Control & Implementation of State Space Search

Dr. Keerthy A S

Control & Implementation of State Space Search

- Recursion Based Search
- Pattern Directed Search
- Production Systems
- Blackboard Architecture for Problem Solving

Journey so far…

- Representation of a problem solution as a path from a start state to a goal
- Systematic search of alternative paths
- Backtracking from failures
- Explicit records of states under consideration
  - open list: untried states
  - closed lists: to implement loop detection
- open list is a stack for DFS, a queue for BFS

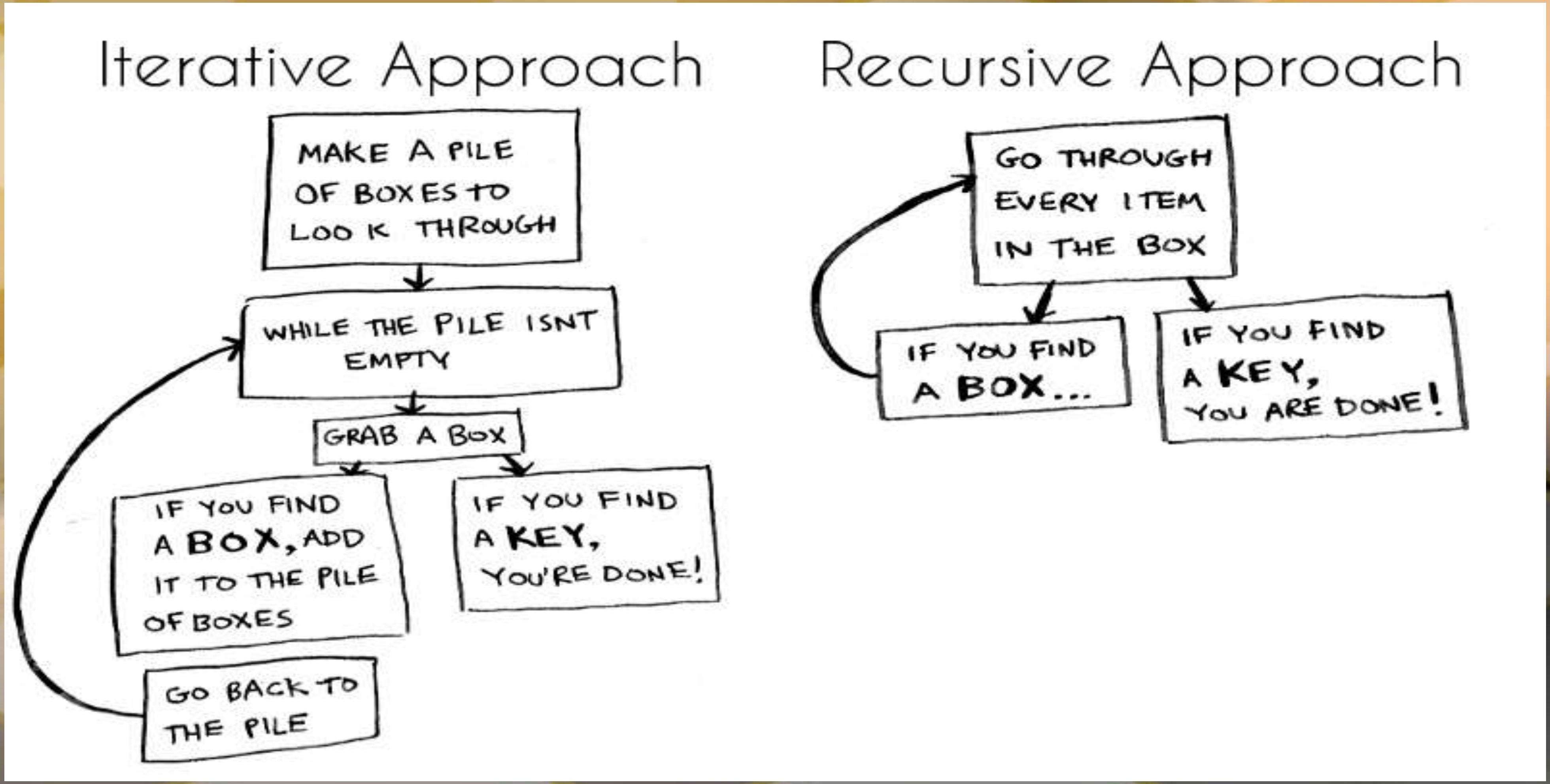
Recursion-Based Search

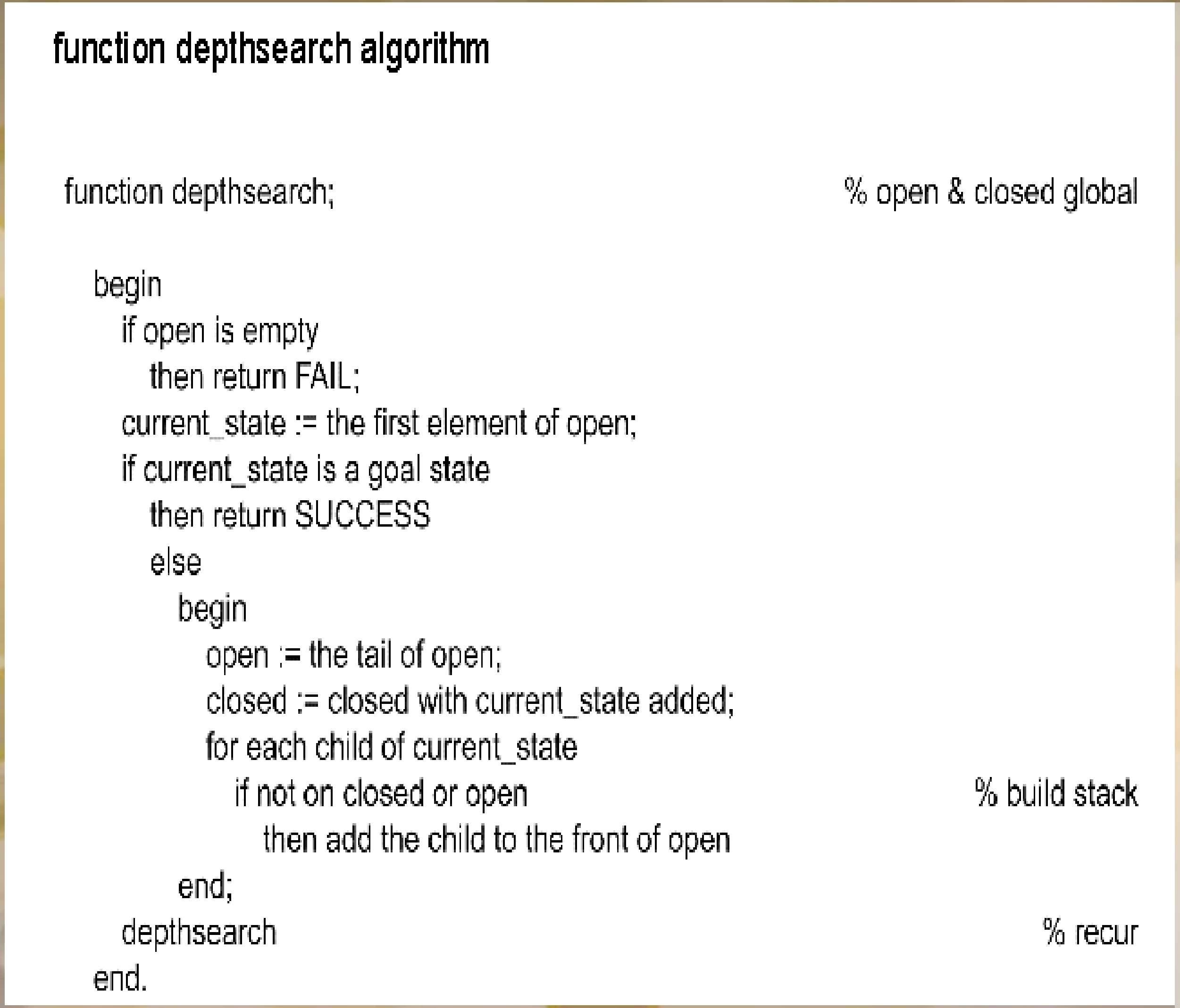
- Recursion is used to define and analyze both data structures and procedures

A recursive procedure consists of:

- A recursive steps: the procedure calls itself to repeat a sequence of actions
- A terminating condition that stops the procedure from recurring endlessly (the recursive version of an endless loop)

Recursion Based Search


Iterative Approach
MAKE A PILE OF BOXES TO LOOK THROUGH
WHILE THE PILE ISNT EMPTY
GRAB A BOX
IF YOU FIND A BOX, ADD IT TO THE PILE OF BOXES
IF YOU FIND A KEY, YOU'RE DONE!
GO BACK TO THE PILE
Recursive Approach
GO THROUGH EVERY ITEM IN THE BOX
IF YOU FIND A BOX...
IF YOU FIND A KEY, YOU ARE DONE!

