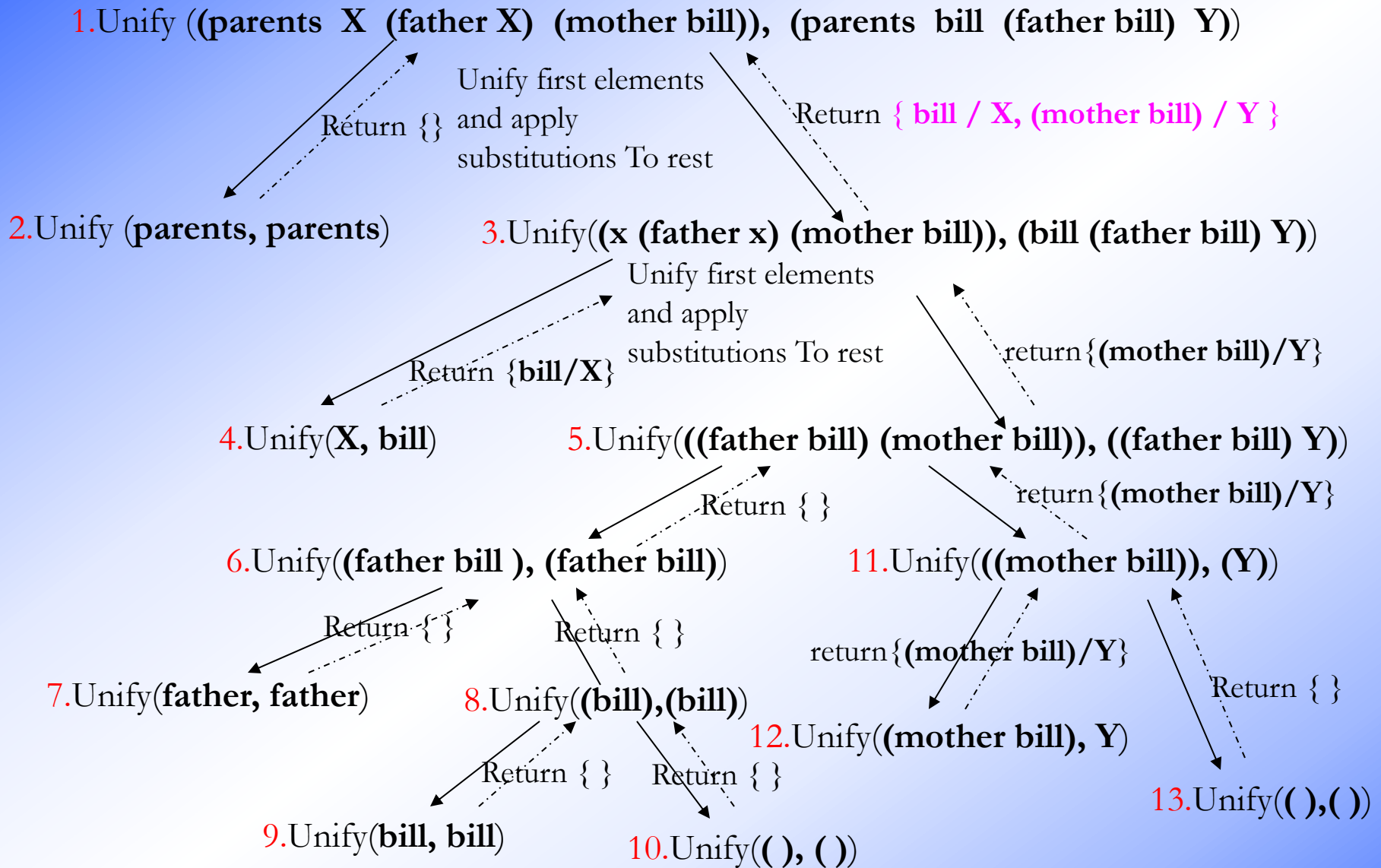
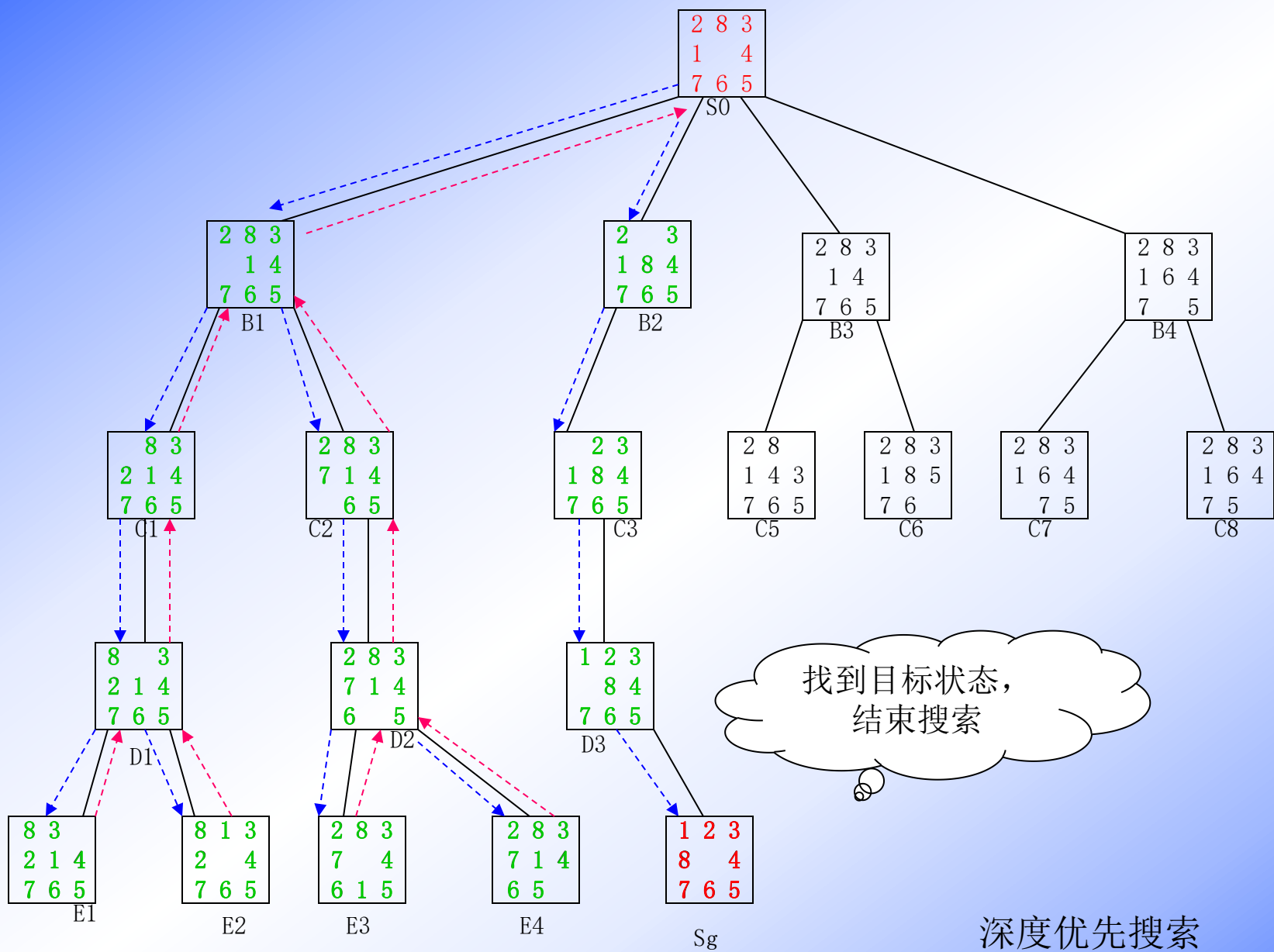


Figure 2.6 full trace of the unification





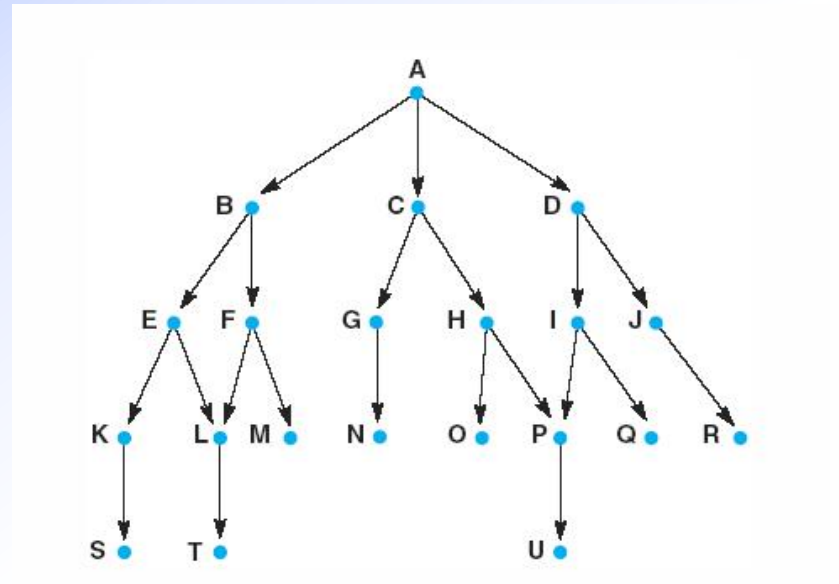
深度优先搜索



```
Function depth_first_search;  
begin  
  open := [ start ] ;  
  closed := [ ] ;  
  While open ≠ [ ] do  
    begin  
      remove leftmost state from open ,call it X;  
      if X is a goal then return SUCCESS  
      else begin  
        generate children of X;  
        put X on closed;  
        discard children of X if already on open or closed;  
        put remaining children on left end of open  
      end  
    end  
  return FAIL  
end.
```



- 1 open=[**A**]; closed=[]
- 2 open=[**B,C,D**]; closed=[**A**]
- 3 open=[**E,F,C,D**]; closed=[**B,A**]
- 4 open=[**K,L,F,C,D**]; closed=[**E,B,A**]
- 5 open=[**S,L,F,C,D**]; closed=[**K,E,B,A**]
- 6 open=[**L,F,C,D**]; closed=[**S,K,E,B,A**]
- 7 open=[**T,F,C,D**]; closed=[**L,S,K,E,B,A**]
- 8 open=[**F,C,D**]; closed=[**T,L,S,K,E,B,A**]
- 9 open=[**M,C,D**]; closed=[**F,T,L,S,K,E,B,A**]
- 10 open=[**C,D**]; closed=[**M,F,T,L,S,K,E,B,A**]
- 11 open=[**G,H,D**]; closed=[**C,M,F,T,L,S,K,E,B,A**]
- 12 and so on **until** either U is found **or** open=[]



```

Function depthsearch;                                % open & closed global
Begin
  If open is empty Then return FAIL;
  current_state := the first element of open;
  If current_state is a goal state
  Then return SUCCESS                                % output SUCCESS
  Else
    Begin
      open := the tail of open;
      close := closed with current_state added;
      For each child of current_state
        If not on close or open
          Then add the child to the front of open    % stack
    End;
  Depthsearch      % tail recursion
End.

```




```
Function depthsearch (current_state)  % closed is global
Begin
  If current_state is a goal
  Then return SUCCESS
  Add current_state to closed;
  While current_state has unexamined children
  Begin
    child := next unexamined child;
    If child is not a member of closed
    Then if depthsearch(child) = SUCCESS
      Then return SUCCESS
    End;
  return FAIL;
End
```



二阶梵塔问题

设有三根柱子，在1号柱子上穿有A、B两个盘片，盘A小于盘B，盘A位于盘B的上面。要求把两个盘片全部移到另一根柱子上，而且规定每次只能移动一片，任何时刻都不能使盘B位于盘A的上面。



设用 $S_k = (S_{k0}, S_{k1})$ 表示问题的状态, S_{k0} 表示盘A所在的柱子号, S_{k1} 表示盘B所在的柱子号。

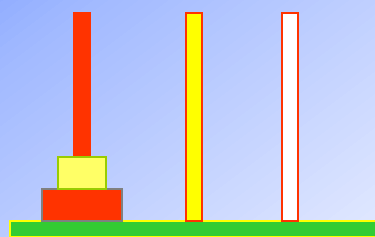
全部可能的状态有以下9种:

$$S_0 = (1,1) \quad S_1 = (1,2) \quad S_2 = (1,3)$$

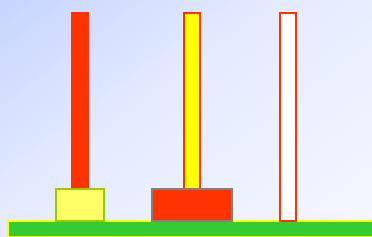
$$S_3 = (2,1) \quad S_4 = (2,2) \quad S_5 = (2,3)$$

$$S_6 = (3,1) \quad S_7 = (3,2) \quad S_8 = (3,3)$$

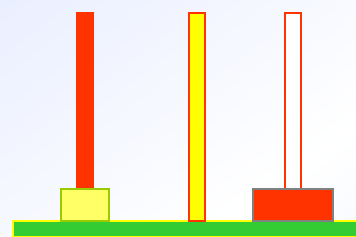




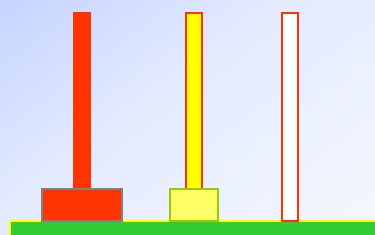
$S_0 = (1, 1)$



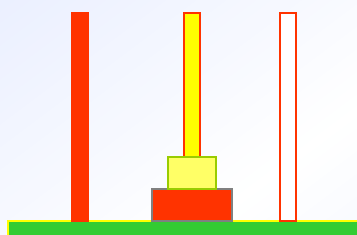
$S_1 = (1, 2)$



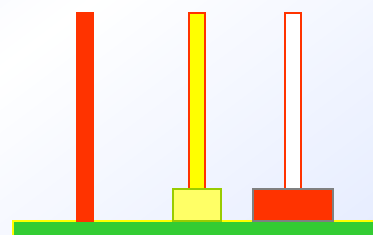
$S_2 = (1, 3)$



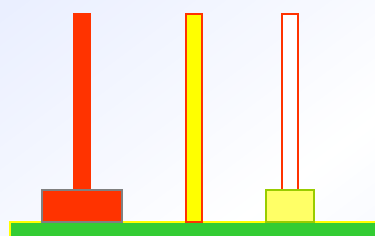
$S_3 = (2, 1)$



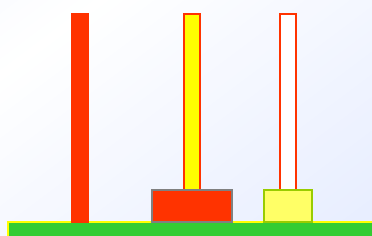
$S_4 = (2, 2)$



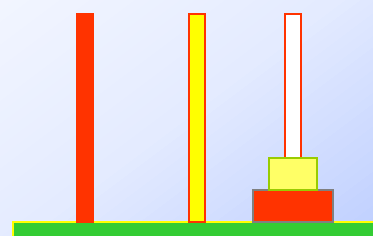
$S_5 = (2, 3)$



$S_6 = (3, 1)$



$S_7 = (3, 2)$



$S_8 = (3, 3)$

二阶梵塔问题状态表示



- 问题的初始状态集合为 $S = \{S_0\}$, 目标状态集合为 $G = (S_4, S_8)$ 。算符分别用 $A(i,j)$ 及 $B(i,j)$ 表示。
 $A(i,j)$ 表示把盘A从i号柱移到j上; $B(i,j)$ 表示把盘B从i号柱移到j号柱上。