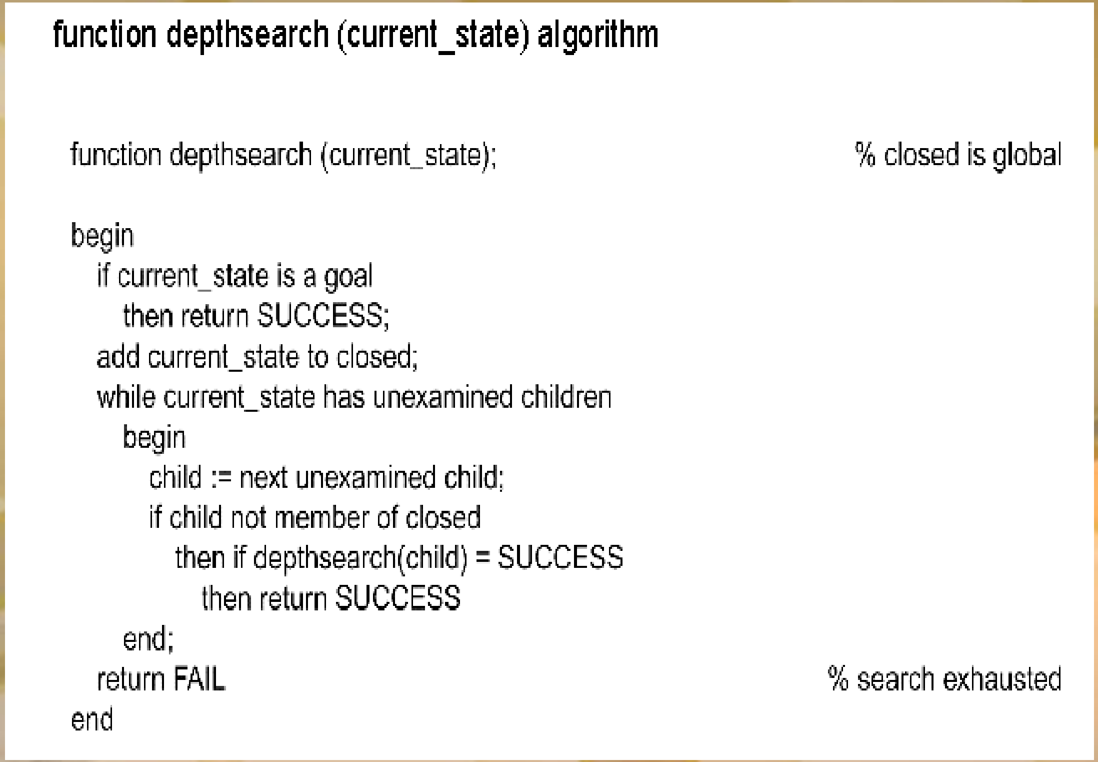


function depthsearch algorithm

```
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
      else
        begin
          open := the tail of open;
          closed := closed with current_state added;
          for each child of current_state
            if not on closed or open                   % build stack
              then add the child to the front of open
        end;
    depthsearch                                        % recur
  end.
```

- Does not utilize the full power of recursion
- Use recursion to organize states and paths through state space
- Global closed list – detects duplicate states and avoid loops & open list has activation records of recursive environments



function depthsearch (current_state) algorithm

```
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 not member of closed
        then if depthsearch(child) = SUCCESS
          then return SUCCESS
    end;
  return FAIL                                            % search exhausted
end
```

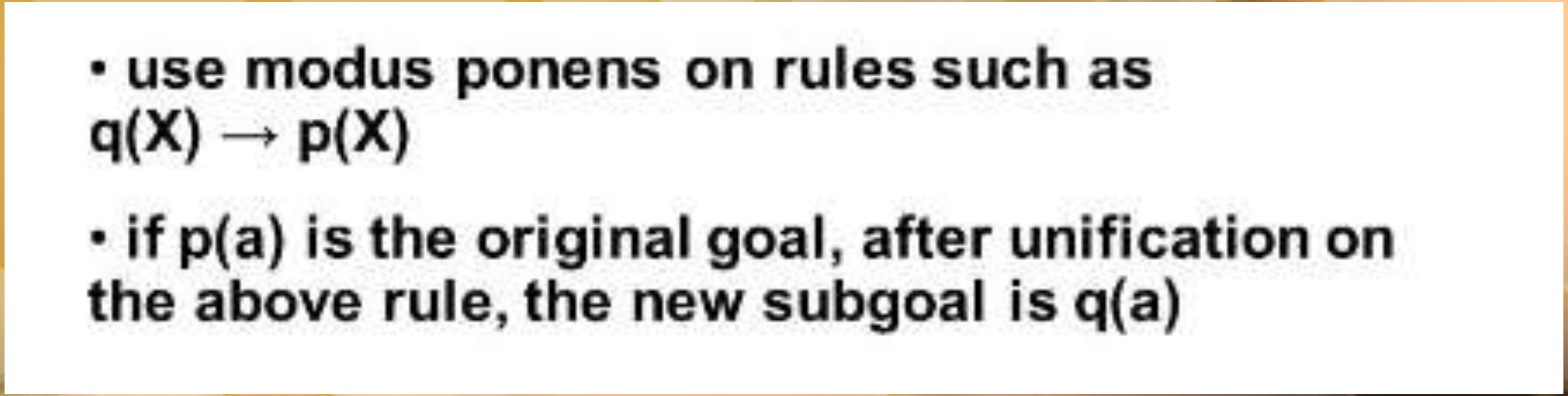
- Produces child states one at a time and recursively searches the descendants of each child before generating its siblings
- If some descendant of the child state is a goal the recursive call returns “success” and ignores the siblings
- If the recursive call fails to find the goal, the next sibling is generated and all of its descendants are searched→ Backtracking

Use recursion

- for clarity, compactness, and simplicity
- call the algorithm recursively for each child
- the open list is not needed anymore, activation records take care of this
- still use the closed list for loop detection

Pattern-Directed Search

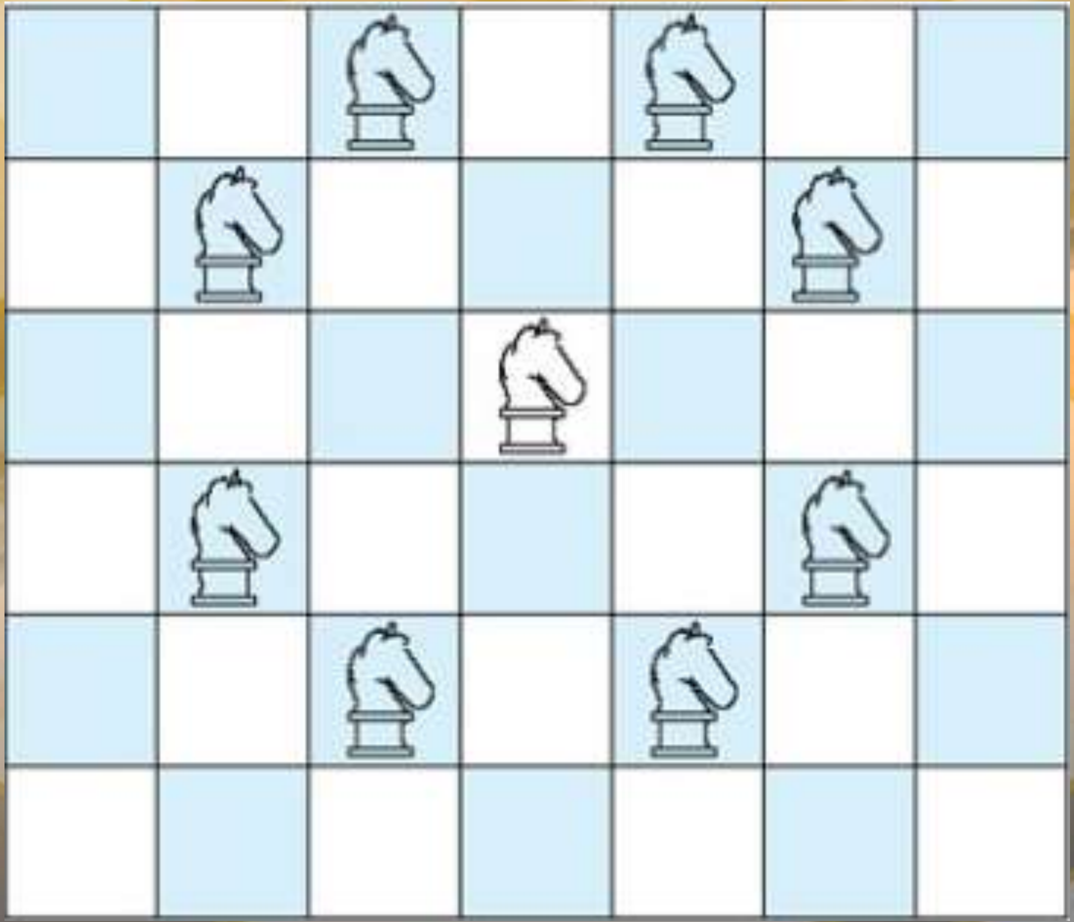
- The pattern-directed search uses **unification** to determine when two expressions match, and it uses **modus ponens** to generate the children states.

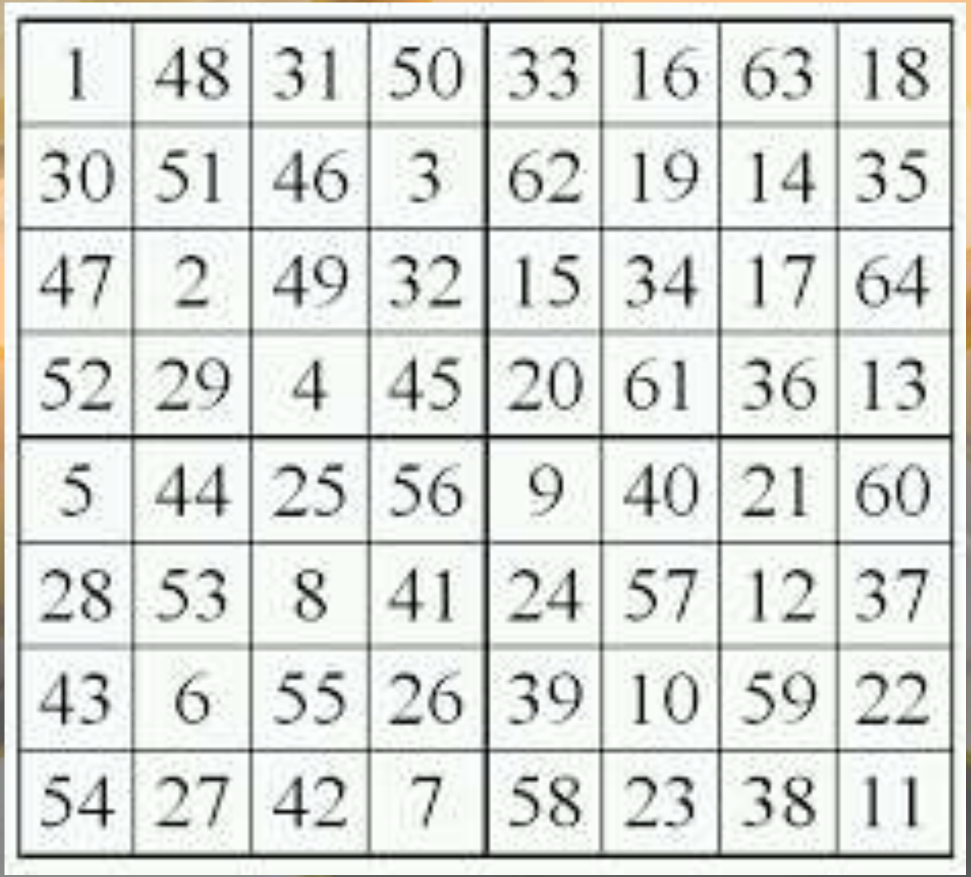


- use modus ponens on rules such as
$q(X) \rightarrow p(X)$
- if p(a) is the original goal, after unification on the above rule, the new subgoal is q(a)

Legal moves of a chess knight

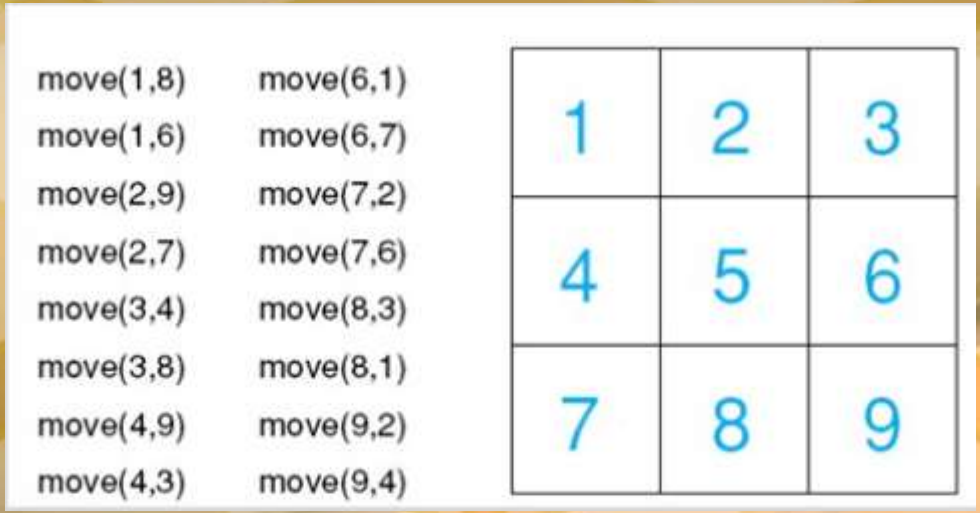
A knight’s tour is a series of moves that knight makes while visiting every cell of n x n of the chessboard exactly once.





1	48	31	50	33	16	63	18
30	51	46	3	62	19	14	35
47	2	49	32	15	34	17	64
52	29	4	45	20	61	36	13
5	44	25	56	9	40	21	60
28	53	8	41	24	57	12	37
43	6	55	26	39	10	59	22
54	27	42	7	58	23	38	11

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



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

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

Production Systems

Production Systems

- The production System is a model of computation that has proved particularly important in AI, Both for implementing search algorithms and for modeling human problem solving.
- A production system provides pattern-directed control of a problem solving process and consists of:
  - a set of production rules,
  - a working memory, and
  - a recognize- act control cycle.

- A schematic drawing of a production system is as shown in figure below:

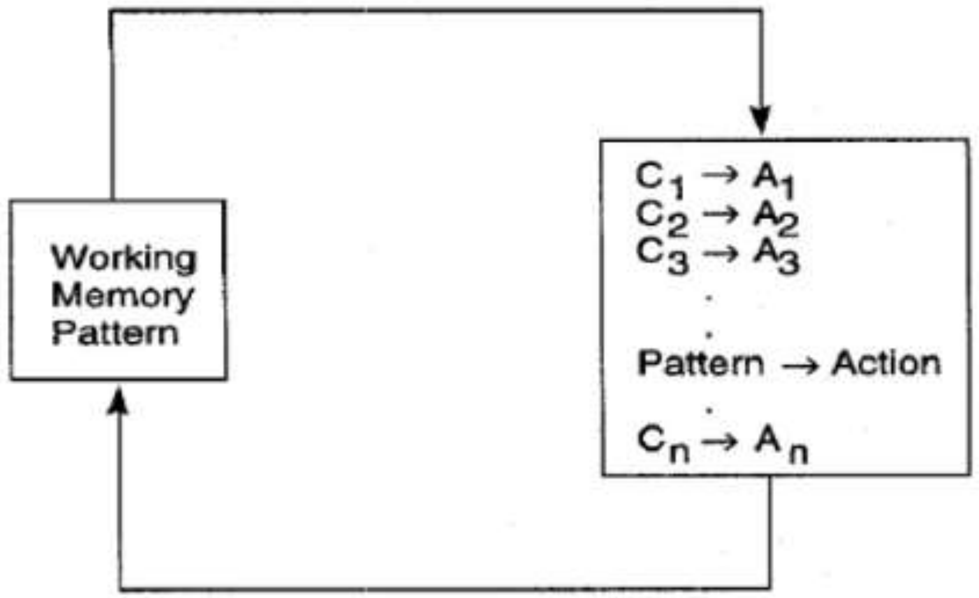


Fig: A production system

- Control loops until working memory pattern no longer matches the condition of any productions.

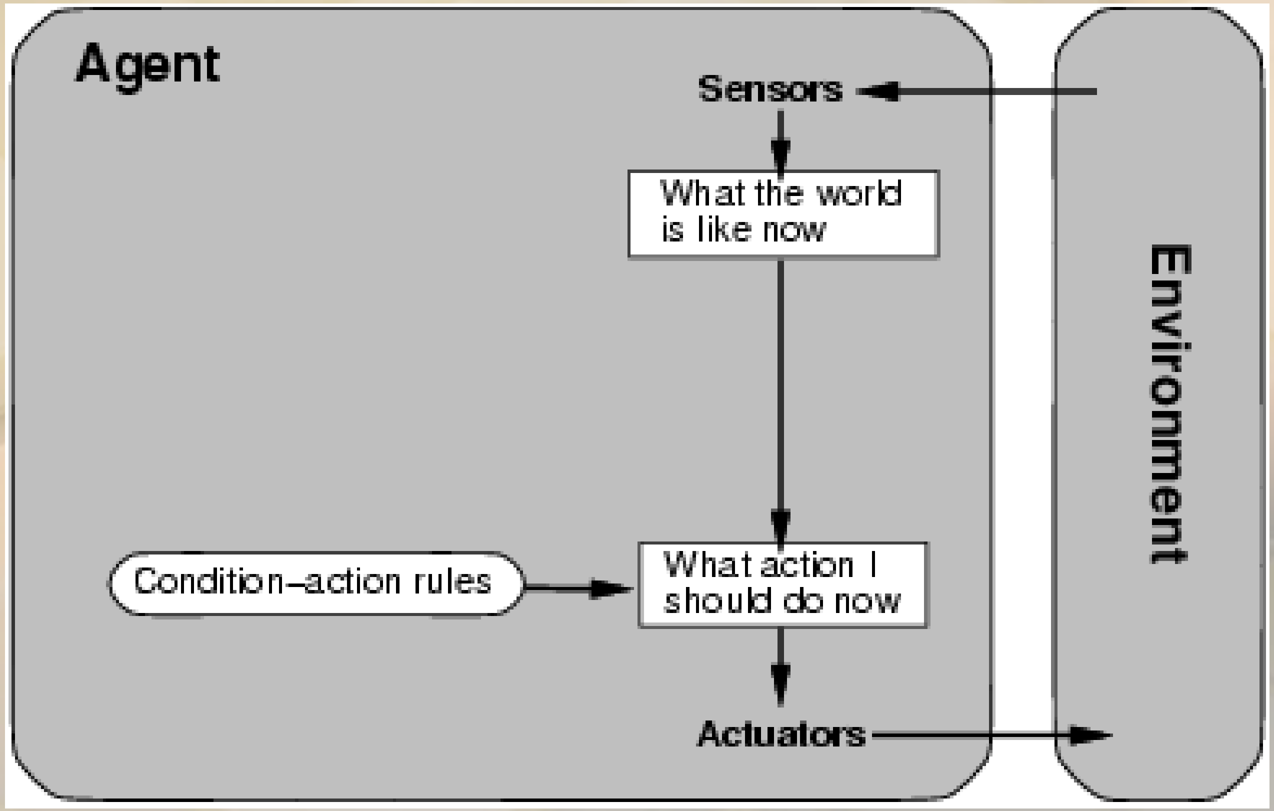
- The set of production rules:
  - These are often simply called productions. A production is a condition-action pair and defines a single chunk of problem-solving knowledge.
    - The condition part of the rule is a pattern that determines when that rule may be applied to a problem instance.
    - The action part defines the associated problem-solving step.
  - E.g., If it is raining then open the umbrella.

- Working memory:
  - Working memory contains a description of the current state of the world(facts/ data) in a reasoning process.
  - This description is a pattern that is matched against the condition part of a production to select appropriate problem solving actions.
  - When the condition part of a rule is matched by the contents of working memory, the action associated with that condition may then be performed.
  - The actions of production rules are specifically designed to alter the contents of working memory.

- The recognize- act cycle: The control structure for a production system is simple:
  - Working memory is initialized with the beginning problem description 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, whose conditions match the patterns in working memory.
  - One of the productions in the conflict set is then selected(conflict resolution) and the production is fired.
  - To fire a rule, its action is performed, changing the contents of working memory.
  - After the selected production rule is fired, the control cycle repeats with the modified working memory.
  - This process terminates when the contents of working memory do not match any rule conditions.