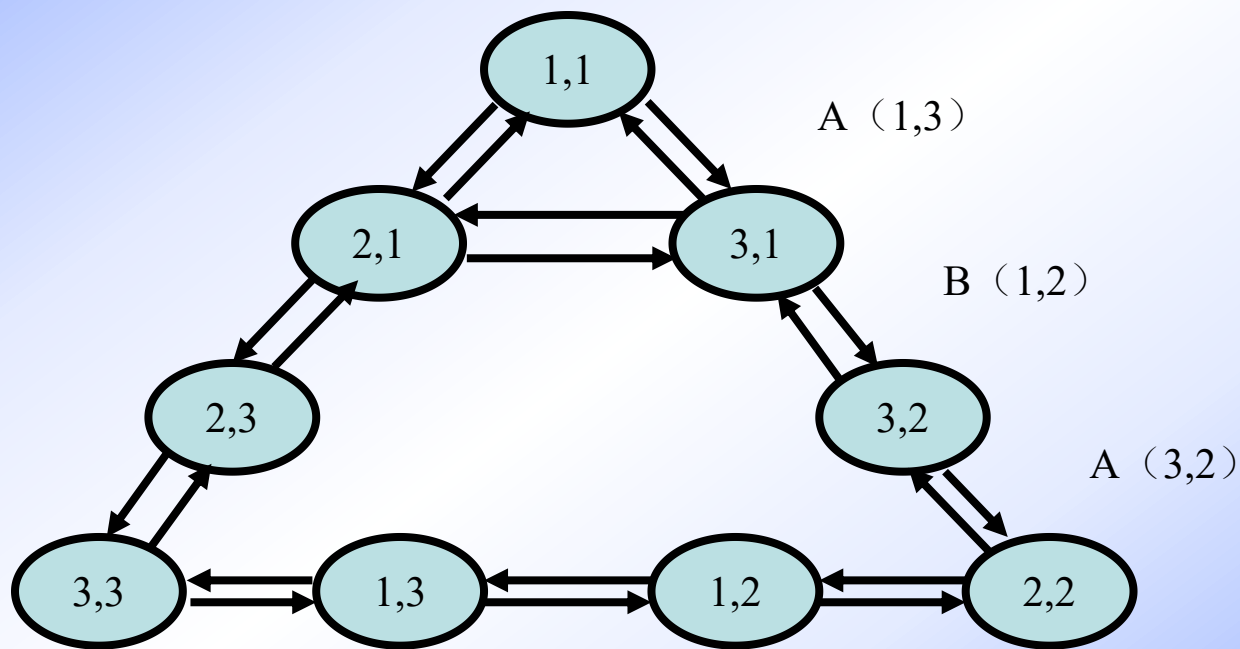
共有12个算符, 它们分别是:

$A(1,2), A(1,3), A(2,1), A(2,3), A(3,1), A(3,2)$

$B(1,2), B(1,3), B(2,1), B(2,3), B(3,1), B(3,2)$



根据9种可能的状态和12种算符，可构成二阶梵塔问题的状态空间图，如图所示。



二阶梵塔问题的状态空间图



三阶梵塔问题。

设有A、B、C三个盘片以及三根柱子，三个盘片按从小到大的顺序穿在1号柱子上，要求把他们全部移到3号柱子上，而且每次只能移动一个盘片，任何时刻都不能把大的盘片压在小的盘片上面。



可把原问题分解为三个子问题：

- 1) 把盘片A及B移到2号柱的双盘片问题。
- 2) 把盘片C移到3号柱的单盘片问题。
- 3) 把盘片A及B移到3号柱的双盘片问题。

其中，子问题1) 与子问题3) 又分别可分解为三个子问题。



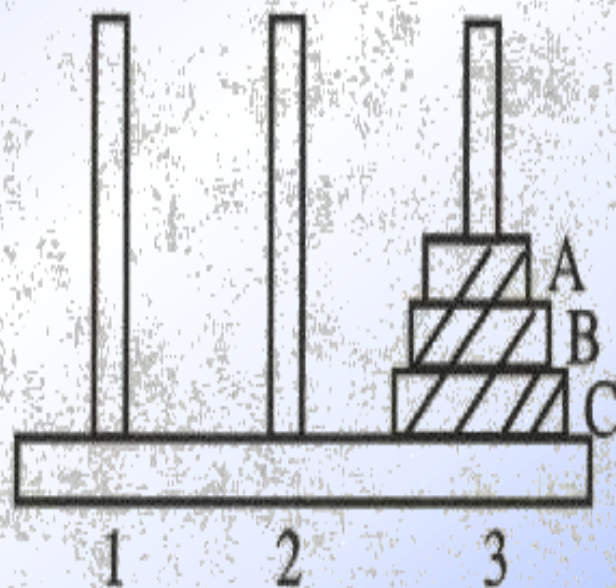
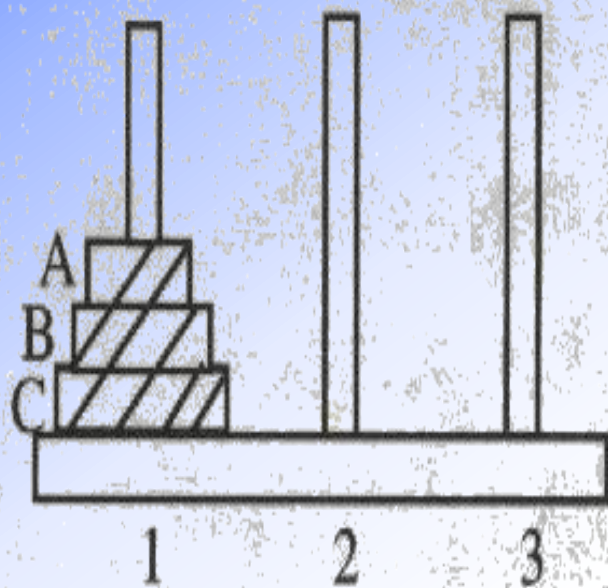


图 4.5 三阶梵塔问题



为了用与/或树把问题的分解过程表示出来，需要先定义问题的状态表示。设用三元组：

$$(i,j,k)$$

表示状态，其中 i 表示盘片 C 所在的柱号； j 表示盘片 B 所在的柱号； k 表示盘片 A 所在的柱号。这样初始问题就可表示为：

$$(1,1,1) \Rightarrow (3,3,3)$$

可用与/或树把分解过程表示出来，如图 4.6 所示。

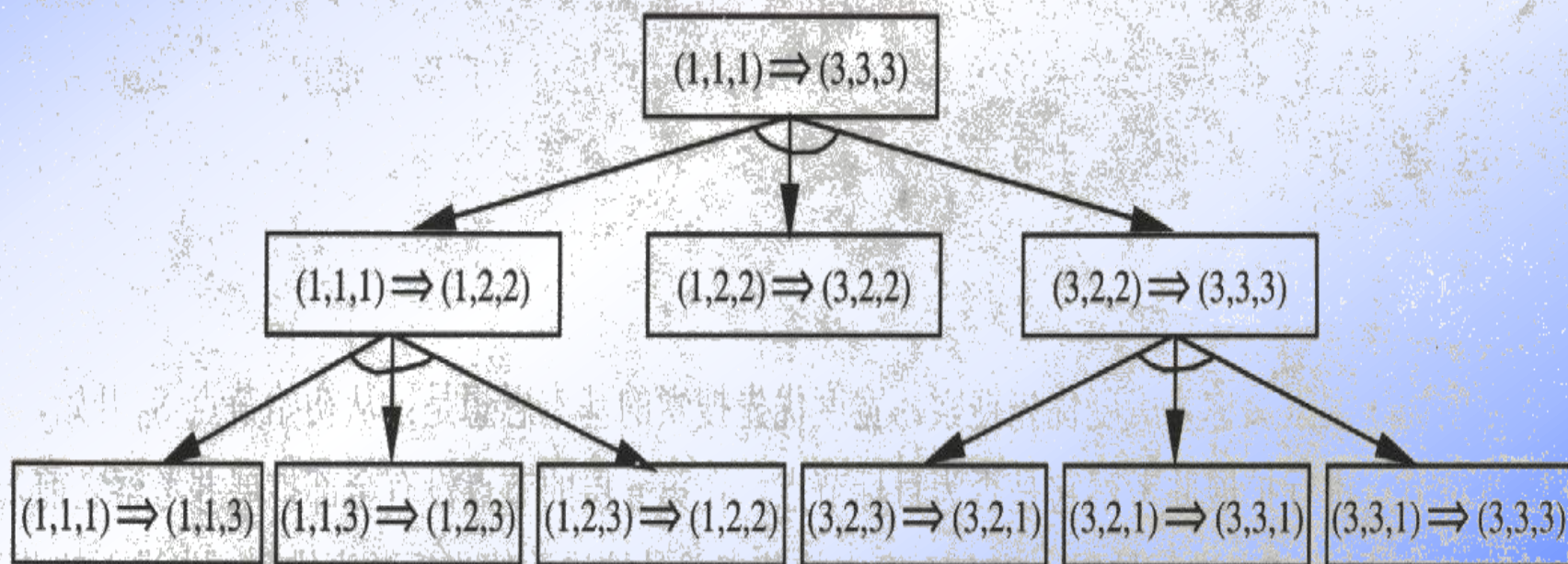


图 4.6 三阶梵塔问题的与/或树

6.1.2 A Recursive Search Example : Pattern - Driven Reasoning

- Suppose we want to write **an algorithm** that determines whether **a predicate calculus expression** is a **logical consequence** (逻辑结论) of some set of **assertions** (断言) .
- Example:
 - Given an assertion: $q(a)$
 - Given an implication (rule) :
$$q(x) \rightarrow p(x)$$
 - Given a goal: $p(a)$?



```

Function pattern_search ( current_goal )
Begin
    If current_goal is a member of closed                                % test for loops
    Then return FAIL
    Else add current_goal to closed;
    While there remain in data base unifying facts or rules do
        Begin
            Case
                current_goal unifies with a fact:
                return SUCCESS;
                current_goal is a conjunction (  $p \wedge \dots$  ) :
                begin
                    for each conjunct do
                        call pattern_search on conjunct;
                        if pattern_search succeeds for all conjuncts
                        then return SUCCESS
                        else return FAIL
                    end;
                current_goal unifies with rule conclusion (  $p \text{ in } q \rightarrow p$  ) :
                begin
                    apply goal unifying substitutions to premise (  $q$  ) ;
                    call pattern_search on premise;
                    if pattern_search succeeds
                    then return SUCCESS
                    else return FAIL
                end;
            end; % end case
        End;
    return FAIL
End

```



- We need a **control regime** (机制) that **systematically** searches the space, avoiding **meaningless** paths and loops.
- A second issue is the **logical connectives** (连接符) in the rule premises:
e.g. $p \leftarrow q \wedge r$; $p \leftarrow q \vee (r \wedge s)$
- An \wedge operator, indicates that **both** expression must be found to be **true**
- An \vee operator, indicates that **either** expression must be found to be **true**
- in addition, the **conjuncts** of the expression must be solved with **consistent** (一致的) **variable bindings**



- The last addition to the algorithm is the ability to solve goals involving **logical negation** (\neg).
- Finally, the algorithm should not **return success** but should **return the bindings** (变量绑定) involved in the solution.




```
function pattern_search ( current_goal )  
begin  
    if current_goal is a member of closed  
    then return FAIL  
    else add current_goal to closed;  
    while there remain unifying facts or rules do  
        begin  
            case  
                current_goal unifies with a fact :  
                return unifying substitutions;  
                current_goal is negated (  $\neg p$  ) :  
                begin  
                    call pattern_search on p;  
                    if pattern_search returns FAIL  
                    then return { };  
                    else return FAIL ;    %negation is true  
                end;  
            end  
        end  
    end  
end;
```



current_goal is a conjunction ($p \wedge \dots$) :

begin

for each conjunction do

begin

call **pattern_search on conjunct;**

if pattern_search returns FAIL

then return FAIL

else apply substitutions to other conjuncts;

end;

if pattern_search returns **SUCCESS for all conjuncts**

then return composition of unifications;

else return FAIL;

end;



current_goal is a disjunction ($p \vee \dots$) :

begin

repeat for each disjunct

call **pattern_search on disjunct**

until no more disjuncts or **SUCCESS;**

if pattern_search returns **SUCCESS**

then return substitutions

else return **FAIL;**

end;



current_goal unifies with the rule conclusion (p in $q \rightarrow p$):

begin

apply goal unifying substitutions to premise (q) ;

call **pattern_search on premise;**

if pattern_search returns **SUCCESS**

then return composition of p and q substitutions

else return FAIL

end;

end;

end

return FAIL

end



反向推理过程

- 反向推理过程描述如下：（递归算法）

procedure achieve (G)

将规则库中的规则后件同当前数据库内容进行匹配，若匹配成功，则找出一条可用规则送入可用规则集S；否则，用下一条规则进行匹配。



if S 为空

then 向用户直接询问假设 G 的真假性，若用户确认 G 为真或假，则问题已求解；否则，把用户回答的有关 G 的证据添加到数据库中。

else

while G 未求解 do

begin

调用 $\text{choose-rule}(S)$ ，从 S 中选择一条规则 R ：

$G' \leftarrow R$ 的前件；

if G' 未求解

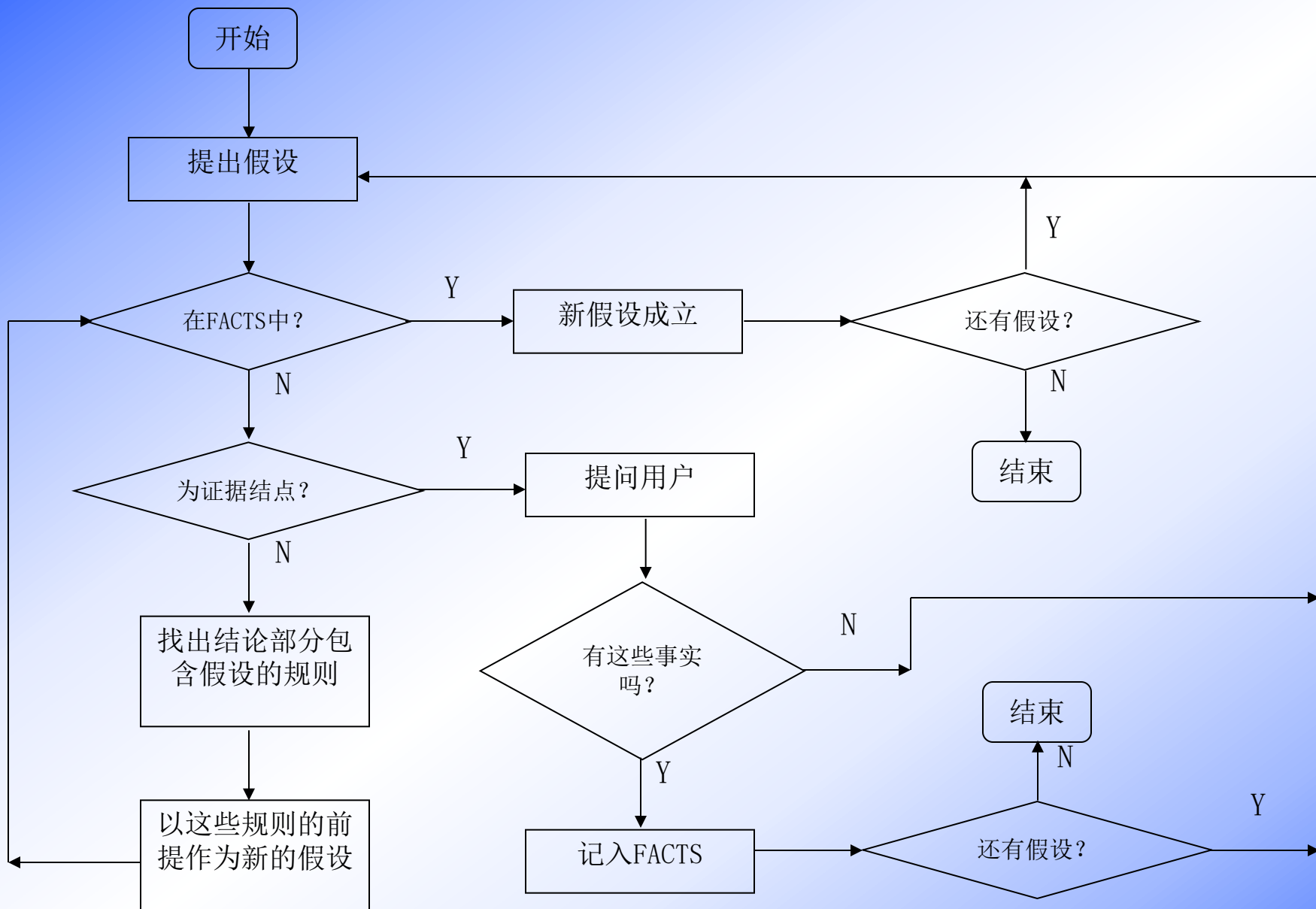
then 调用 $\text{achieve}(G')$

if G' 为真

then 把规则 R 的后件添加到数据库中

end





反向推理示意图



6.2 Production System

6.2.1 Definition and History

- **DEFINITION (production system)**

a production system is defined by :

1. The set of production rules.

- Simply called **productions**.
- A production is a **condition-action** pair defines **a single chunk** (组块) of knowledge.
- **The condition part** is **a pattern** that **determines** when that rule may be applied .
- **The action part** defines the associated problem-solving steps



2. Working memory (工作存储器) contains a description of **the current state**.

- This description is **a pattern** (模式) that is matched against (与...匹配) **the condition part** of a production to **select** appropriate production rule.
- When **the conditions of a rule** is matched by **the contents of working memory**, the **actions** of the rule **may then be performed**.
- The **actions** of rules **alter** (改变) the **contents** of working memory.



3. The **recognize – act** cycle (“识别 – 行动”)

- **Working memory** is initialized with the **beginning description**.
- The **current state** is maintained as **a set of patterns** in working memory.
- These patterns are **matched against** the conditions of the production rules;
- This produces **a subset of the production rules**, called **the conflict set (冲突集)**, whose **conditions** match **the patterns** in working memory.