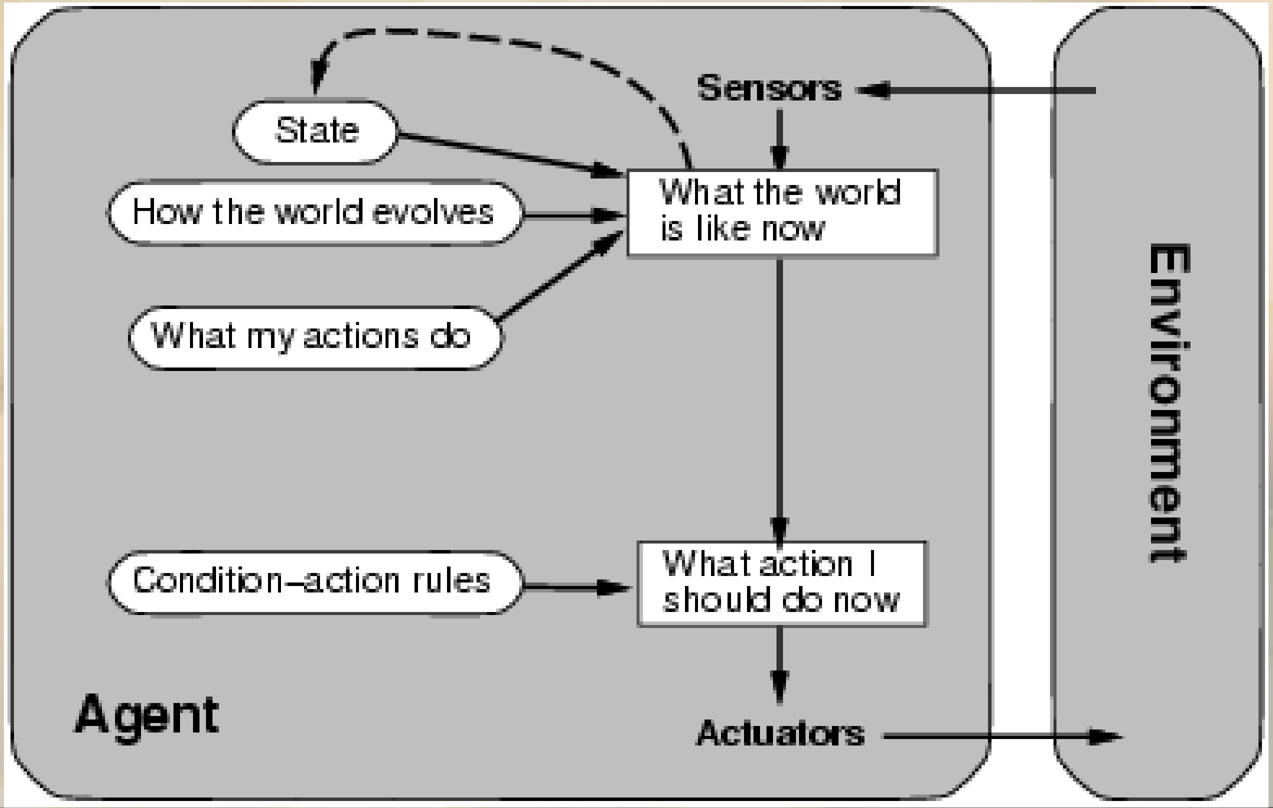
Simple reflex agent

- Select action on the basis of current percept, ignoring all past percepts



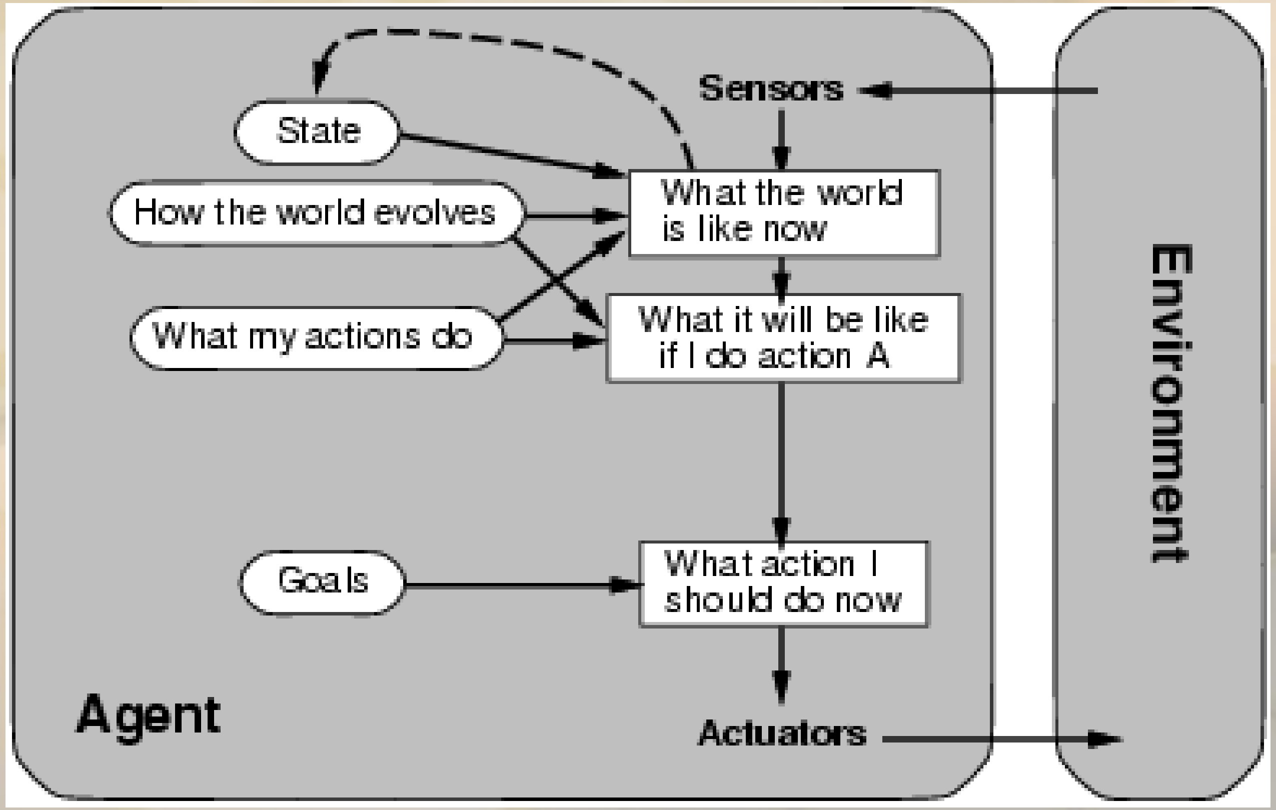
Model-based reflex agent

- Maintains internal state that keeps track of aspects of the environment that cannot be currently observed



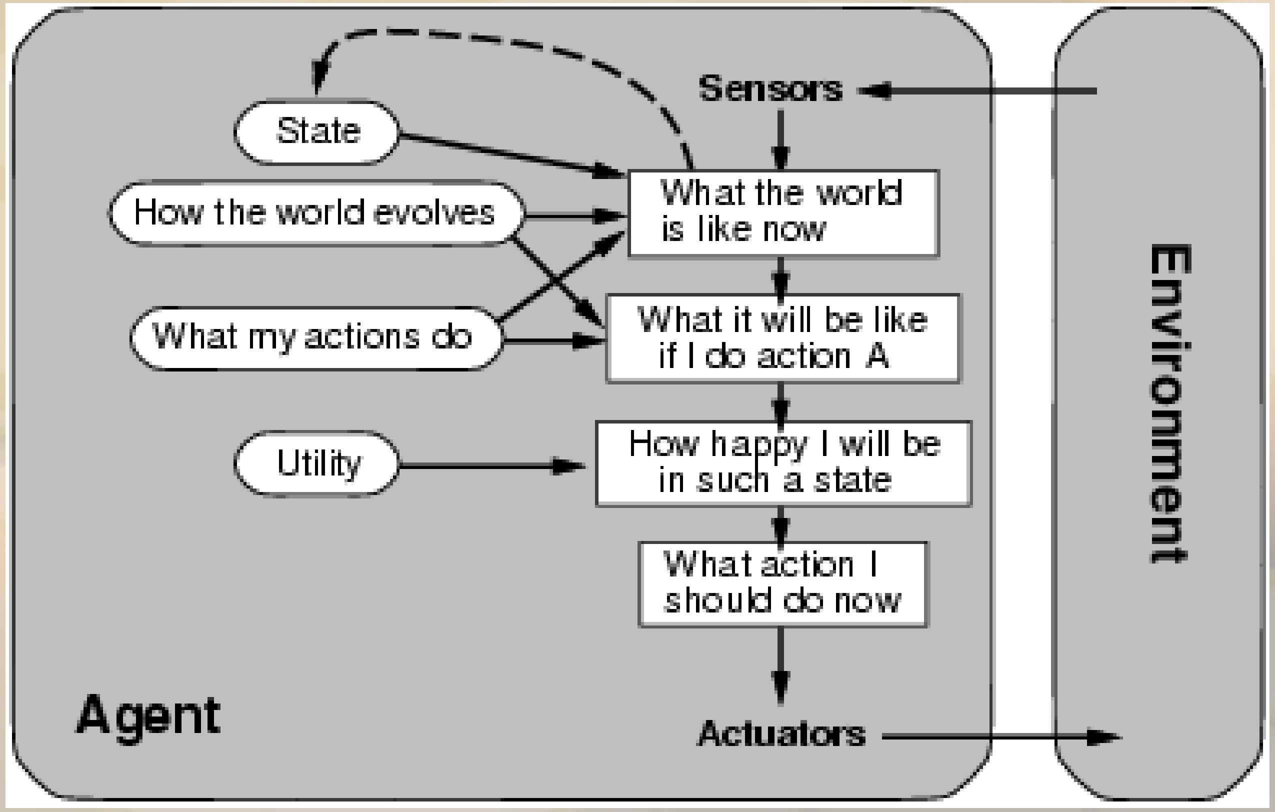
Goal-based agent

- The agent uses goal information to select between possible actions in the current state