- The productions in **the conflict set** are said **to be enabled** (可行的 , 可用的) .
- **One of them** is then selected out, this is called **conflict resolution** (冲突消解)
- The selected production is **fired** (激活 , 使用) , that is, its action is **performed**, **changing** the contents of working memory.
- The control cycle **repeats** with **the modified working memory**.
- The process **terminates** when the contents of working memory **do not match** any rule's conditions.



- **Conflict resolution** (冲突消解) chooses a rule from the conflict set for firing.
- Conflict resolution **strategies** may be simple, such as **selecting the first rule whose condition matches the contents** of working memory.
- Conflict resolution strategies may involve complex **rule selection heuristics**.
- This is an important way to **add heuristic control**.

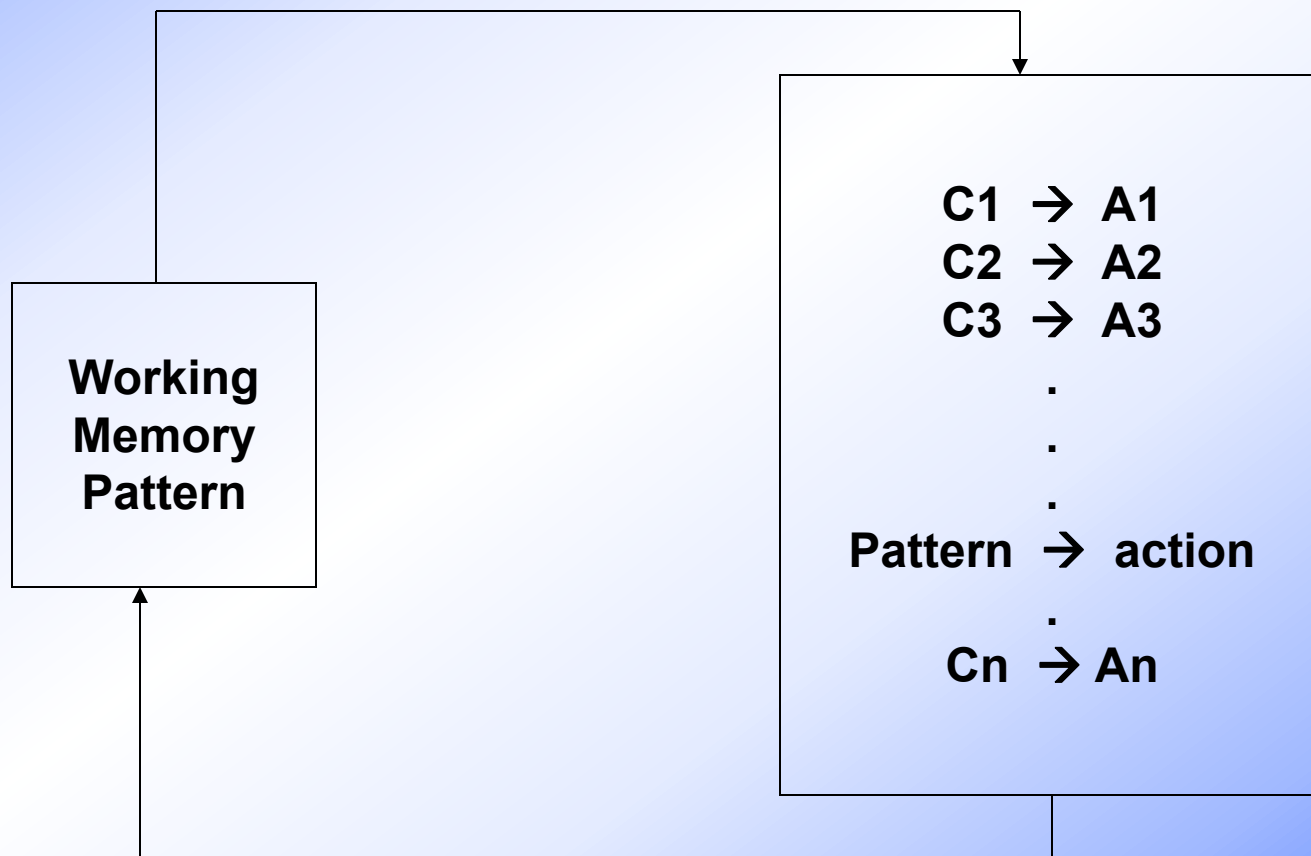


- The **pure production system model** has no mechanism for **recovering from** dead ends in the search;
- It simply continues until **no more productions are enabled** and halts.
- Most **practical implementations** of production systems allow **backtracking** to a previous (前一个) state of working memory .



Figure 6.1 A production system.

Control loops until **working memory pattern**
no longer matches **the conditions** of any productions



Production set :

1. $ba \rightarrow ab$
2. $ca \rightarrow ac$
3. $cb \rightarrow bc$

Iteration #	Working memory	Conflict set	Rule fired
0	cbaca	1,2,3	1
1	cabca	2	2
2	acbca	2,3	2
3	acbac	1,3	1
4	acabc	2	2
5	aacbc	3	3
6	aabcc	\emptyset	Halt

Figure 6.2 Trace of a simple production system.



6.2.2 Examples of Production System

EXAMPLE 6.2.1

the 8-puzzle, revisited

- First introduced in Chapter 3
- Used in Chapter 4 to explore different search strategies
- the 8-puzzle is both **complex enough** to be interesting and **small enough** to be tractable
(易于处理)



Start state:

2	8	3
1	6	4
7		5

Goal state:

1	2	3
8		4
7	6	5

● **Production set :**

condition

action

- 1. Goal state in working memory** → **halt**
- 2. Blank is not on the left edge** → **move the blank left**
- 3. Blank is not on the top edge** → **move the blank up**
- 4. Blank is not on the right edge** → **move the blank right**
- 5. Blank is not on the bottom edge** → **move the blank down**



Start state:

2	8	3
1	6	4
7		5

Goal state:

1	2	3
8		4
7	6	5

- **Working memory** : is the **present** board state and **goal** state.
- **Control regime** (机制) :
 1. Try each **production rule** in order.
 2. Do not allow **loops**.
 3. **Stop** when goal is found.



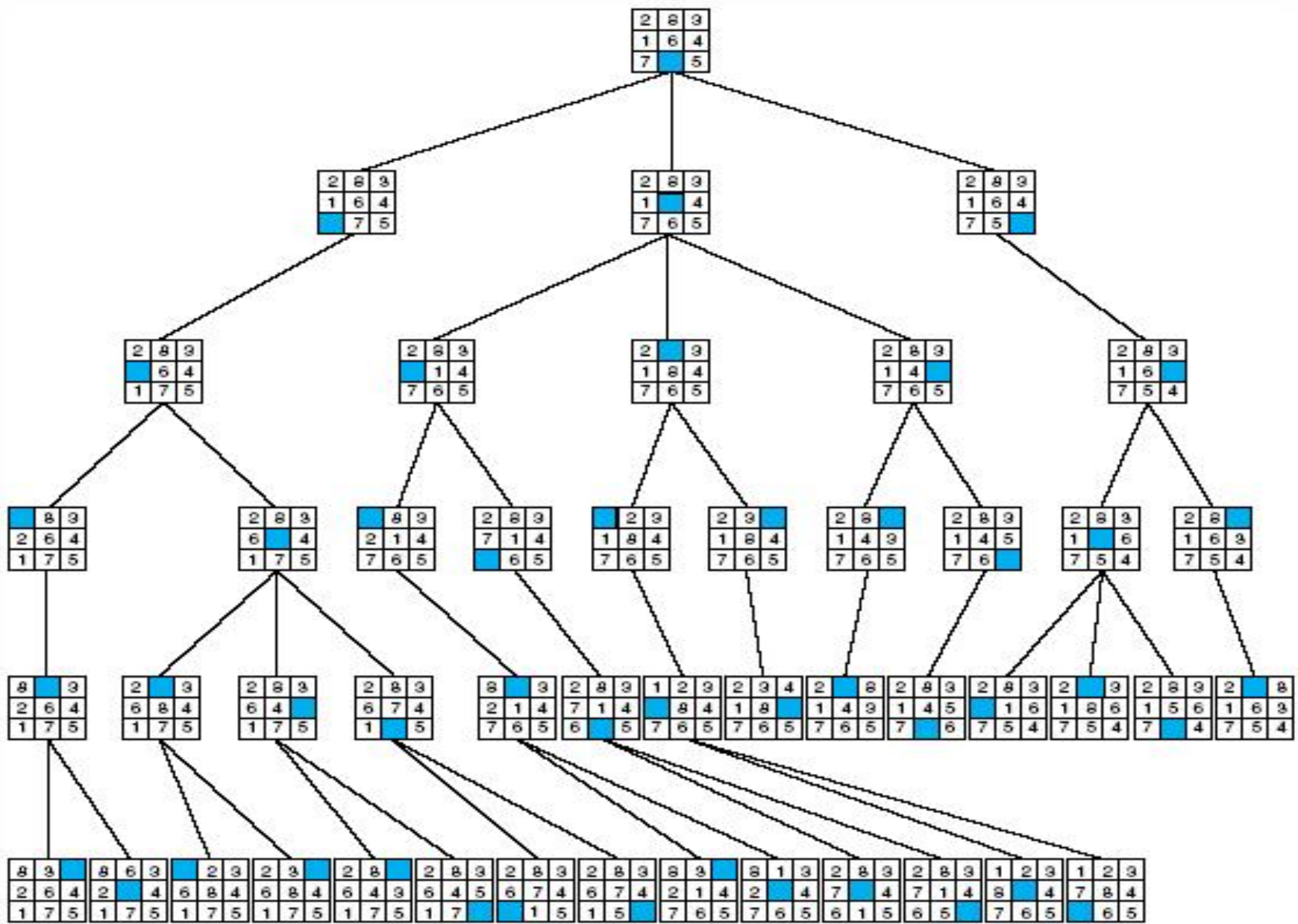


Figure 6.4 the 8-puzzle searched by a production system with loop detection and depth bound 5



EXAMPLE 6.2.2: the knight's tour (骑士周游) problem:

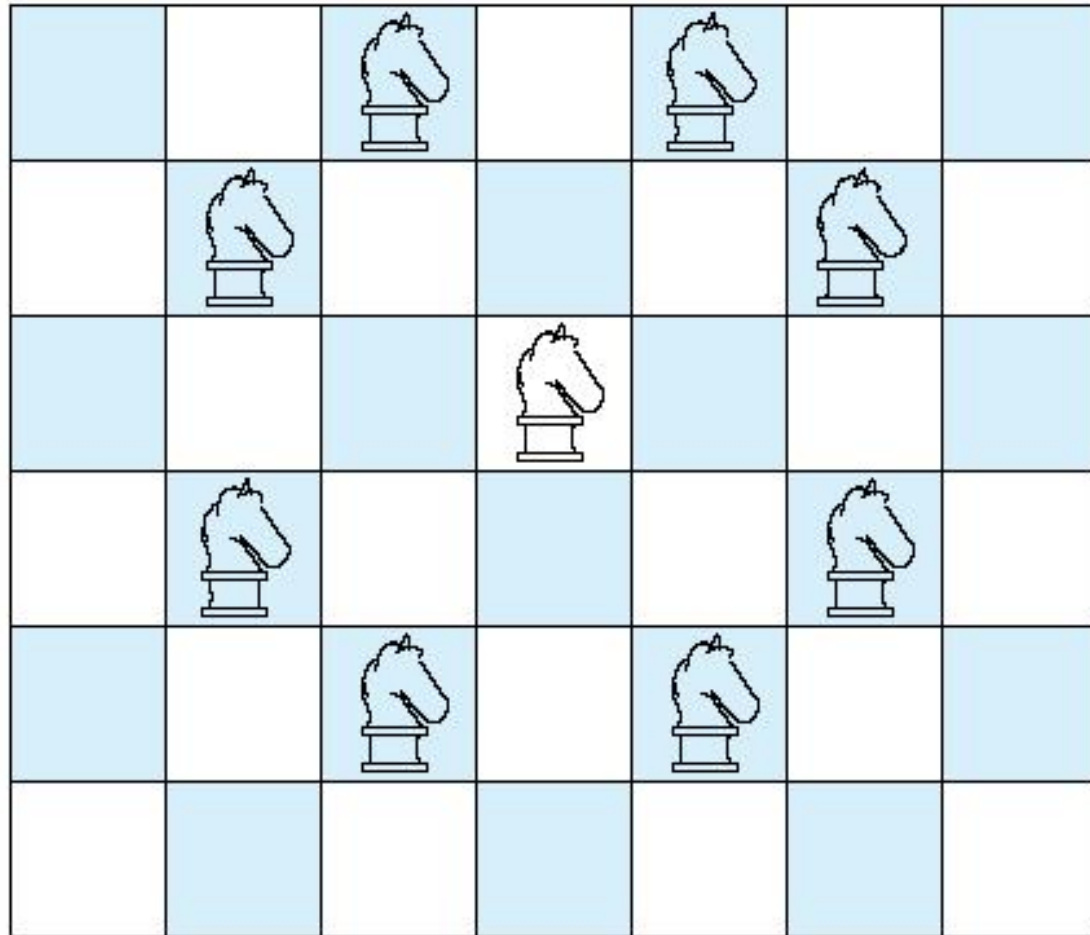
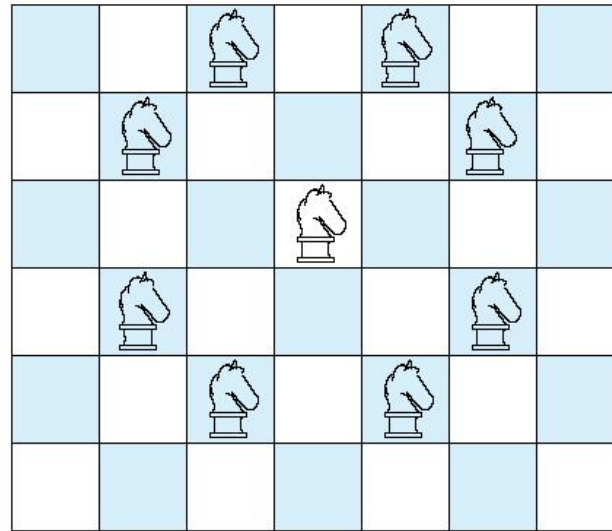


Figure 6.5 **Legal moves** of a chess knight
合法移动——最多可以有8种可能的移动



The knight's tour (骑士周游) problem:



- **traditional definition** : lands on each square exactly once
- **simplified version** : Find a path from one square to another



move(1, 8) move(6, 1)

move(1, 6) move(6, 7)

move(2, 9) move(7, 2)

move(2, 7) move(7, 6)

move(3, 4) move(8, 3)

move(3, 8) move(8, 1)

move(4, 9) move(9, 2)

move(4, 3) move(9, 4)

1	2	3
4	5	6
7	8	9

A 3×3 chessboard with move rules
for the simplified knight tour problem.



Rule#	Condition	Action
1.	knight on square 1	→ move knight to square 8
2.	knight on square 1	→ move knight to square 6
3.	knight on square 2	→ move knight to square 9
4.	knight on square 2	→ move knight to square 7
5.	knight on square 3	→ move knight to square 4
6.	knight on square 3	→ move knight to square 8
7.	knight on square 4	→ move knight to square 9
8.	knight on square 4	→ move knight to square 3
9.	knight on square 6	→ move knight to square 1
10.	knight on square 6	→ move knight to square 7
11.	knight on square 7	→ move knight to square 2
12.	knight on square 7	→ move knight to square 6
13.	knight on square 8	→ move knight to square 3
14.	knight on square 8	→ move knight to square 1
15.	knight on square 9	→ move knight to square 2
16.	knight on square 9	→ move knight to square 4

Production rules for the 3×3 knight problem



A production system solution to the 3×3 knights tour problem

Find a path from square 1 to square 2

Iteration #	Working memory		Conflict set (rule #'s)	Fire rule
	Current square	Goal square		
0	1	2	1, 2	1
1	8	2	13, 14	13
2	3	2	5, 6	5
3	4	2	7, 8	7
4	9	2	15, 16	15
5	2	2		halt



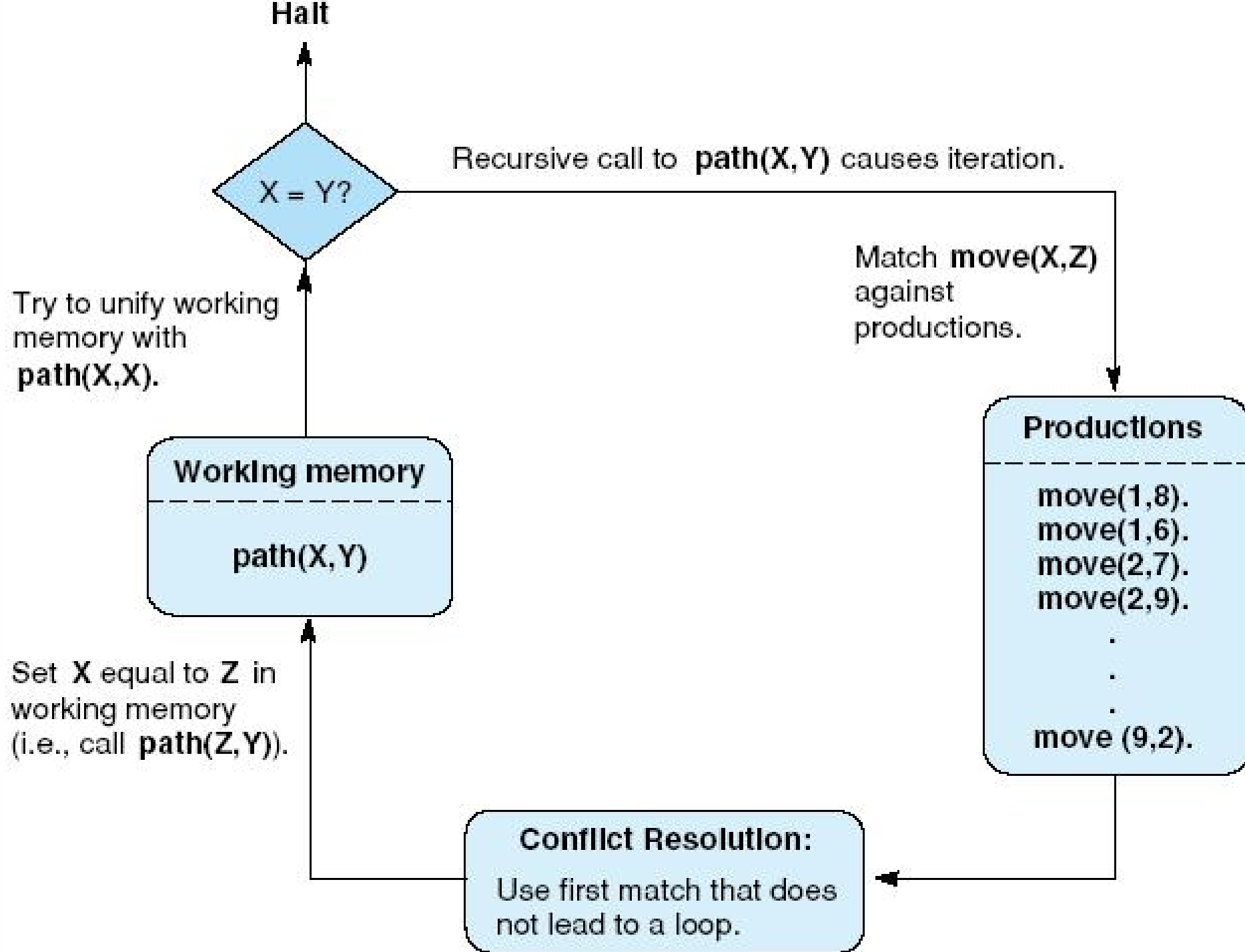
- We may also implement a **recursive process**.
(Recursive Backtrack, Depth-first in PROLOG)
- A **path** can be defined **recursively** by two **predicate calculus formulas**:

$\forall X \text{ path } (X, X)$

$\forall X, Y [\exists Z [\text{path}(X, Y) \leftarrow \text{move}(X, Z) \wedge \text{path}(Z, Y)]]$

- The **recursive process** may be **described** with the following figure.





- The modified **recursive path controller** for the **production system** is :

$\forall X \text{path}(X, X)$

$\forall X, Y [\text{path}(X, Y) \leftarrow \exists Z [\text{move} (X, Z) \wedge$
 $\neg (\text{been} (Z)) \wedge$
 $\text{assert} (\text{been} (Z)) \wedge$
 $\text{path} (Z, Y)]]$

Assert(x):把X移进工作内存

Been(x):代表棋盘一方格



EXAMPLE 6.2.3

the full knight's tour : 8×8 chessboard

- Condition:

$\text{current_row} \leq 6 \wedge$

$\text{current_column} \leq 7$

- Action:

$\text{new_row} = \text{current_row} + 2 \wedge$

$\text{new_column} = \text{current_column} + 1$

- ——向下移动两格并向右移动一格的定义。类似地可以写出如图6-5所示共8种可能的移动。



EXAMPLE 6.2.3: the full knight's tour

- Rewritten rule

move(square(row, column),

square(newrow, nwecolumn)) ←

less_than_or_equals (row, 6) ∧

equals (newrow, plus (row, 2)) ∧

less_than_or_equals (column, 7) ∧

equals (newcolumn, plus (column, 1)



EXAMPLE 6.2.4:

- the financial advisor (理财顾问) as a production system



- Production Rules :

1. $\text{saving_account}(\text{inadequate}) \rightarrow \text{investment}(\text{savings})$
2. $\text{saving_account}(\text{adequate}) \wedge \text{income}(\text{adequate})$
 $\rightarrow \text{investment}(\text{stocks})$
3. $\text{saving_account}(\text{adequate}) \wedge \text{income}(\text{inadequate})$
 $\rightarrow \text{investment}(\text{combination})$
4. $\forall X \text{ amount_saved}(X) \wedge \exists Y(\text{dependents}(Y)$
 $\wedge \text{greater}(X, \text{minsavings}(Y)))$
 $\rightarrow \text{saving_account}(\text{adequate})$



5. $\forall X \text{amount_saved}(X) \wedge \exists Y(\text{dependents}(Y) \wedge \neg \text{greater}(X, \text{minsavings}(Y))) \rightarrow \text{saving_account}(\text{inadequate})$
6. $\forall X \text{earnings}(X, \text{steady}) \wedge \exists Y(\text{dependents}(Y) \wedge \text{greater}(X, \text{minincome}(Y))) \rightarrow \text{income}(\text{adequate})$
7. $\forall X \text{earnings}(X, \text{steady}) \wedge \exists Y(\text{dependents}(Y) \wedge \neg \text{greater}(X, \text{minincome}(Y))) \rightarrow \text{income}(\text{inadequate})$
8. $\forall X \text{earnings}(X, \text{unsteady}) \rightarrow \text{income}(\text{inadequate})$



- **Working Memory**

- 1. amount_saved(22000)**
- 2. earnings(25000, steady)**
- 3. dependents(3)**



6.2.3 Control of Search in Production System

- The production system model offers **opportunities for using heuristics** :
 1. The Choice of **Data-Driven** or **Goal-Driven** Search Strategy, according to the **branching factor**
 2. Rule Structure
 3. Conflict Resolution



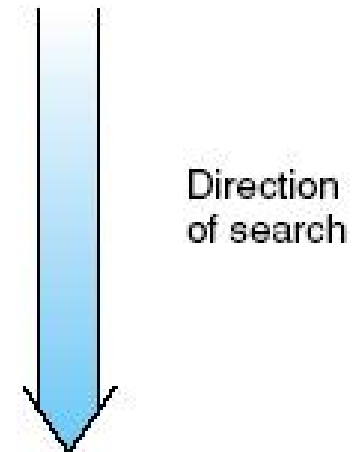
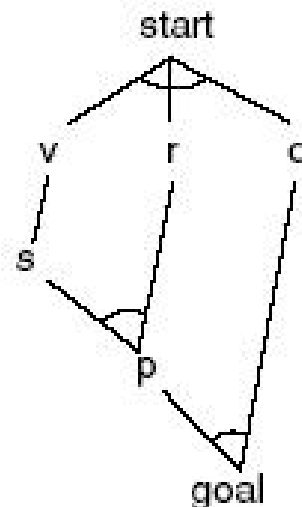
Production set:

1. $p \wedge q \rightarrow \text{goal}$
2. $r \wedge s \rightarrow p$
3. $w \wedge r \rightarrow q$
4. $t \wedge u \rightarrow q$
5. $v \rightarrow s$
6. $\text{start} \rightarrow v \wedge r \wedge q$

Trace of execution:

Iteration #	Working memory	Conflict set	Rule fired
0	start	6	6
1	start, v, r, q	6, 5	5
2	start, v, r, q, s	6, 5, 2	2
3	start, v, r, q, s, p	6, 5, 2, 1	1
4	start, v, r, q, s, p, goal	6, 5, 2, 1	halt

Space searched by execution:



data-driven search in a production system.

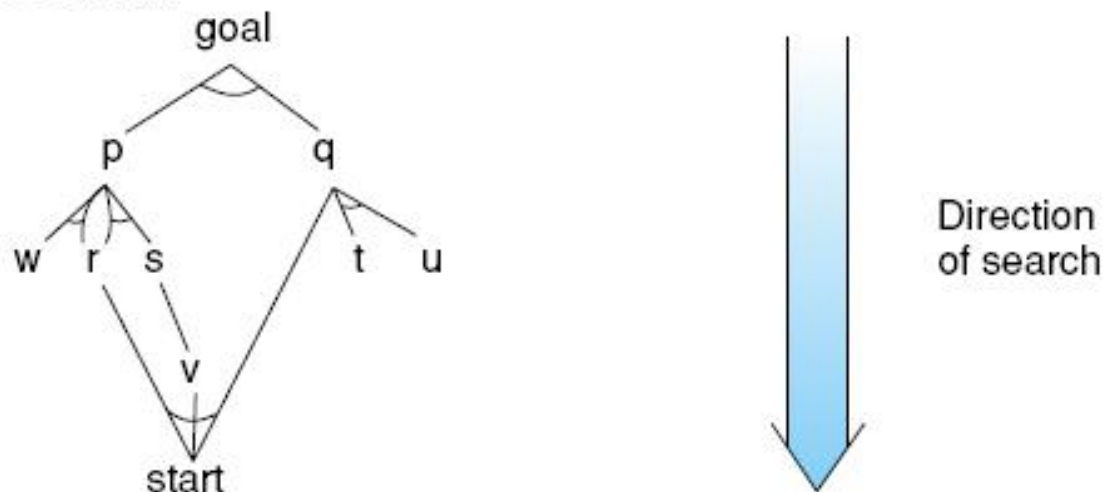


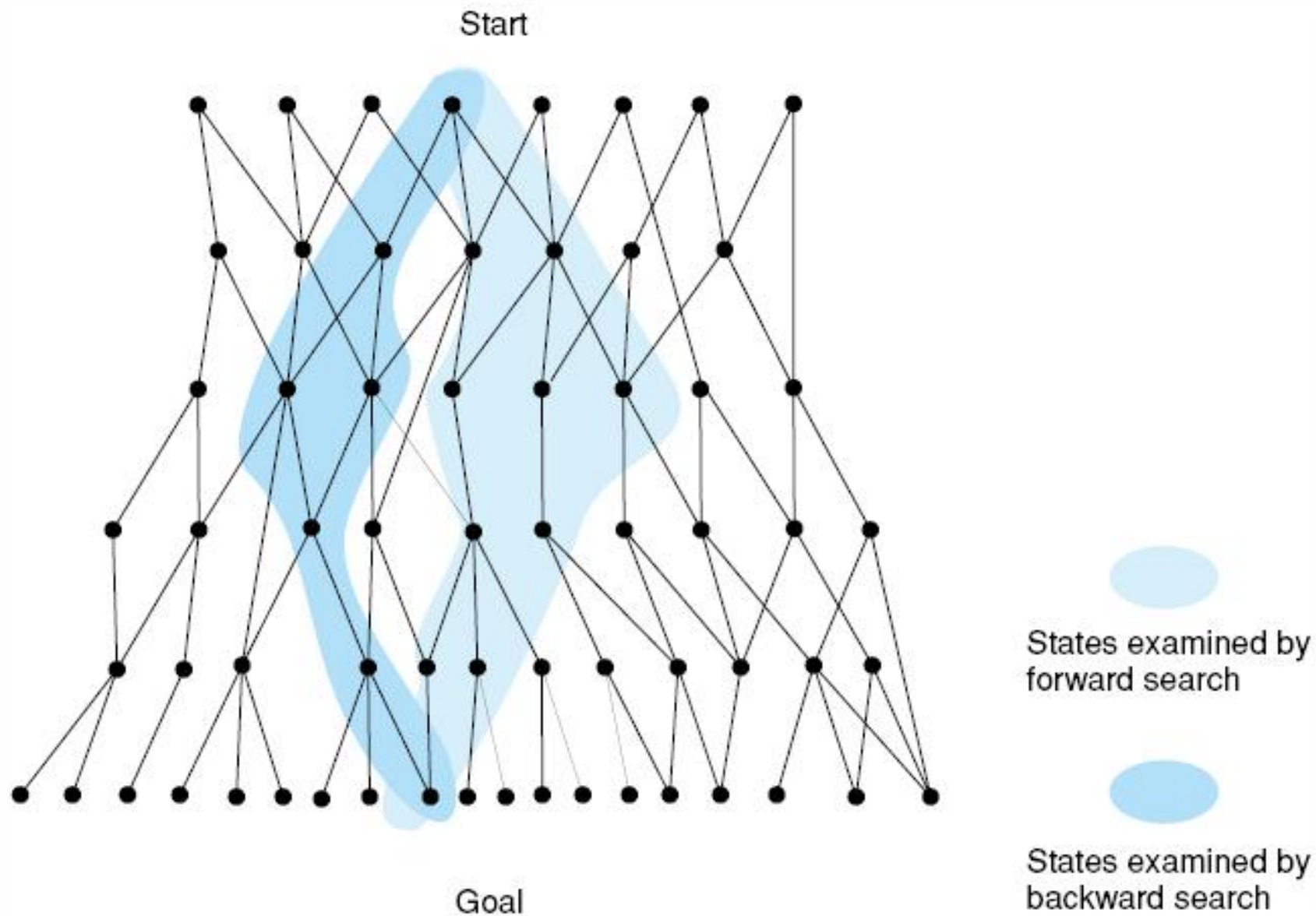
Production set:

1. $p \wedge q \rightarrow \text{goal}$
2. $r \wedge s \rightarrow p$
3. $w \wedge r \rightarrow p$
4. $t \wedge u \rightarrow q$
5. $v \rightarrow s$
6. $\text{start} \rightarrow v \wedge r \wedge q$

Trace of execution:

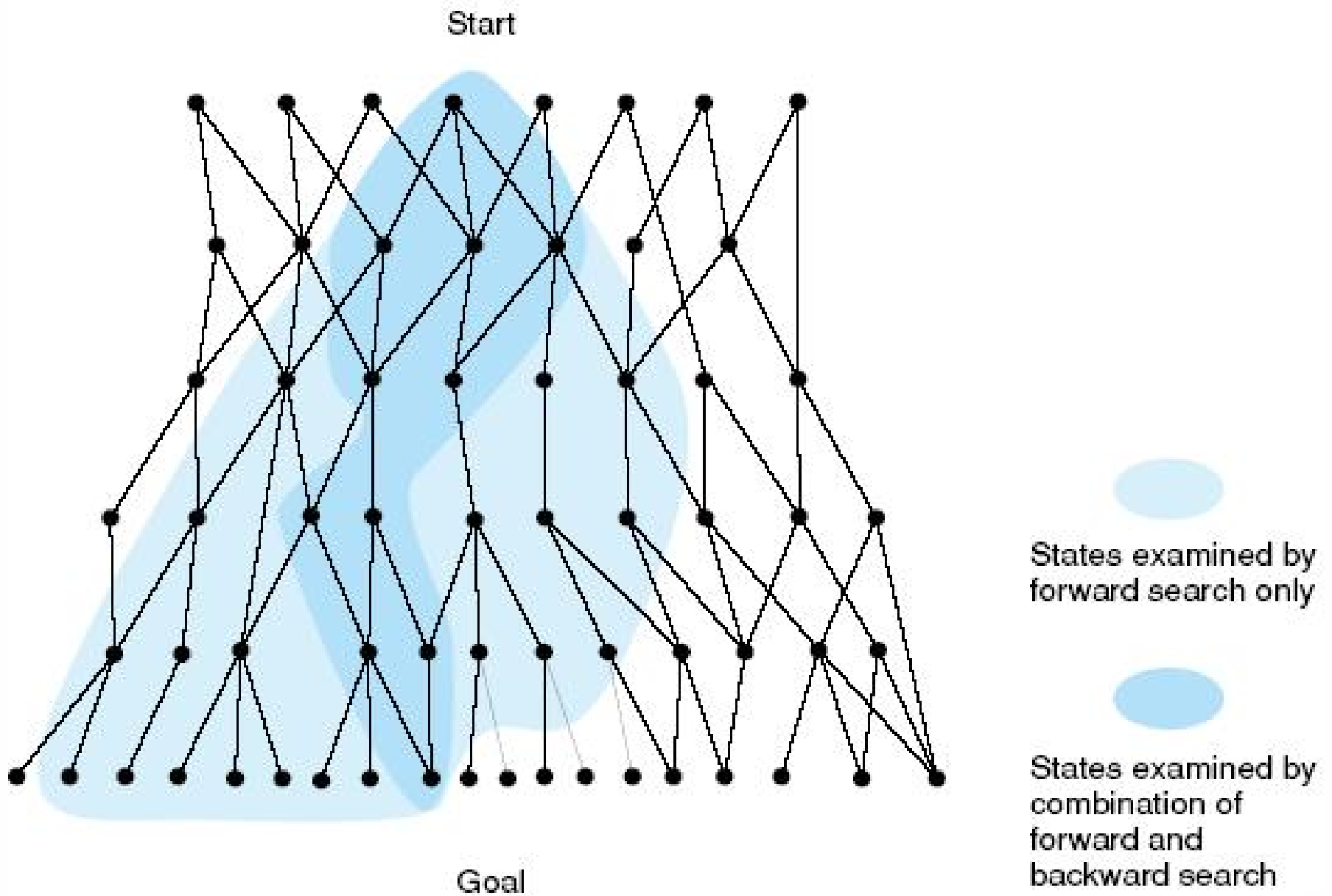
Iteration #	Working memory	Conflict set	Rule fired
0	goal	1	1
1	goal, p, q	1, 2, 3, 4	2
2	goal, p, q, r, s	1, 2, 3, 4, 5	3
3	goal, p, q, r, s, w	1, 2, 3, 4, 5	4
4	goal, p, q, r, s, w, t, u	1, 2, 3, 4, 5	5
5	goal, p, q, r, s, w, t, u, v	1, 2, 3, 4, 5, 6	6
6	goal, p, q, r, s, w, t, u, v, start	1, 2, 3, 4, 5, 6	halt

Space searched by execution:



Bi-directional search missing in both directions,
resulting in excessive (额外的) search





Bi-directional search meeting in the middle,
Eliminating much of the space examined by **uni-directional** (单向的) search



Control of Search through Rule Structure

- The structure of rules in a production system, including **the distinction between the condition and the action**, determines the fashion in which the space is searched.
- EXAMPLE two **logically equivalent** rules :

$$\forall X (\text{foo}(X) \wedge \text{goo}(X) \rightarrow \text{moo}(X))$$

$$\forall X (\text{foo}(X) \rightarrow \text{moo}(X) \vee \neg \text{goo}(X))$$

- They don't have **the same behavior** in the search implementation.
- **The order** in which **conditions are tried** also determines the search fashion.