Sensors
State
How the world evolves
What my actions do
What the world is like now
What it will be like if I do action A
Goals
What action I should do now
Agent
Actuators
Environment

Utility-based agent

- The agent uses a utility function to evaluate the desirability of states that could result from each possible action


Sensors
State
How the world evolves
What my actions do
Utility
What the world is like now
What it will be like if I do action A
How happy I will be in such a state
What action I should do now
Agent
Actuators
Environment

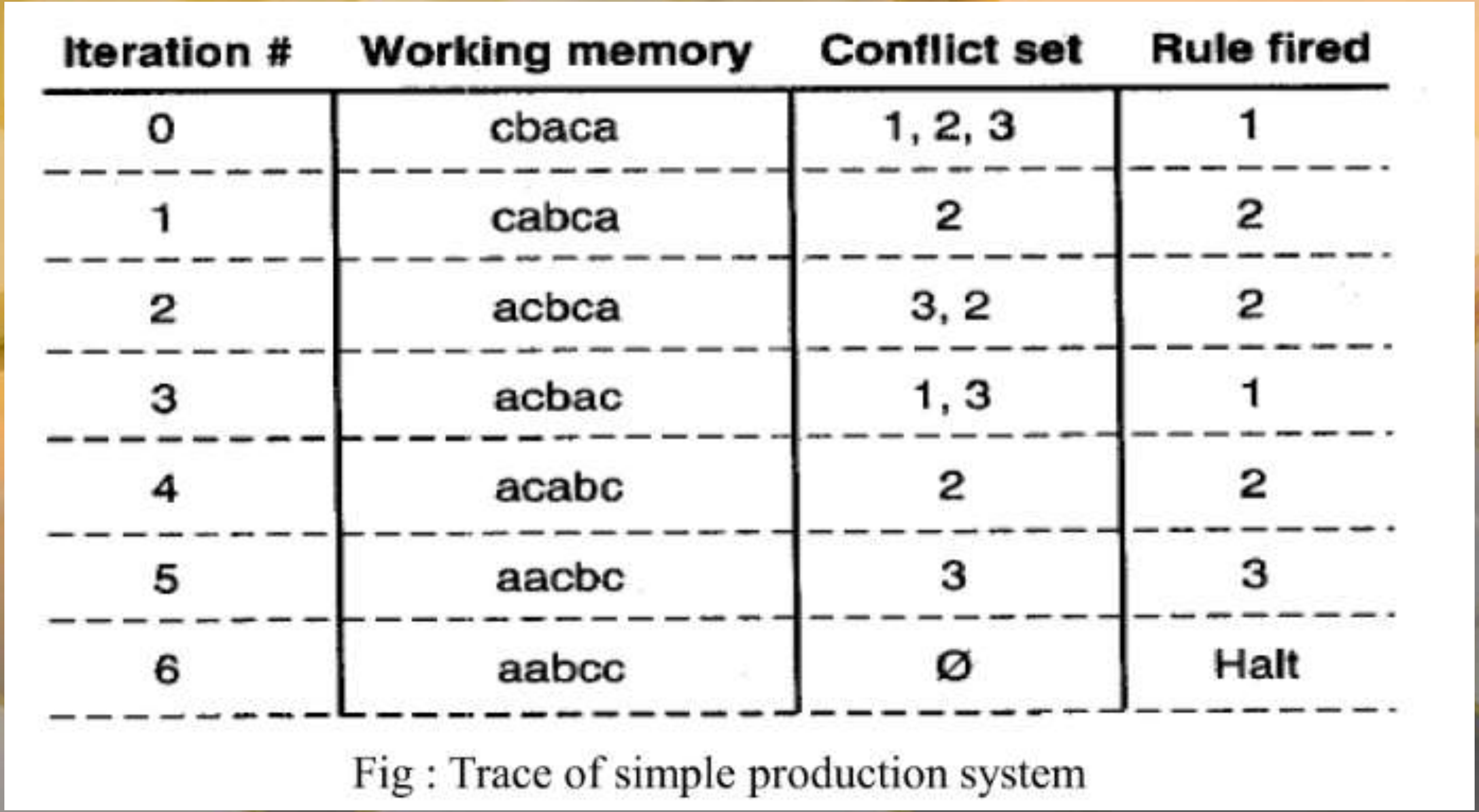
- Example of production system execution:
  - Problem: sorting a string composed of the letters a, b, and c.
  - Working memory: cbaca
  - Production set:

```
1. ba → ab
2. ca → ac
3. cb → bc
```

  - In this example, a production is enabled if its condition matches a portion of the string in working memory.
  - When a rule is fired, the substring that matched the rule condition is replaced by the string on the right hand side of the rule.

- Working memory: cbaca
- 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	3, 2	2
3	acbac	1, 3	1
4	acabc	2	2
5	aacbc	3	3
6	aabcc	Ø	Halt

Fig : Trace of simple production system

Advantages of Production Systems for AI

- The production system offers a general framework for implementing search.
- Because of its simplicity, modifiability, and flexibility in applying problem solving knowledge, the production system has proved to be an important tool for the construction of expert systems and other AI applications.
- The major advantages of production systems for artificial intelligence include:
  - Separation of knowledge and control
  - A natural mapping onto state space search.
  - Modularity of production rules.
  - Pattern directed control.
  - Opportunities for heuristic control of search
  - Tracing and explanation.
  - Language independence.
  - A plausible model of human problem solving

- Separation of knowledge and Control:
  - The production system is an elegant model of separation of knowledge and control in a computer program.
  - Control is provided by the recognize-act cycle of the production system loop, and the problem –solving knowledge is encoded in the rules themselves.
  - The advantages of this separation include ease of modifying the rule base without requiring a change in the code for program control and , conversely, the ability to alter the code for program control without changing the set of production rules.

- A Natural mapping onto state space search:
  - The components of a production system map naturally into the constructs of state space search. The successive states of working memory form the nodes of a state space graph.
  - The production rules are the set of possible transitions between states, with conflict resolution implementing the selection of a branch in the state space. These rules simplify the implementation, debugging, and documentation of search algorithms

- Modularity of production rules:
  - An important aspect of the production system model is the lack of any syntactic interactions between production rules. Rules may only effect the firing of other rules by changing the pattern in working memory.
  - This syntactic independence supports the incremental development of expert systems by successively adding, deleting, or changing the knowledge (rules) of the system.

- Pattern directed control:
  - The problems addressed by AI programs require particular flexibility in program execution.
  - This goal is served by the fact that the rules in a production system may fire in any sequence.
  - The descriptions of a problem that make up the current state of the world determine the conflict set and, consequently, the particular search path and solution.
- Tracing and Explanation:
  - The modularity of rules and the iterative nature of their execution make it easier to trace execution of a production system.
  - Because each rule corresponds to a single chunk of problem-solving knowledge, the rule content should provide a meaningful explanation of the system's current state and action

- Language Independence:
  - The production system control model is independent of the representation chosen for rules and working memory, as long as that representation supports pattern matching.
- A Plausible Model of human problem solving:
  - Modeling human problem solving was among the first uses of production systems.
  - E.g.,
    - IF the 'traffic light' is green
    - THEN the action is go
    - IF the 'traffic light' is red
    - THEN the action is stop

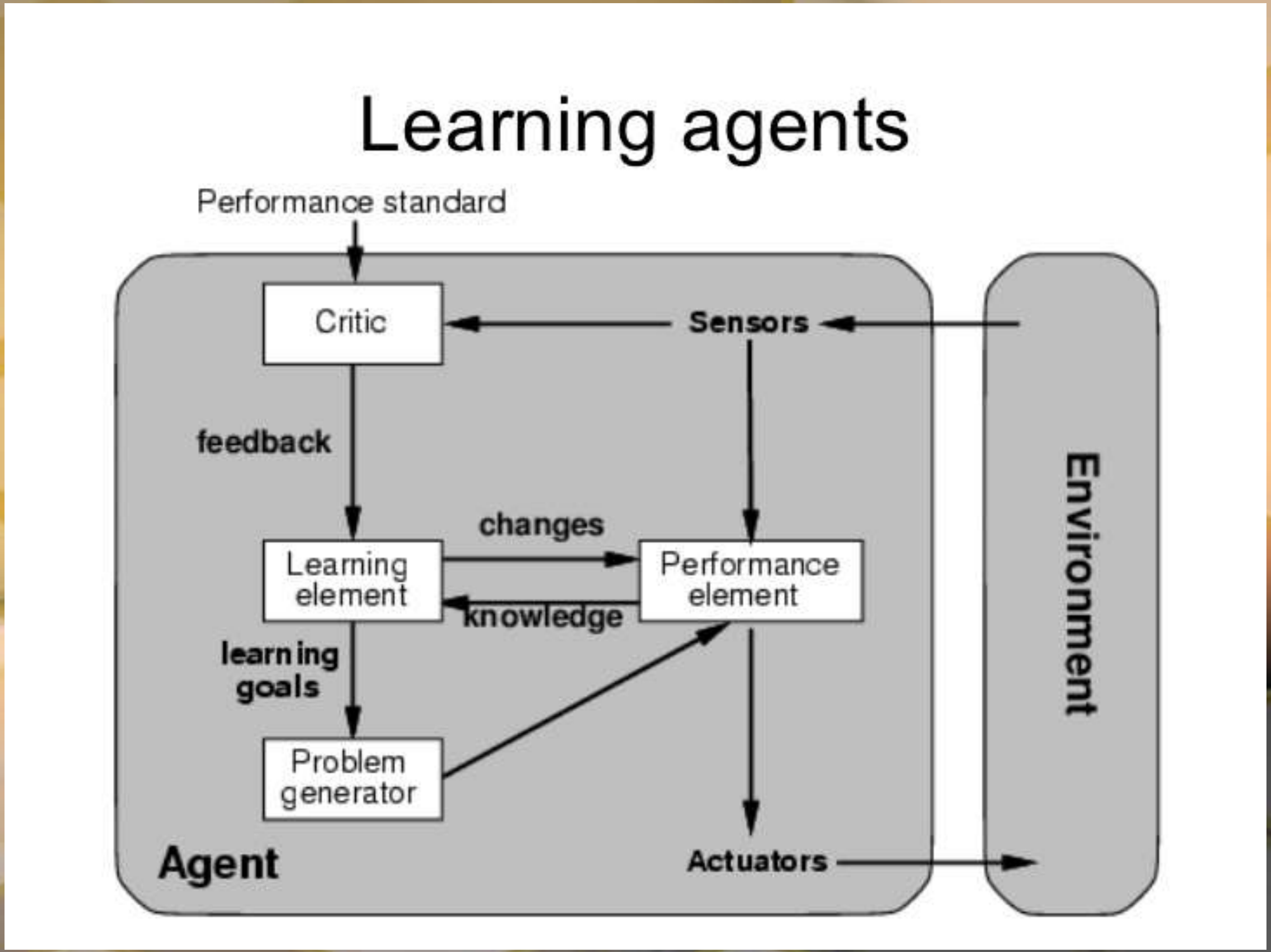
Disadvantages of Production Systems

- There are three main shortcomings:
    - Opaque relations between rules.
    - Ineffective search strategy.
    - Inability to learn.