例：知识表示的挑战

信息：某人X有一笔贷款Y。在person(人)和loan(贷款)之间有has-loan(借贷)关系。

知识：所有申请贷款者至少18周岁。这些知识片段告诉我们关于人和贷款的一般信息，而不仅仅是关于特定的人—贷款实例。



信息

John 有 \$1750 贷款

Harry 有 \$2500 贷款

知识

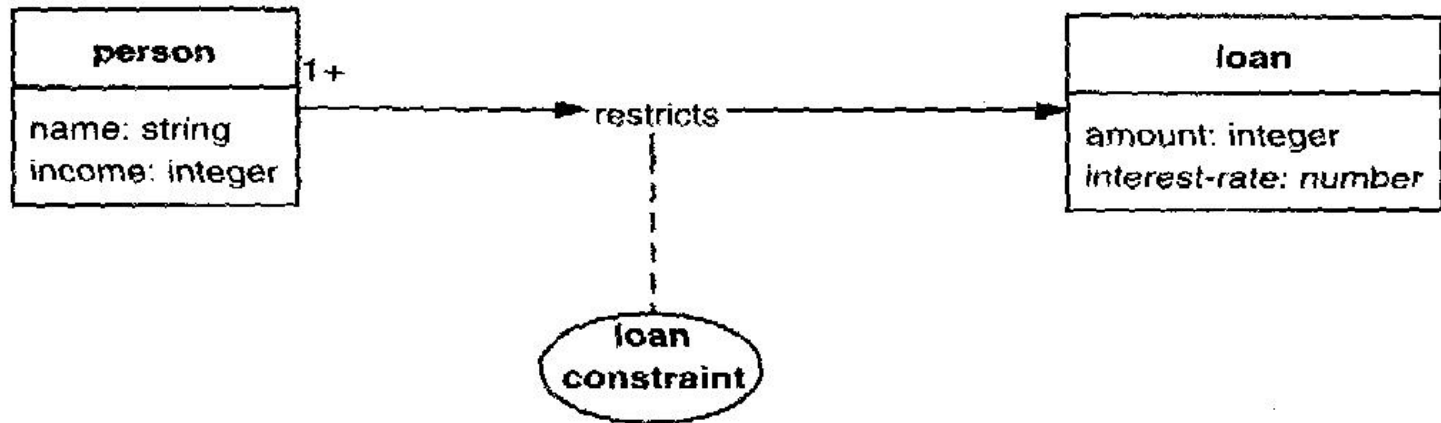
贷款人年龄至少 满18 岁

收入达到 \$10 000 的人可贷款的最高限额为 \$2000

收入在 \$10 000 到 \$20 000 之间的人贷款最高限额为 \$3000



用这种方式，领域构建一些能够用于区别知识类型的规则类型。如图1。模式下边以规则类型“实例”的方式列出了真正的规则。



```
person.income <= 10 000
```

```
RESTRICTS
```

```
loan.amount <= 2 000
```

```
person.income > 10 000 AND person.income <= 20 000
```

```
RESTRICTS
```

```
loan.amount <= 3 000
```

图1.贷款—评估知识规则类型



● Control of search through conflict resolution

Conflict resolution **strategies** supported :

1. Refraction (折射) . Once a rule has fired, it may not fire again until **the working memory elements** that **match its conditions** have been **modified**.
2. Recency (最新性) . **Prefers rules** whose conditions match with **the patterns most recently added to working memory**. This **focuses the search** on a single line of reasoning (一条推理线) .
3. Specificity (特殊性) . Assumes that it is appropriate to use a **more specific** rule rather than a **more general** one. One rule is more specific than another if it **has more conditions**.

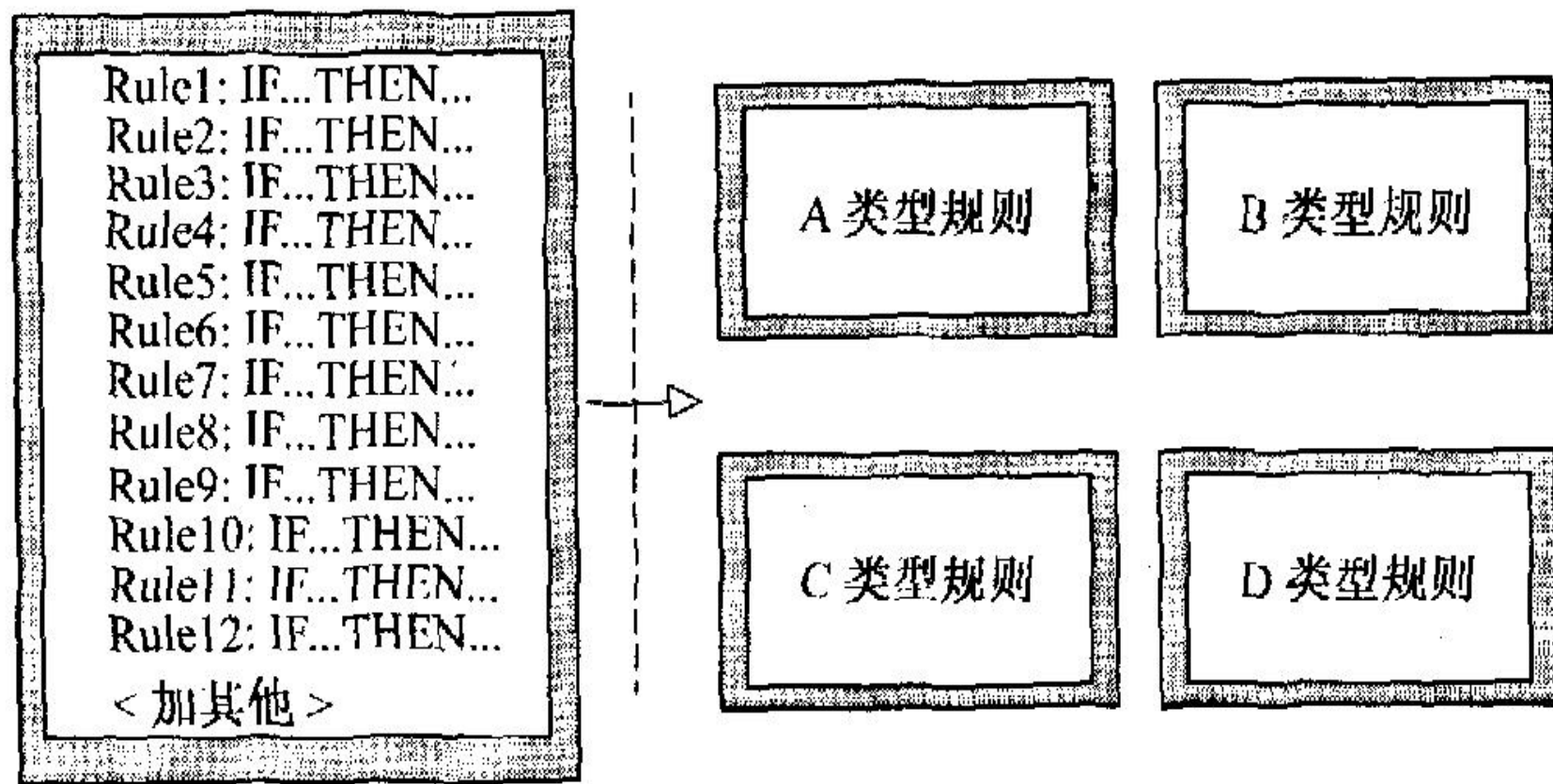


6.2.4 Advantages of Production System for AI

- Separation of **knowledge** and **control**
- A natural **mapping** onto state space search
- **Modularity** (模块性) of production rules
- Pattern-directed (模式导向) control
- Opportunities for **heuristic control** of search
- Tracing and explanation (跟踪与解释)
- A plausible (合理) model of **human problem solving**



不是一个包含所有规则的大的平面知识库，而是一个将知识库细分成具有相似分区的小部分(如规则集)，目的是识别知识片段具有相似结构的部分。



单独平面知识库

包含相似结构规则的多规则库

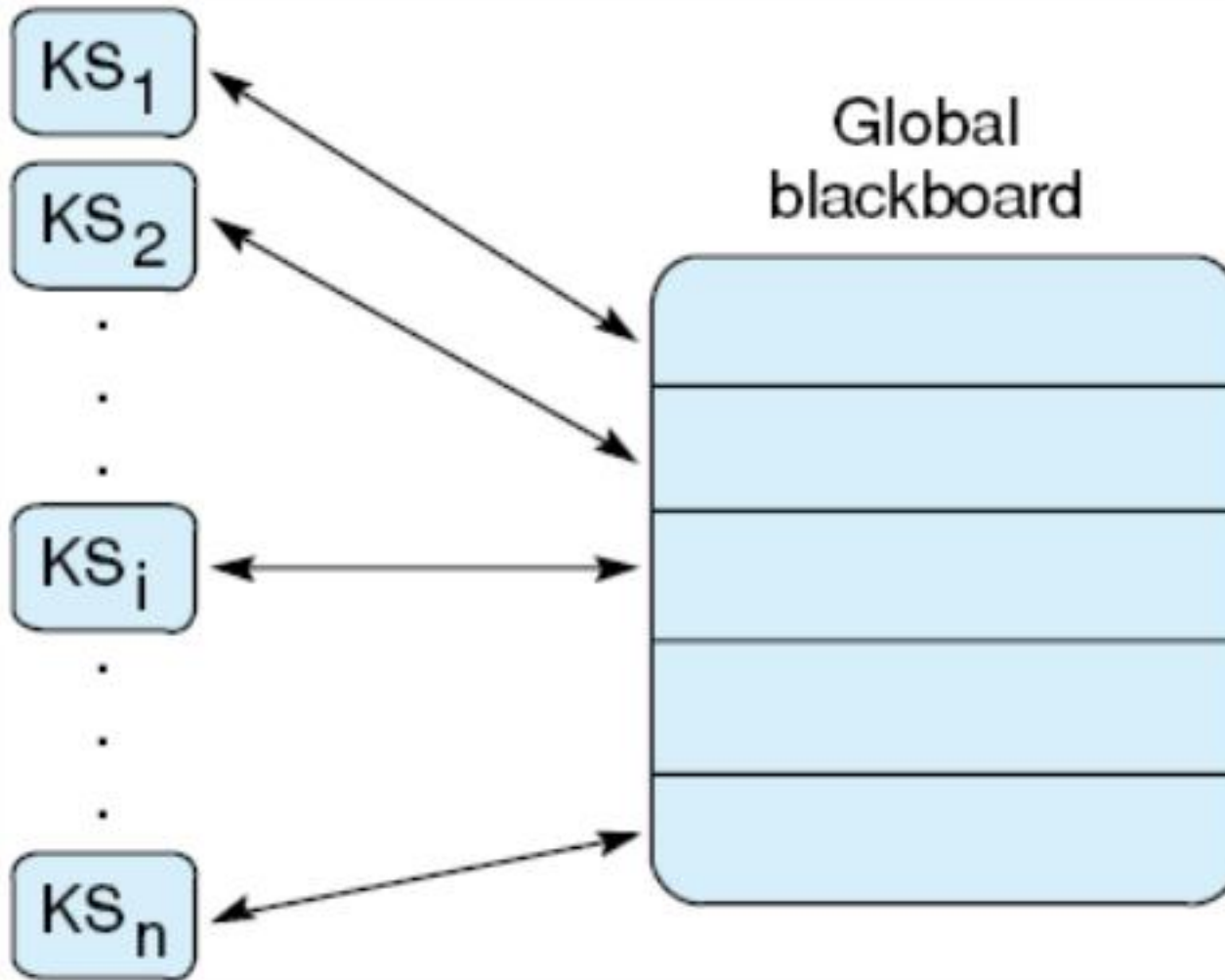


6.3 The Blackboard Architecture for Problem Solving

- Blackboards (黑板结构) extend production systems by allowing us to organize **Working memory** into **separate modules**, each of which corresponds to **a different subset** of the production rules.
- Blackboards **integrate** these separate sets of production rules and **coordinate** (协调) the actions of these multiple problem solving **agents** (主体), sometimes called **knowledge sources** (知识源), within a **single global structure**, the blackboard.



Figure 6.13 Blackboard architecture



- A blackboard is a **central global** data base for the **communication** of knowledge sources (知识源) .
- Knowledge sources are **independent** and **asynchronous** (异步的) .
- Knowledge sources focus on **related aspects** of a particular problem.