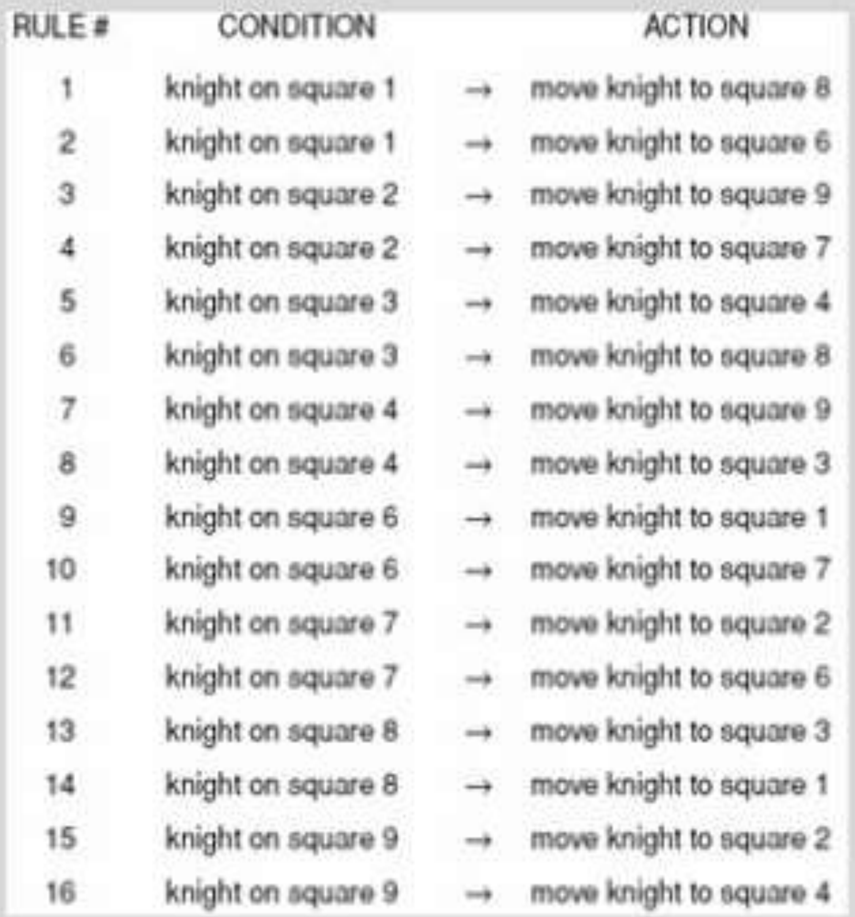
- Opaque relations between rules:
  - Although the individual production rules tend to be relatively simple and self-documented, their logical interactions within the large set of rules may be opaque.
  - productions systems make it difficult to observe how individual rules serve the overall strategy.
- Ineffective search strategy:
  - The rule interpreter applies an exhaustive search through all the production rules during each cycle.
  - Production systems with a large set of rules (over 100 rules) can be slow, and thus large rule-based systems can be unsuitable for real-time applications.

- Inability to learn:
  - In general, rule-based systems do not have an ability to learn from the experience.
  - Unlike a human, who knows when to 'break the rules', a production system cannot automatically modify its knowledge base, or adjust existing rules or add new ones.
  - The knowledge engineer is still responsible for revising and maintaining the system.

Where does learning come in?

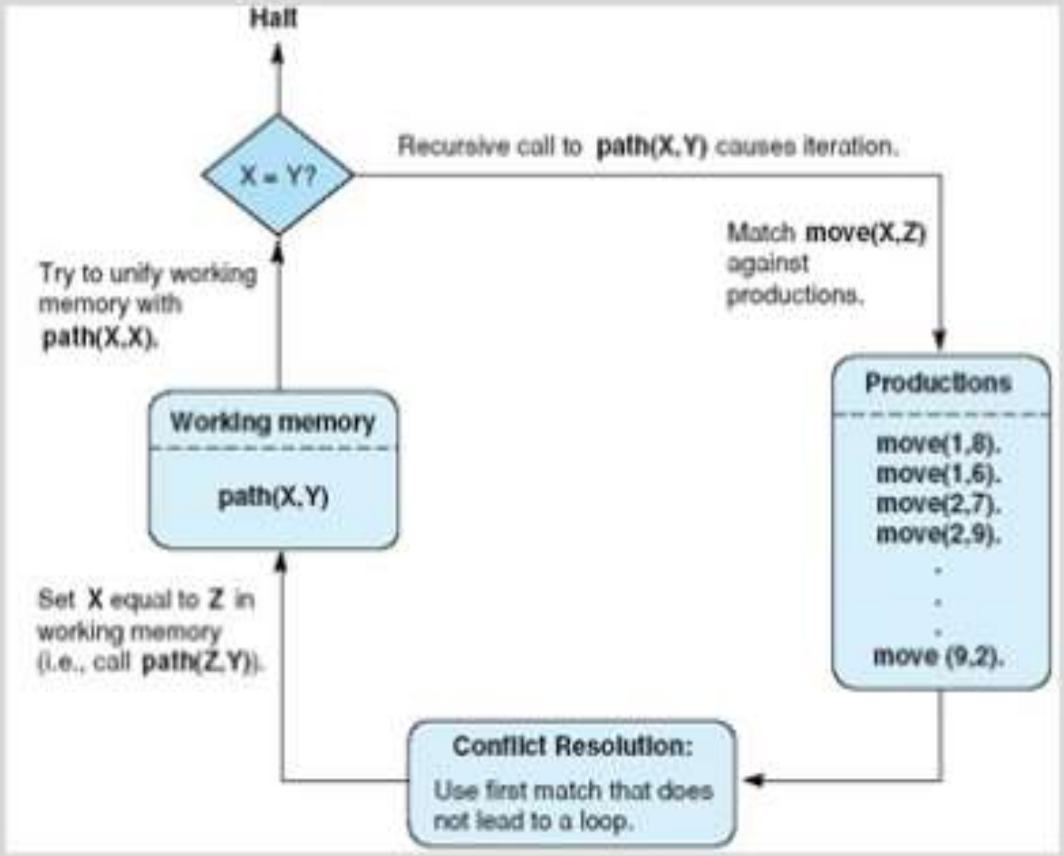


Production rules for the 3 x 3 knight problem.



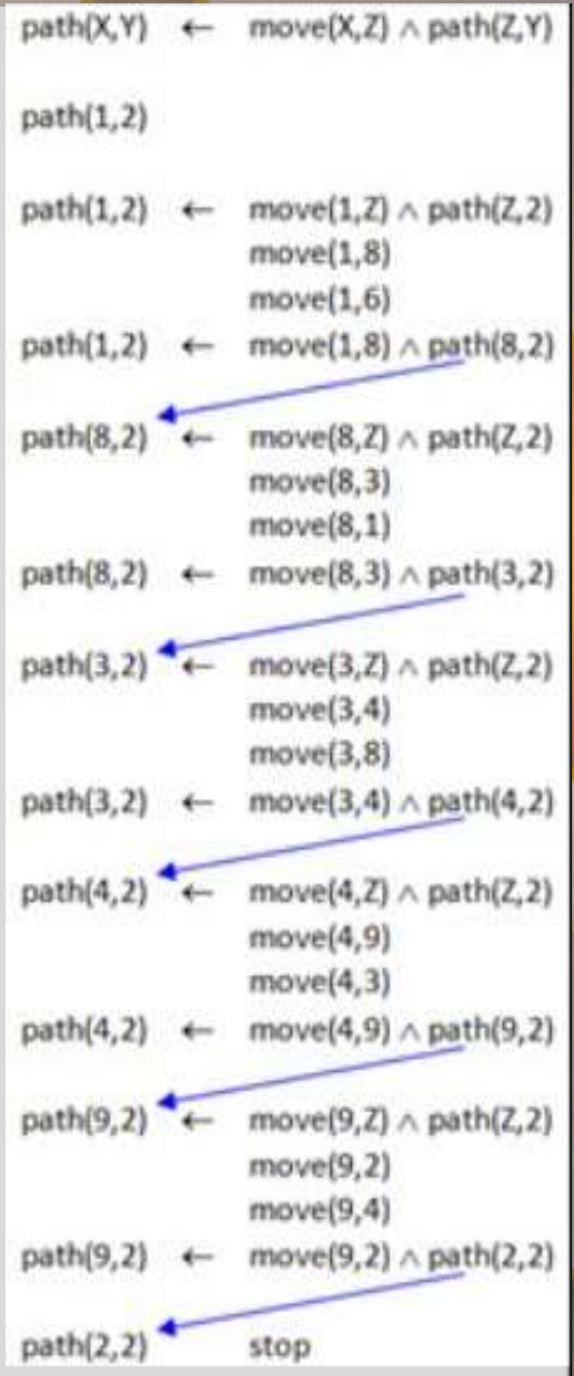
RULE #	CONDITION		ACTION
1	knight on square 1	→	move knight to square 8
2	knight on square 1	→	move knight to square 6
3	knight on square 2	→	move knight to square 9
4	knight on square 2	→	move knight to square 7
5	knight on square 3	→	move knight to square 4
6	knight on square 3	→	move knight to square 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 system...


Halt
X = Y?
Recursive call to path(X,Y) causes iteration.
Match move(X,Z) against productions.
Try to unify working memory with path(X,X).
Working memory
path(X,Y)
Productions
move(1,8).
move(1,6).
move(2,7).
move(2,9).
.
.
.
move (9,2).
Set X equal to Z in working memory (i.e., call path(Z,Y)).
Conflict Resolution:
Use first match that does not lead to a loop.

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

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



path(X,Y) ← move(X,Z) ∧ path(Z,Y)

path(1,2)

path(1,2) ← move(1,Z) ∧ path(Z,2)
move(1,8)
move(1,6)

path(1,2) ← move(1,8) ∧ path(8,2)

path(8,2) ← move(8,Z) ∧ path(Z,2)
move(8,3)
move(8,1)

path(8,2) ← move(8,3) ∧ path(3,2)

path(3,2) ← move(3,Z) ∧ path(Z,2)
move(3,4)
move(3,8)

path(3,2) ← move(3,4) ∧ path(4,2)

path(4,2) ← move(4,Z) ∧ path(Z,2)
move(4,9)
move(4,3)

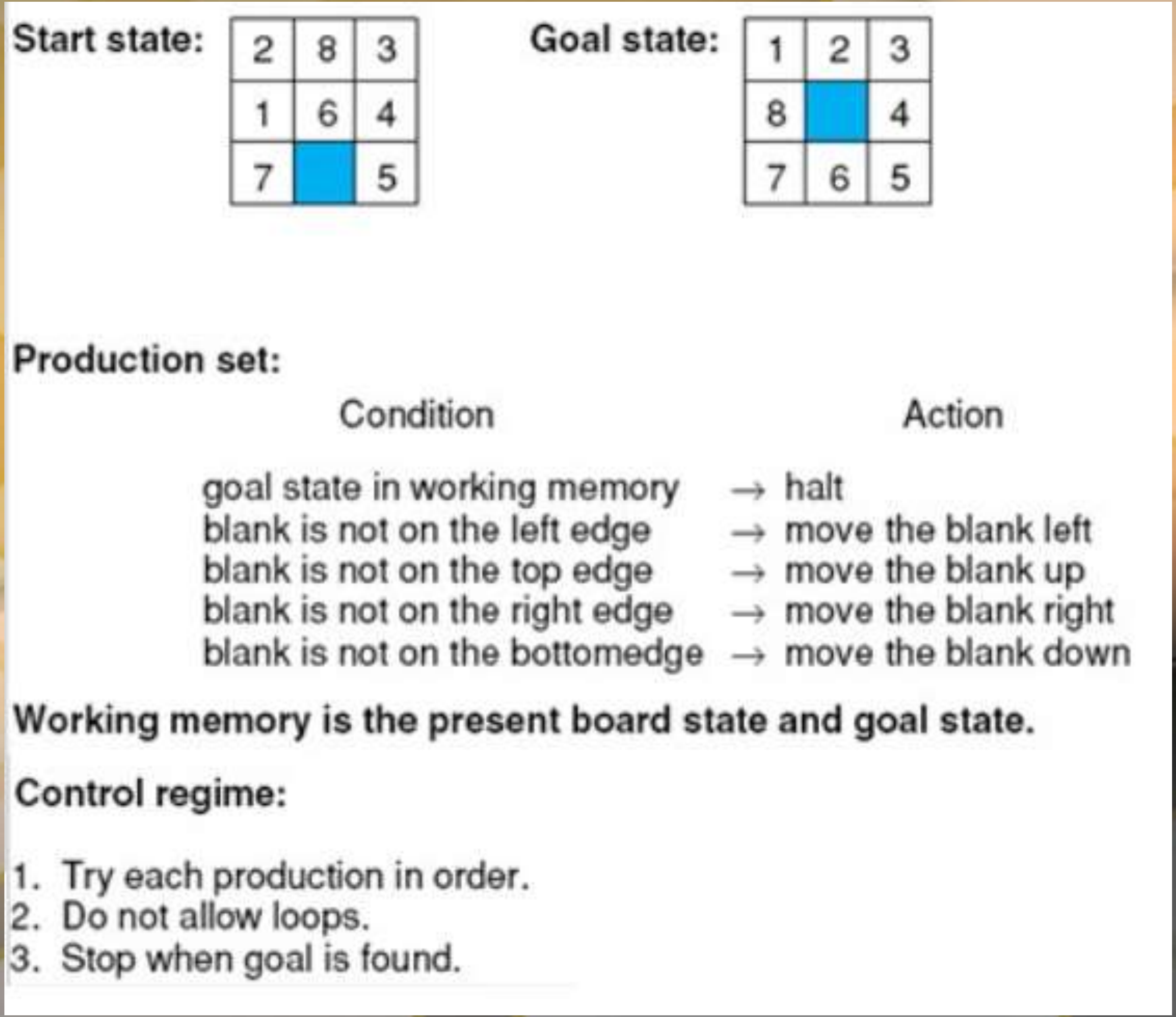
path(4,2) ← move(4,9) ∧ path(9,2)

path(9,2) ← move(9,Z) ∧ path(Z,2)
move(9,2)
move(9,4)

path(9,2) ← move(9,2) ∧ path(2,2)

path(2,2) stop

The 8-puzzle as a production System



Start state:

2	8	3
1	6	4
7	🟦	5

Goal state:

1	2	3
8	🟦	4
7	6	5

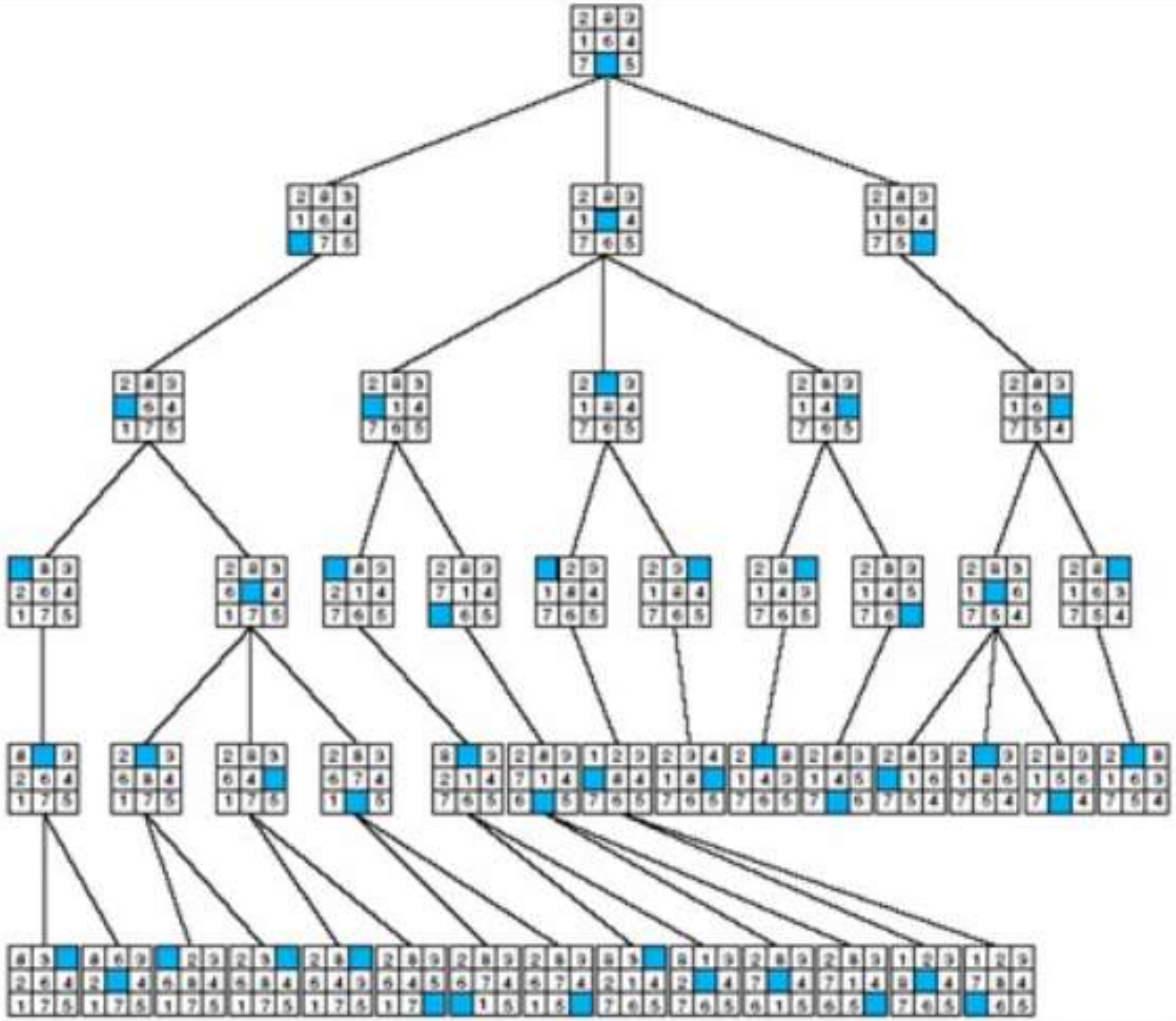
Production set:

Condition		Action
goal state in working memory	→	halt
blank is not on the left edge	→	move the blank left
blank is not on the top edge	→	move the blank up
blank is not on the right edge	→	move the blank right
blank is not on the bottomedge	→	move the blank down

Working memory is the present board state and goal state.