产生式系统 (production system)

小结



产生式

- 一种知识表示方法，常用来表示有因果关系的知识。
- 形式：
 - 条件 \rightarrow 行动
 - 前提 \rightarrow 结论
 - “**if.....then.....**”
- 例如：
 - 烫手 \rightarrow 缩手
 - 下雨 \rightarrow 地面湿
 - 下雨 \wedge 甲未打伞 \rightarrow 甲被淋湿
 - 所有人会死 \wedge 甲是人 \rightarrow 甲会死



- \rightarrow 左边表示条件(左半部分), 右边表示结论(右半部分)
- 一般可以写成 $A_1 \wedge A_2 \wedge \dots \wedge A_n \rightarrow B$ 的形式;
 - 下雨 \wedge 甲未打伞 \rightarrow 甲被淋湿



产生式系统

➤ 把一组产生式放在一起，让它们互相配合，协同作用，一个产生式的结论可以供另一个产生式作为前提使用，以这种方式求解问题的系统称为产生式系统。

- $A \rightarrow B, B \rightarrow C, C \rightarrow D, A$
- 推出D？



- 产生式系统的构成

- 一组产生式规则(set of rules)

- 综合数据库(global database)

- 控制机制(control system)



综合数据库

- 存放已知的事实和推导出的事实;
- 和database（数据库）不同：
 - database: 强调数据的管理（存取、增、删、改等）
 - 产生式系统: 抽象的概念
 - ✓ 只是说明数据在此存放，和物理实现没关系。
 - ✓ 具体实现时，用DBMS和文件等都可以。
 - ✓ 数据是广义的，可以是常量、变量、谓词、图像等。
- 数据结构：
 - 符号串、向量、集合、数组、树、表格、文件等;



控制机制

- 控制机制完成的工作有：
 - 匹配规则条件部分；
 - 多于一条规则匹配成功时，选择哪条规则执行(点燃)；
 - 如何将匹配规则的结论部分放入综合数据库（是直接添加到数据库中，还是替换其中的某些东西）；
 - 决定系统何时终止；



产生式系统的运行过程

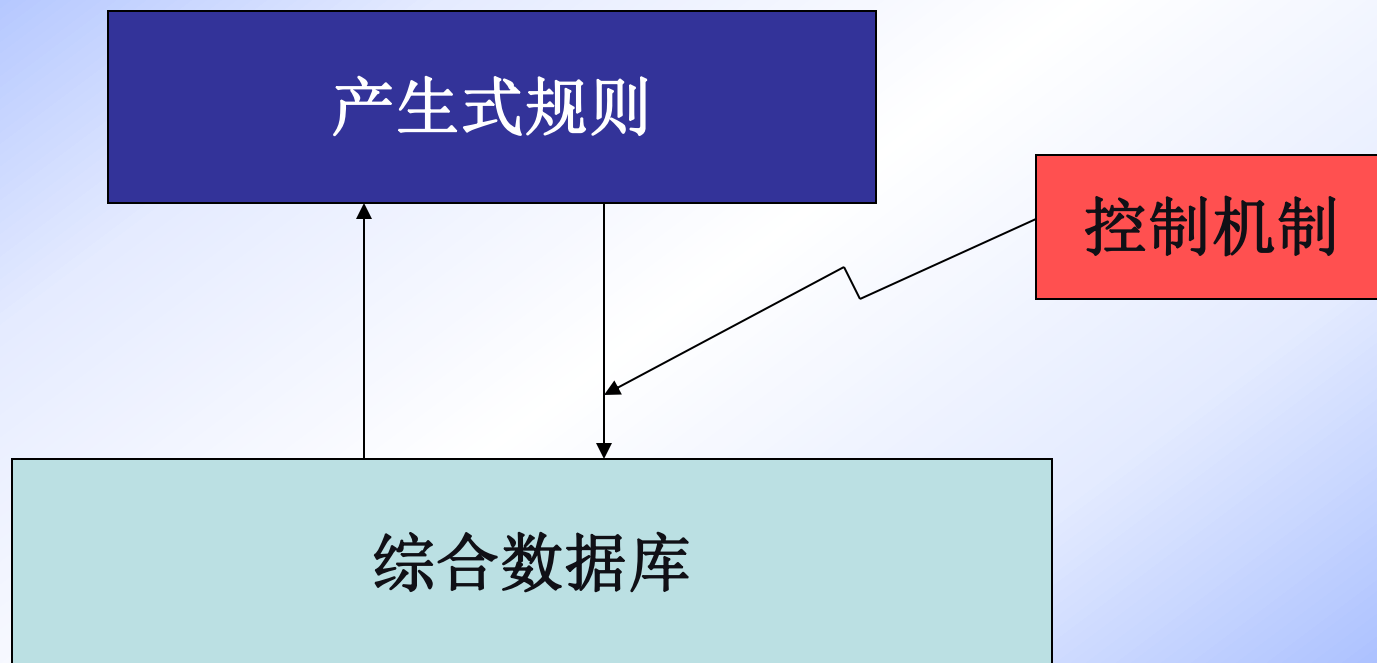
- 建立产生式规则；
- 将已知的事实放入综合数据库；
- 考察每一条产生式规则，如果条件部分和综合数据库中的数据匹配，则规则的结论放入综合数据库；




算法

- 令**DATA**为综合数据库
 1. 初始化**DATA** ;
 2. 如果满足终止条件, 终止。否则:
 - a) 选择一个可应用于**DATA**之上的规则**R**;
 - b) **R**应用于**DATA**之上产生的结果→**DATA**。





传教士与野人问题



应用举例

- 传教士与野人问题描述：

有**3**名传教士和**3**名野人来到一条河的左岸，欲乘一条船渡河到右岸，该船的最大负载能力为**2**人，传教士与野人均可撑船。在任何时候，不论是在右岸还是左岸，如果野人人数超过传教士人数，那么，野人就会吃掉传教士。为了规划出一个渡河方案，把**6**个人都安全地渡过河去，请用产生式表示法表示求解该问题的所有规则。



问题解答：(1) 综合数据库

解:用三元组表示左岸的状态, 即 (M_L, C_L, B_L) , 其中

$$0 \leq M_L, C_L \leq 3, B_L \in \{0, 1\}$$

其中M、C分别代表某一岸上传教士与野人的数目, $B=1$ 表示船在这一岸, $B=0$ 则表示船不在。

此时问题描述简化为: $(3, 3, 1) \rightarrow (0, 0, 0)$ 。

对于 **$N=3$** 的 **$M-C$** 问题, 状态空间的总数为 $4 \times 4 \times 2 = 32$, 根据约束条件的要求, 可以看出只有20个合法状态。再进一步分析后, 又发现有4个合法状态实际上是不可能达到的。因此实际的问题空间仅有**16**种状态符合要求, 其它的状态不符合要求。

下表列出分析的结果:



(M_L, C_L, B_L)	(M_L, C_L, B_L)
$(0\ 0\ 1)$ 达不到	$(0\ 0\ 0)$
$(0\ 1\ 1)$	$(0\ 1\ 0)$
$(0\ 2\ 1)$	$(0\ 2\ 0)$
$(0\ 3\ 1)$	$(0\ 3\ 0)$ 达不到
$(1\ 0\ 1)$ 不合法	$(1\ 0\ 0)$ 不合法
$(1\ 1\ 1)$	$(1\ 1\ 0)$
$(1\ 2\ 1)$ 不合法	$(1\ 2\ 0)$ 不合法
$(1\ 3\ 1)$ 不合法	$(1\ 3\ 0)$ 不合法
$(2\ 0\ 1)$ 不合法	$(2\ 0\ 0)$ 不合法
$(2\ 1\ 1)$ 不合法	$(2\ 1\ 0)$ 不合法
$(2\ 2\ 1)$	$(2\ 2\ 0)$
$(2\ 3\ 1)$ 不合法	$(2\ 3\ 0)$ 不合法
$(3\ 0\ 1)$ 达不到	$(3\ 0\ 0)$
$(3\ 1\ 1)$	$(3\ 1\ 0)$
$(3\ 2\ 1)$	$(3\ 2\ 0)$
$(3\ 3\ 1)$	$(3\ 3\ 0)$ 达不到



(2) 规则集合:

由摆渡操作组成。

该问题主要有两种操作:

p_{mc} 操作(规定为从左岸划向右岸),

q_{mc} 操作(从右岸划向左岸)。

每次摆渡操作, 船上人数有五种组合, 因而组成有**10**条规则的集合。下面定义的规则前**5**条为 p_{mc} 操作(从左岸划向右岸), 后**5**条为 q_{mc} 操作(从右岸划向左岸)。



规则集合

- R1: if $(M_L, C_L, B_L=1)$ then (M_L-1, C_L, B_L-1) ; (p_{10} 操作)
- R2: if $(M_L, C_L, B_L=1)$ then (M_L, C_L-1, B_L-1) ; (p_{01} 操作)
- R3: if $(M_L, C_L, B_L=1)$ then (M_L-1, C_L-1, B_L-1) ; (p_{11} 操作)
- R4: if $(M_L, C_L, B_L=1)$ then (M_L-2, C_L, B_L-1) ; (p_{20} 操作)
- R5: if $(M_L, C_L, B_L=1)$ then (M_L, C_L-2, B_L-1) ; (p_{02} 操作)
- R6: if $(M_L, C_L, B_L=0)$ then (M_L+1, C_L, B_L+1) ; (q_{10} 操作)
- R7: if $(M_L, C_L, B_L=0)$ then (M_L, C_L+1, B_L+1) ; (q_{01} 操作)
- R8: if $(M_L, C_L, B_L=0)$ then (M_L+1, C_L+1, B_L+1) ; (q_{11} 操作)
- R9: if $(M_L, C_L, B_L=0)$ then (M_L+2, C_L, B_L+1) ; (q_{20} 操作)
- R10: if $(M_L, C_L, B_L=0)$ then (M_L, C_L+2, B_L+1) ; (q_{02} 操作)



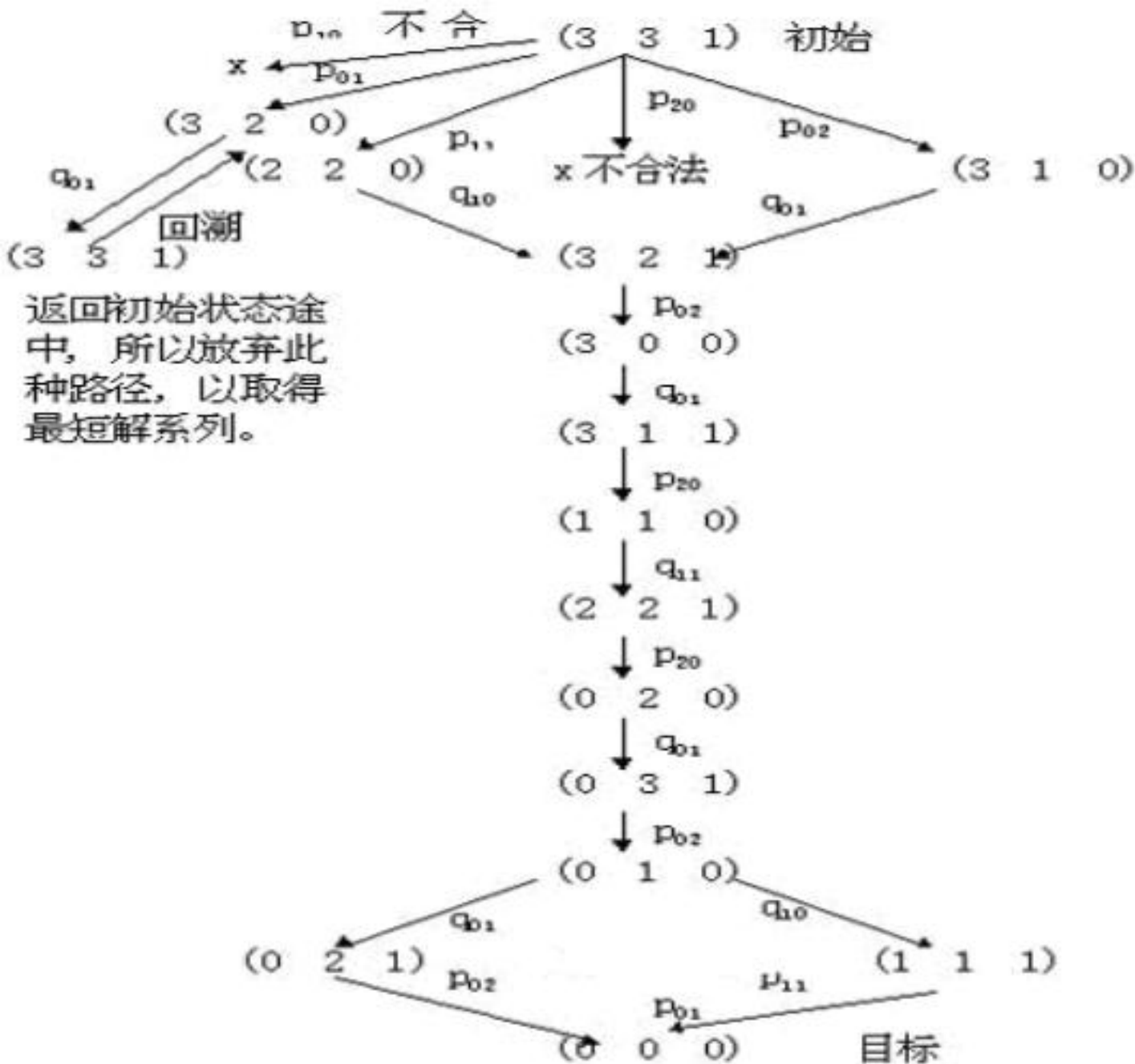
(3) 初始和目标状态: $(3, 3, 1)$ 和 $(0, 0, 0)$ 。

建立了产生式系统描述之后, 就可以通过控制策略, 对状态空间进行搜索, 求得一个摆渡操作序列, 使其实现目标状态。

状态空间图是一个有向图, 其节点可表示问题的各种状态(综合数据库), 节点之间的弧线代表一些操作(产生式规则), 它们可把一种状态导向另一种状态。这样建立起来的状态空间图, 描述了问题所有可能出现的状态及状态和操作之间的关系, 因而可以较直观地看出问题的解路径及其性质。

实际上只有问题空间规模较小的问题才可能作出状态空间图。由于每个摆渡操作都有对应的逆操作, 即 p_{mc} 对应 q_{mc} , 所以该图也可表示成具有双向弧的形式。





M-C 问题状态空间图

从状态空间图看出解序列相当之多, 但最短解序列只有4个, 例如:

(p11、q10、p02、q01、p20、q11、p20、q01、p02、q01、p02)、
(p11、q10、p02、q01、p02、q11、p20、q01、p02、q10、p11)、
(p02、q01、p02、q01、p20、q11、p20、q01、p02、q01、p02)、
(p02、q01、p02、q01、p20、q11、p20、q01、p02、q10、p11),

若给定其中两个状态分别作为初始和目标状态, 就立即可找出对应的解序列来。在一般情况下, 求解过程就是对状态空间搜索出一条解路径的过程。

以上这个例子说明了建立产生式系统描述的过程, 这也就是所谓问题的表示。对问题表示的好坏, 往往对求解过程的效率有很大影响。一种较好的表示法会简化状态空间和规则集表示。

其中的一条解路径为:

(3 3 1) → (3 1 0) → (3 2 1) → (3 0 0) → (3 1 1) → (1 1 0) → (2 2 1) →
(0 2 0) → (0 3 1) → (0 1 0) → (0 2 1) → (0 0 0)



用语句叙述的解路径(即过河方案)如下:

- (1) 初始状态: 3个传教士、3个野人和船均在左岸;
- (2) 2个野人由左岸过河到右岸;
- (3) 1个野人划船返回左岸;
- (4) 2个野人(包括返回的那个)由左岸过河到右岸;
- (5) 1个野人划船返回左岸;
- (6) 2个传教士由左岸过河到右岸;
- (7) 1个传教士和一个野人返回左岸;
- (8) 两个传教士(包括返回的那个)由左岸过河到右岸;
- (9) 1个野人返回左岸;
- (10) 2个野人由左岸过河到右岸;
- (11) 1野人返回左岸;
- (12) 2个野人由左岸过河到右岸, 至此, 传教士与野人全部过河, 此时3个传教士、3个野人和船全在右岸。



例1

- 八数码游戏(eight puzzle)

2	3	7
	5	1
4	8	6

1	2	3
8		4
7	6	5



- 游戏说明:

- 一个棋盘有9个方格，放了8个数（1-8）；
- 初始时，8个数随机放置；
- 数字移动规则：空格周围的数字可移动到空格中；
- 如果通过移动数字，达到一个目标状态，则游戏成功结束；
- 求一个走步序列；

- 问题:

- 怎样用一个产生式系统描述并解决上述问题？



- 产生式系统的描述:

- 综合数据库: 存放棋盘的状态。

- ✓ 棋盘的状态: **8**个数字在棋盘上的位置分布。

- ✓ 每走一步, 状态就会发生变化;

- ✓ 存放棋盘的当前状态;

- 规则: 规则是数字移动的方法。

- ✓ 空格的移动:

- ✓ 如果空格左边有数字, 则将左边的数字移到空格上;

- ✓ 如果空格右边有数字, 则将右边的数字移到空格上;

- ✓ 如果空格上边有数字, 则将上边的数字移到空格上;

- ✓ 如果空格下边有数字, 则将下边的数字移到空格上;



➤ 控制机制:

- ✓ 用当前数据库与规则左半部分进行匹配，确定可执行的规则集；
- ✓ 从中选择一条规则执行，用规则的右半部分代替数据库中的状态；
- ✓ 如果当前数据库中的状态与目标状态相同，则终止；



例2

- 问题：设字符转换规则

$$A \wedge B \rightarrow C$$

$$A \wedge C \rightarrow D$$

$$B \wedge C \rightarrow G$$

$$B \wedge E \rightarrow F$$

$$D \rightarrow E$$

已知：A, B

求：F



一、综合数据库

$\{x\}$, 其中 x 为字符

二、规则集

1. IF $A \wedge B$ THEN C

2. IF $A \wedge C$ THEN D

3. IF $B \wedge C$ THEN G

4. IF $B \wedge E$ THEN F

5. IF D THEN E



三、控制策略

顺序排队

四、初始条件

$\{A, B\}$

五、结束条件

$F \in \{x\}$



求解过程

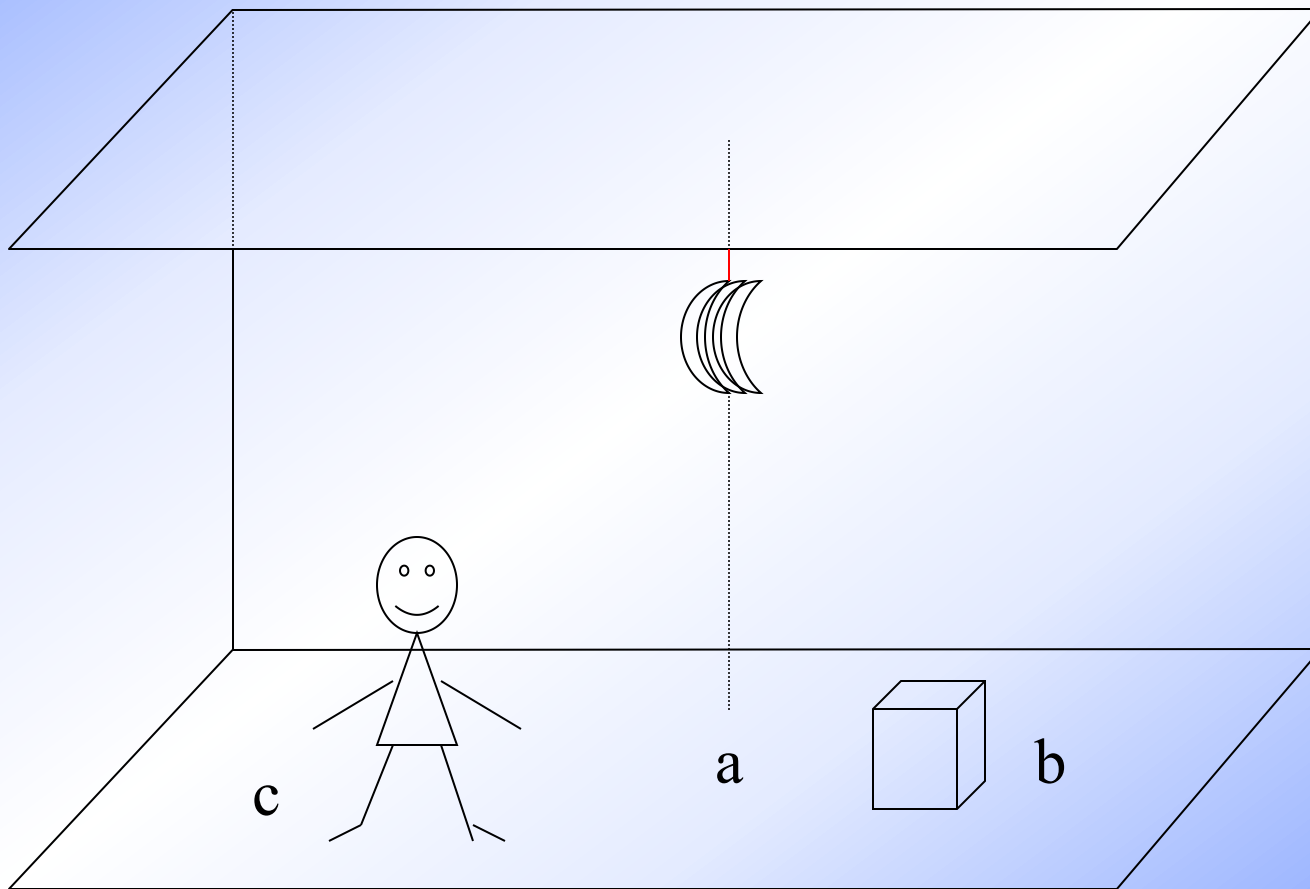
数据库	可触发规则	被触发规则
A, B	(1)	(1)
A, B, C	(2) (3)	(2)
A, B, C, D	(3) (5)	(3)
A, B, C, D, G	(5)	(5)
A, B, C, D, G, E	(4)	(4)
A, B, C, D, G, E, F		

1, IF $A \wedge B$ THEN C
3, IF $B \wedge C$ THEN G
5, IF D THEN E

2, IF $A \wedge C$ THEN D
4, IF $B \wedge E$ THEN F



例3: 猴子摘香蕉问题



- 综合数据库

(M, B, Box, On, H)

M: 猴子的位置

B: 香蕉的位置

Box: 箱子的位置

On=0: 猴子在地板上

On=1: 猴子在箱子上

H=0: 猴子没有抓到香蕉

H=1: 猴子抓到了香蕉



- 初始综合数据库

(c, a, b, 0, 0)

- 结束综合数据库

(x1, x2, x3, x4, 1)

其中**x1~x4**为变量。



(M, B, Box, On, H)

● **规则集**

r1: IF (x, y, z, 0, 0) THEN (w, y, z, 0, 0)

r2: IF (x, y, x, 0, 0) THEN (z, y, z, 0, 0)

r3: IF (x, y, x, 0, 0) THEN (x, y, x, 1, 0)

r4: IF (x, y, x, 1, 0) THEN (x, y, x, 0, 0)

r5: IF (x, x, x, 1, 0) THEN (x, x, x, 1, 1)

其中**x, y, z, w**为变量



r1: IF (x, y, z, 0, 0) THEN (w, y, z, 0, 0)表示猴子
走动

r2: IF (x, y, x, 0, 0) THEN (z, y, z, 0, 0)表示猴子
推动箱子

r3: IF (x, y, x, 0, 0) THEN (x, y, x, 1, 0)表示猴子
站到箱子上

r4: IF (x, x, x, 1, 0) THEN (x, x, x, 1, 1)表示猴子
摘取香蕉

其中**x, y, z, w**为变量



猴子摘香蕉问题 **monkey-and-banana problem** （解法二）

用一个四元组 (**W**, **x**, **Y**, **z**) 表示问题的状态:

W: 猴子的水平位置

x: 当猴子在箱子顶上时取**x=1**; 否则, 取**x=0**

Y: 箱子的水平位置

z: 当猴子摘到香蕉时取**z=1**; 否则, 取**z=0**



- 算符如下：

(1) **goto(U)**: 猴子走到水平位置**U**，用规则表示为
$$(W, 0, Y, z) \rightarrow (U, 0, Y, z)$$

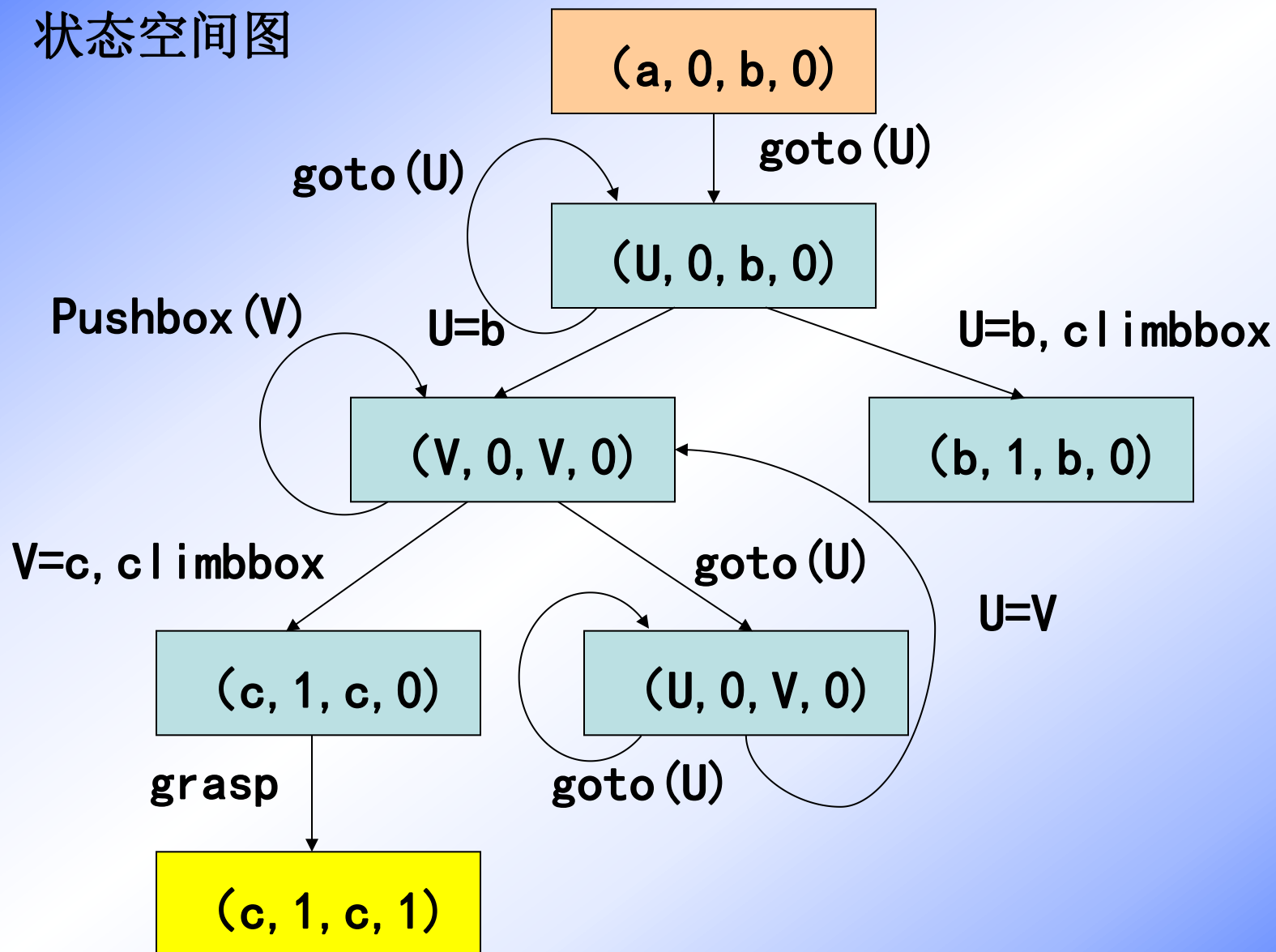
(2) **pushbox(V)**: 猴子把箱子推到水平位置**V**，即有
$$(W, 0, W, z) \rightarrow (V, 0, V, z)$$

(3) **climbbox**: 猴子爬上箱顶，即有
$$(W, 0, W, z) \rightarrow (W, 1, W, z)$$

(4) **grasp**: 猴子摘到香蕉，即有
$$(c, 1, c, 0) \rightarrow (c, 1, c, 1)$$



状态空间图



实例分析——

汽车故障诊断（用产生式系统实现）

- 图中给出了汽车故障诊断领域所发现的一些直觉形式知识片段。



汽车故障诊断领域的知识片段

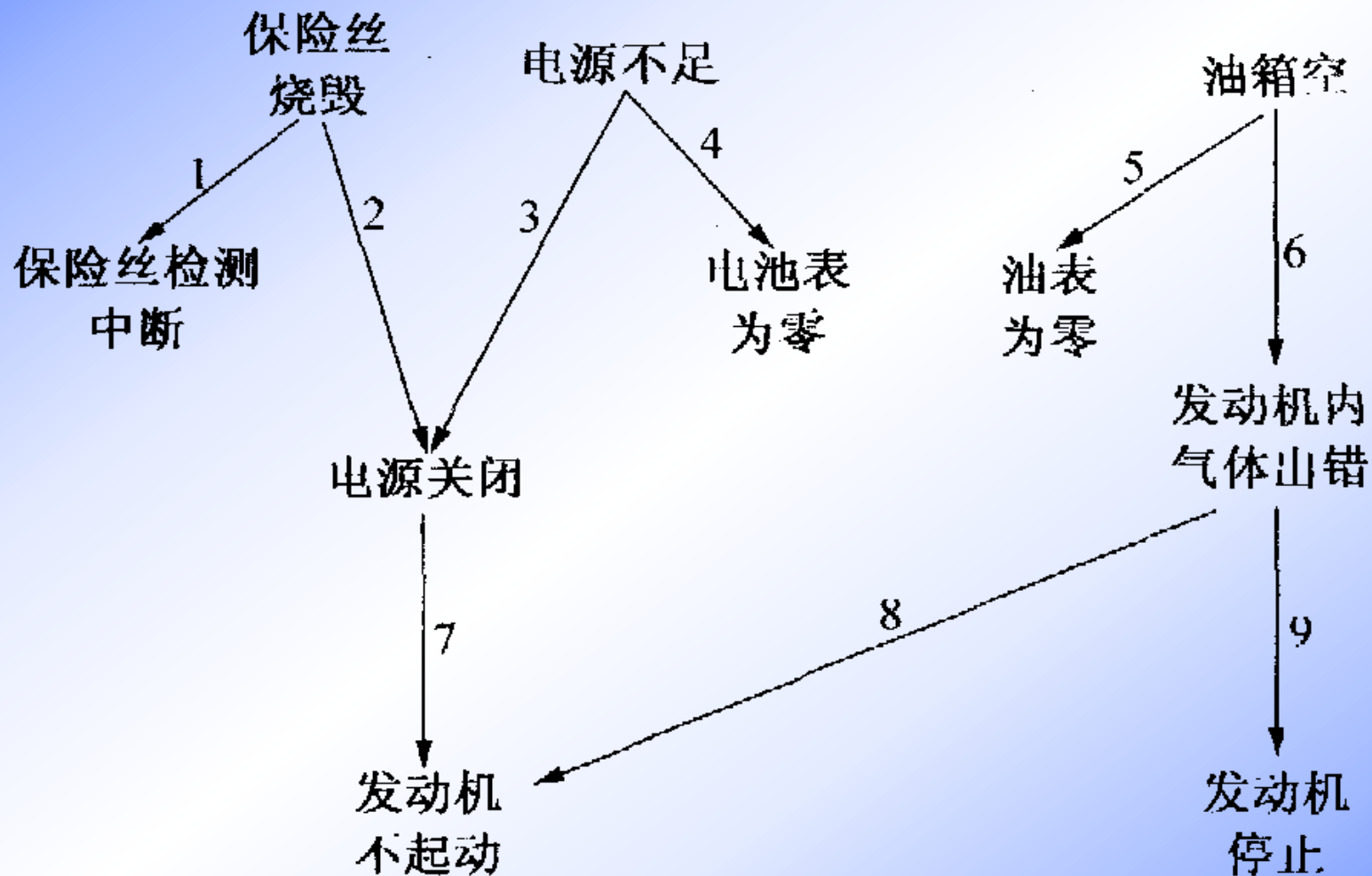


图2. 汽车故障诊断领域的知识片段



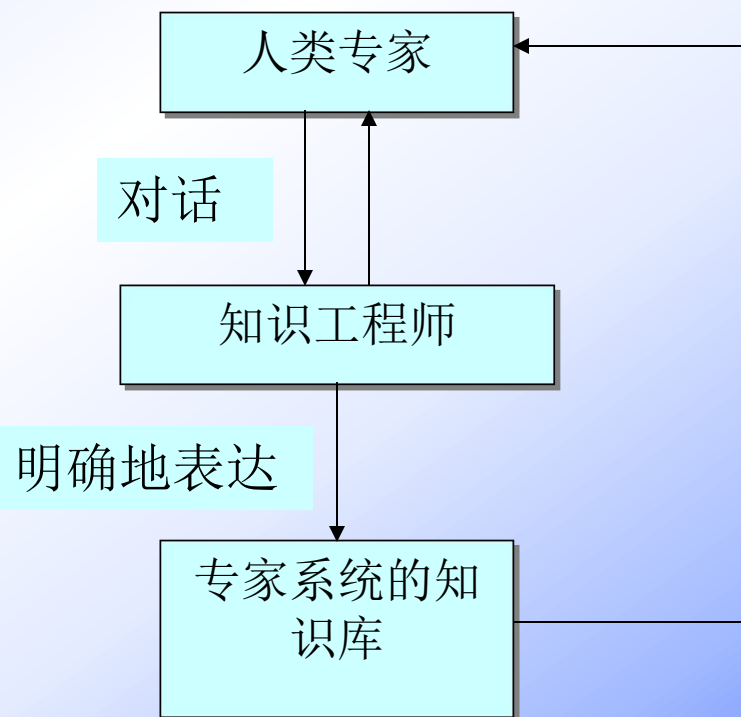
一般应用程序与产生式系统的区别

- 前者把问题求解的知识隐含地编入程序，而后者则把其应用领域的问题求解知识单独组成一个实体, 即为知识库。
- 更明确地说，一般应用程序把知识组织为两级：数据级和程序级；
- 产生式系统则将知识组织成三级：数据、知识库和控制。



知识获取

- ❖ 静态知识模型的建立
- ❖ 知识库的建立



术语标注和概念抽象

一、可观察的方面：

保险丝检测、电池表、油表

二、汽车状态：

- 不可见状态：保险丝、电池、油箱、电源、发动机汽体
- 可见状态：发动机行为。



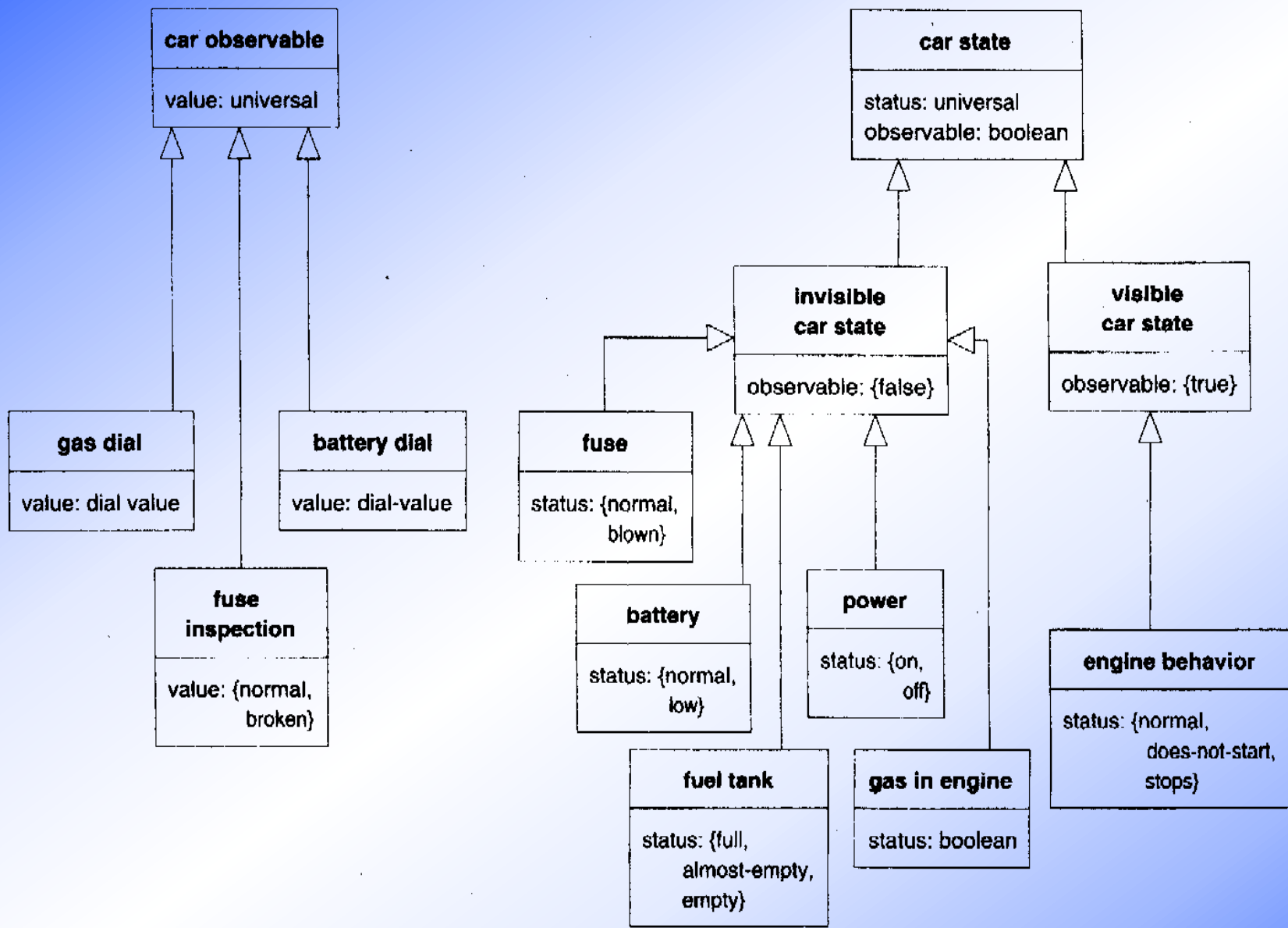


图2. 汽车诊断领域中概念之间的子类型关系



规则的形式

从图中得出汽车状态间依赖性规则：

规则6: FUEL-TANK. status=empty==>

GAS-IN-ENGINE. status=false

规则3: BATTERY. status=low==>

POWER. status=off



规则类型1:对应图1中的6个箭头 (2-3和6-9)。

```
RULE-TYPE state-dependency;  
    ANTECEDENT: invisible-car-state;  
                CARDINALITY: 1;  
    CONSEQUENT: car-state;  
                CARDINALITY: 1;  
    CONNECTION—SYMBOL:  
                causes;  
END RULE-TYPE state-dependency;
```



规则类型2： 对应图1中的3个箭头（1、4、5）

RULE-TYPE **manifestation-rule**;

DESCRIPTION: “Rule stating the relation between an internal state and its external behavior in terms of observable value”;

ANTECEDENT:

invisible-car-state;

CONSEQUENT:

car-observable;

CONNECTION-SYMBOL:

has-manifestation;

END RULE-TYPE **manifestation-rule**;



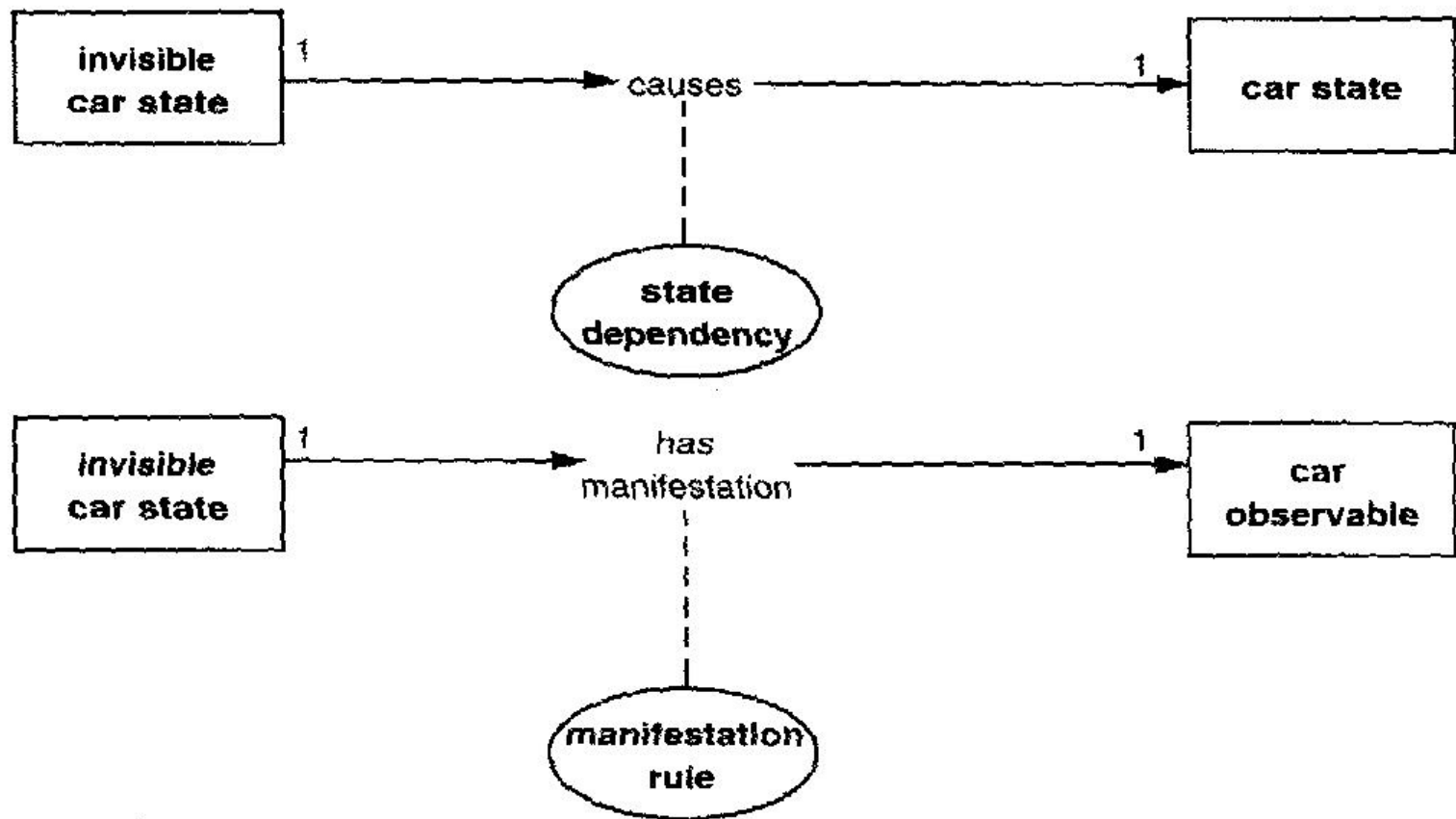


图3 规则类型的图形表示

注：有向线段从前件画到连接符号，再从连接符号到后件。规则类型的名称放在一个椭圆框中并用虚线与连接符号连接，数字表示基数〔前件或后件中表达式的最大值和最小值〕。



规则-类型结构能够将知识库建立为只有相似结构的更小的划分(如规则集)。

fule-supply. status=blocked

CAUSES

gas-in-engine. status=false;

battery. status = low

CAUSES

power. value = off;



图2中的子类型层次为通常规则中的前件和后件提供了类型。图3给出了一个规则类型的图形表示，用汽车领域中的两个规则类型作为例子。有向线段从前件画到连接符号，再从连接符号到后件。规则类型的名称放在一个椭圆框中并用虚线与连接符号连接。由于规则类型和“作为类的关系”具有相似性，所以使用虚线：两个例子都是复杂实体。与线相连的数字表示基数。基数用于给出前件或后件中表达式的最大值和最小值的限制。在这个例子中规则必须只有一个条件和结论。



知识库

具有**state—dependency**(状态—依赖性)和**manifestation**(表象—规则)规则类型的实例的知识库规范说明包括两部分。

1)USE槽定义哪种类型的领域知识实例存储在知识库中。格式如下：

`< type > FROM < domain schema >`

其中，后半部分定义在哪个领域模式中定义类型。在汽车例子中只有一个模式，在更加复杂的应用中经常需要引入多重领域模式。



2) EXPRESSION槽包含实际实例。可以用半正式方式描述规则实例，其中的连接符号用于从后件中分离前件表达式。在这里也可以使用正式语言。但应注意，分析过程中知识库的形式和范围极易变化。因此在知识类型尚未被确认和生效时，避免规则实例过于形式化十分重要。

图4给出了一个知识库的例子，其中包含汽车应用的因果模型。它使用图2中定义的两个规则类型。图4中的实例对应图1中列出的知识片段。



KNOWLEDGE-BASE car-network;

USES:

state-dependency FROM car-diagnosis-schema,
manifestation-rule FROM car-diagnosis-schema;

EXPRESSIONS:

/ *state dependencies* /

fuse. status=blown GAUSES power. status = off;
battery. status = low CAUSES power. status=off;
power. status = off CAUSES

engine-behavior. status=does-not-start;
fule-tank. status=empty CAUSES gas-in—engine. status = false;
gas-1n-engine. status = false CAUSES
engine-behavior. status = does-not-start;
gas-in-engine. status=false CAUSES
engine-behavior. status=slops;

/ *manifestation rules* /

fuse. status = blown HAS-MANIFESTATION
fuse-inspection. value = broken;
battery. status = low HAS-MANIFESTATION battery-dial. value=zero;
fuel-tank. status=empty HAS-MANIFESTATION gas-dial. value=zero;
END KNOWLEDGE-BASE car-network;

图4. 包含两种规则类型实例的“汽车网络”知识库



产生式系统的特点

- 规则的形式固定：
 - 规则分为左半部分和右半部分；
 - 左半部分是条件，右半部分是结论；
- 知识模块化：
 - 知识元：通俗的说，知识元是产生式规则的条件中的独立部分， A_i ；所有的规则或数据库中的数据都是由知识元构成；



产生式系统的特点(续)

- 产生式之间的相互影响是间接的
 - 产生式之间的作用通过综合数据库的变化完成，因此是数据驱动的；
 - 易扩展：规则的添加和删除较为自由，因为没有相互作用；
 - 添加规则不能造成矛盾； $A \rightarrow B$ ， $A \rightarrow \sim B$ 。



Exercises P. 220

1. c

2.

3.

5

7

9.

10.



1. (a) The member algorithm of Section 6.1.1 recursively determines whether a given element is a member of a list.

(b) Write an algorithm to count the number of elements in a list.

(c) Write an algorithm to count the number of atoms in a list.

(The distinction between atoms and elements is that an element may itself be a list.)

The approach question 1b. requires is to move recursively down the list counting each element in the list. That element may either be an atom or a structure, i.e., even if the structure has atoms within it, it is only counted once, as an element of the list.



1c. requires that each element of the list be broken down into its constituent atoms, and then these atoms counted. This is often called “car-cdr” recursion, because not only does the algorithm recurse down the list but also takes apart each structure that makes up the list. An example answer may be found in the `length` and `count_atoms` LISP functions in Figure 16.3, Section 16.1.7.



2. Write a recursive algorithm (using open and closed lists) to implement breadth-first search. Does recursion allow the omission of the open list when implementing breadth-first search? Explain.

The open list may not be omitted, since it is maintained as a queue and not a stack. The recursive environment mimics the action of a stack. A queue, on the other hand must be explicitly maintained, as the results of each recursive call are understood. This is discussed in detail for PROLOG, and an algorithm given, in Section 15.4.



3. Trace the execution of the recursive depth-first search algorithm (the version that does not use an open list) on the state space of Figure 6.5.

Of course, the result will be the same as the version that does use the open list! The recursive environment itself operates as a stack. To get the results required, assume loop detection in the algorithm, assume a depth bound, and evaluate the next state by taking the production rules in order. Note that the order of production rules generating the graph of Figure 3.14 differs from the order of the rules in Figure 6.5. Thus, your result will have a different search space.



5. Using the move and path definitions for the knight's tour of Section 6.2, trace the execution of `pattern_search` on the goals:

(a) `path(1,9)`.

(b) `path(1,5)`.

(c) `path(7,6)`.

When the move predicates are attempted in order, c leads to looping in the search. It can be important to discuss loop detection and backtracking in this situation.

5a. Produce a trace like that of Figure 6.7:

Iteration	current state	goal state	conflict rules	rule fired
0	1	9	1,2	1
1	8	9	13,14	13
2	3	9	5,6	5
3	4	9	7,8	7
4	9	9		halt



5c. Again, as in Figure 6.6

Iteration	current state	goal state	conflict rules	rule fired
0	7	6	11,12	11
1	2	6	3,4	3
2	9	6	15,16	15
3	2	6	3,4	3
4	9	6	15,16	15
etc.				

Obviously we need loop detection in our algorithm for the production system; we have this, of course, when we implement our search algorithms as in Chapter 3. Note also that the original query had a one move direct solution that was missed by our algorithm.

Bestfirst search can remedy (处理) this problem.



7. Using the rule in Example 5.3.3 as a model, write the eight move rules needed for the full 8 x 8 version of the knight's tour.

The forms of these rules will vary, of course but they will all add + or - 2 and 1 to the row and column measures of the present state to produce the new state. At some time in the production of the new state we must check that it will be on the board. This can be done as in the text by checking the current state to see if the present rule can be applied, or as in our example below assuming the current state is okay (ok) and then constraining the new state.



The representation used here is a PROLOG-like version of the predicate calculus where each state or board position is called “state(Row, Col):”

CONDITION: state(Row, Col)

ACTION: state(NewRow, NewCol) \wedge NewRow is Row - 2 \wedge
NewRow > 0

NewCol is Col + 1 \wedge NewCol < 9.

The failure of any of the and constraints gets the system to consider the next rule or backtrack. The reader can create the seven other rules.



9. Consider the **financial advisor problem** discussed in Chapters 2, 3, and 4. Using predicate calculus as a representation language:

- (a) Write the problem explicitly as a production system.
- (b) Generate the state space and stages of working memory for the data-driven solution to the example in Chapter 3.
- 10.(c) Repeat b for a goal-driven solution.

For this answer we refer to the rules presented in Section 2.4.

9a. Put rules 1 through 8 in the production memory. Place the minsavings and minincome functions in the production memory also, to be used whenever a rule requires this mathematical calculation performed. Facts 9, 10, and 11 will be obtained from the used as needed. Now run the production as in either 9b or 9c.



9b. Go through the rules in order until you produce the result “savings(X)” for some X. When no premise is yet known, as in the first three rules in the first iteration, go on to the next rule. Finally, in rule 4 we are asked about savings and dependents. There are no rules that can conclude these results so the system goes to the user to obtain this information. When it is supplied, minsavings is calculated and then either rule 4 or rule 5 fires to conclude about the adequacy of the savings account. This new fact is added as a result that is now known by the system. At this point if it is breadth – first data driven search we continue through the remaining 3 rules, obtaining information on earnings and calculating minincome. Otherwise, if we are doing depth-first search we take the result of rule 4 or 5 and go back to rule 1 again.

9c. Search should look something like Figure 3.26, depending only on the rule ordering. How this graph is generated depends on whether the goal driven search is breadth-first or depth-first.



11. Section 6.2.3 presented the general conflict resolution strategies of refraction, recency, and specificity. Propose and justify two more strategies.

Two easy additional strategies would be to keep the response as general as possible, and thus not cut down on the set of possible solutions deducible by the production system, and similarity to previous firing conditions that might make it easier to perform repeated similar tasks, such as iteration or looping. These strategies would have to be spelled out in terms of the currently activated rules.



12. Suggest two applications appropriate for solution using the blackboard architecture. Briefly characterize the organization of the blackboard and knowledge sources for each implementation.

The blackboard is now a very popular problem-solving paradigm (典范). One important application would be to monitor (监控) all the processes that must come together in computer assisted manufacturing (计算机辅助制造). Another might be to describe all the diverse (多样的) reasoning processes that can come together in a diagnosis problem (Skinner and Luger 1992).