Control regime:

1. Try each production in order.
2. Do not allow loops.
3. Stop when goal is found.



Control of search in Production Systems

Conflict Resolution

- The set of rules that have their conditions satisfied by working memory elements forms a conflict set.
- A conflict is said to occur if more than one rules are found to fire in one cycle.
- In such a scenario, the control structure must determine which rule to fire from this conflict set of active rules.
- This process of selection is called as conflict resolution so, conflict resolution normally selects a single rule to fire.
- There are three conflict resolution mechanism:
  - Refraction
  - Recency and
  - specificity

- Refraction:
  - Refraction specifies that once a rule has fired, it may not fire again until the working memory elements that match its conditions have been modified.
  - This discourage looping.
- Recency:
  - The recency strategy prefers rules whose conditions match with the patterns most recently added to working memory.
  - This focuses the search on a single line of reasoning.
- Specificity:
  - This strategy assumes that a more specific problem-solving rule is preferable to a general rule.
  - A rule is more specific than another if it has more conditions,
  - This implies that it will match fewer potential working memory patterns.

Production Systems and inference systems

- In a production system, the domain knowledge is represented by a set of IF-THEN production rules and data is represented by a set of facts about the current situation.
- The rule interpreter compares each rule stored in the rule base with facts contained in the working memory.
- When the IF (condition) part of the rule matches a fact, the rule is fired and its THEN (action) part is executed.
- The fired rule may change the set of facts in the working memory by adding a new fact.
- The matching of the rule's IF parts to the facts produces inference chains. The inference chain indicates how a production system applies the rules to reach a conclusion.

- There are two principal ways in which rules are executed.
  - Forward chaining: Forward chaining starts from the start state(Known facts or data) and proceeds towards the goal state (conclusion) and
  - Backward chaining: On the other hand backward chaining starts from the goal state and proceeds towards start state.

- Forward chaining:
  - Sometimes called the data driven approach.
  - Working from the facts(data) to conclusion(goal).
  - To chain forward, match data in working memory against 'conditions' of rules in the rule-base. When one of them fires, this is liable to produce more data. So the cycle continues.
  - Forward chaining is the best choice if:
    - All the facts are provided with the problem statement; or:
    - There are many possible goals, and a smaller number of patterns of data; or There isn't any sensible way to guess what the goal is at the beginning of the consultation.

- Forward Chaining example:
  - Initial content of working memory(Known fact) is Start
  - Production Rule set:

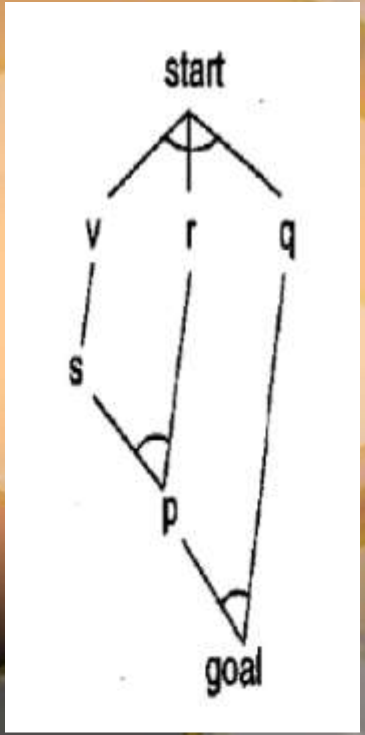
$$\begin{array}{lll} 1. & p \wedge q & \rightarrow goal \\ 2. & r \wedge s & \rightarrow p \\ 3. & w \wedge r & \rightarrow p \\ 4. & t \wedge u & \rightarrow q \\ 5. & v & \rightarrow s \\ 6. & start & \rightarrow v \wedge r \wedge q \end{array}$$

  - Problem : Prove If p and q true Then goal is true

    i.e., prove $p \wedge q \rightarrow goal$

- The following table shows the sequence of rules firing and the stages of working memory in the execution . Execution halts when a goal is reached. A Trace of execution is shown in the following table:

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



- **Conflict resolution strategy used here is:** choose rule that has fired least recently(or not at all.

- Backward chaining:
  - It is also known as goal driven search. A goal driven search begins with a goal and works backward to establish its truth.
  - To implement this in a production system, the goal is placed in working memory and matched against the ACTIONS of the production rules. These ACTIONS are matched just as the CONDTIONs of the productions were matched in the data- driven reasoning.
  - When the ACTION of a rule is matched, the CONDTIONs are added to working memory and become the new sub-goals of the search.
  - The new sub-goals are then matched to the ACTIONs of other production rules. This process continues until a fact given in the problem's initial description is found.
  - The search stops when the CONDITONs of all the productions fired in this backward fashion are found to be true.
  - These CONDITIONs and the chain of rule firings leading to the original goal form a proof of its truth through successive inferences.

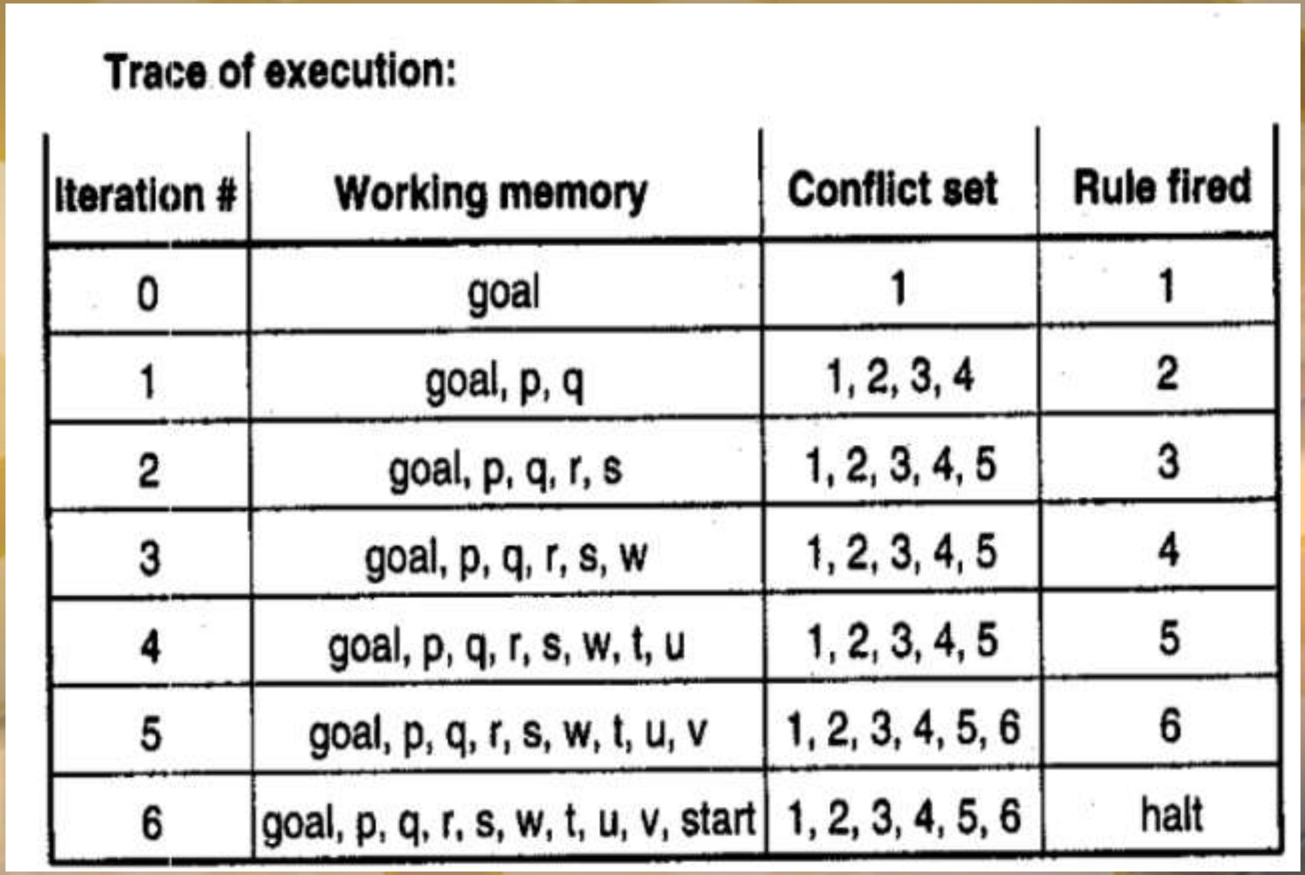
• Backward chaining Example:

- Initial content of working memory is goal
- initially given fact is start
- Production Rule set:

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

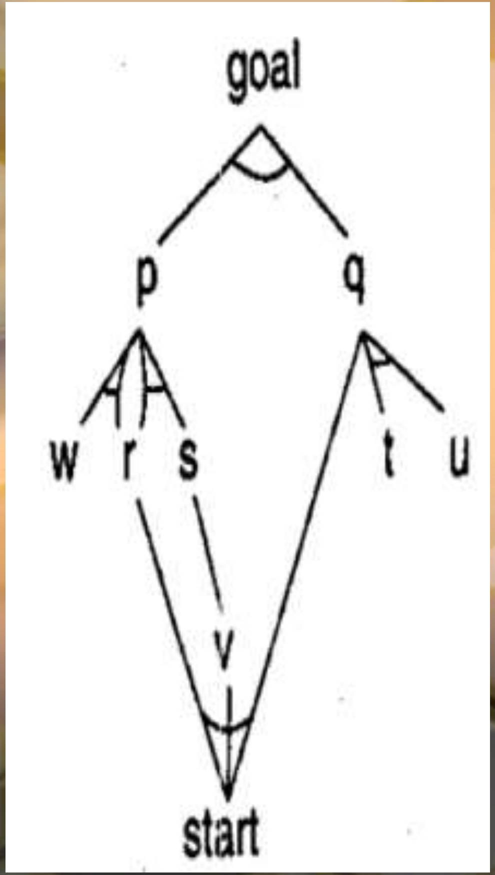
- Problem : Prove If p and q true Then goal is true

  i.e., prove $p \wedge q \rightarrow goal$



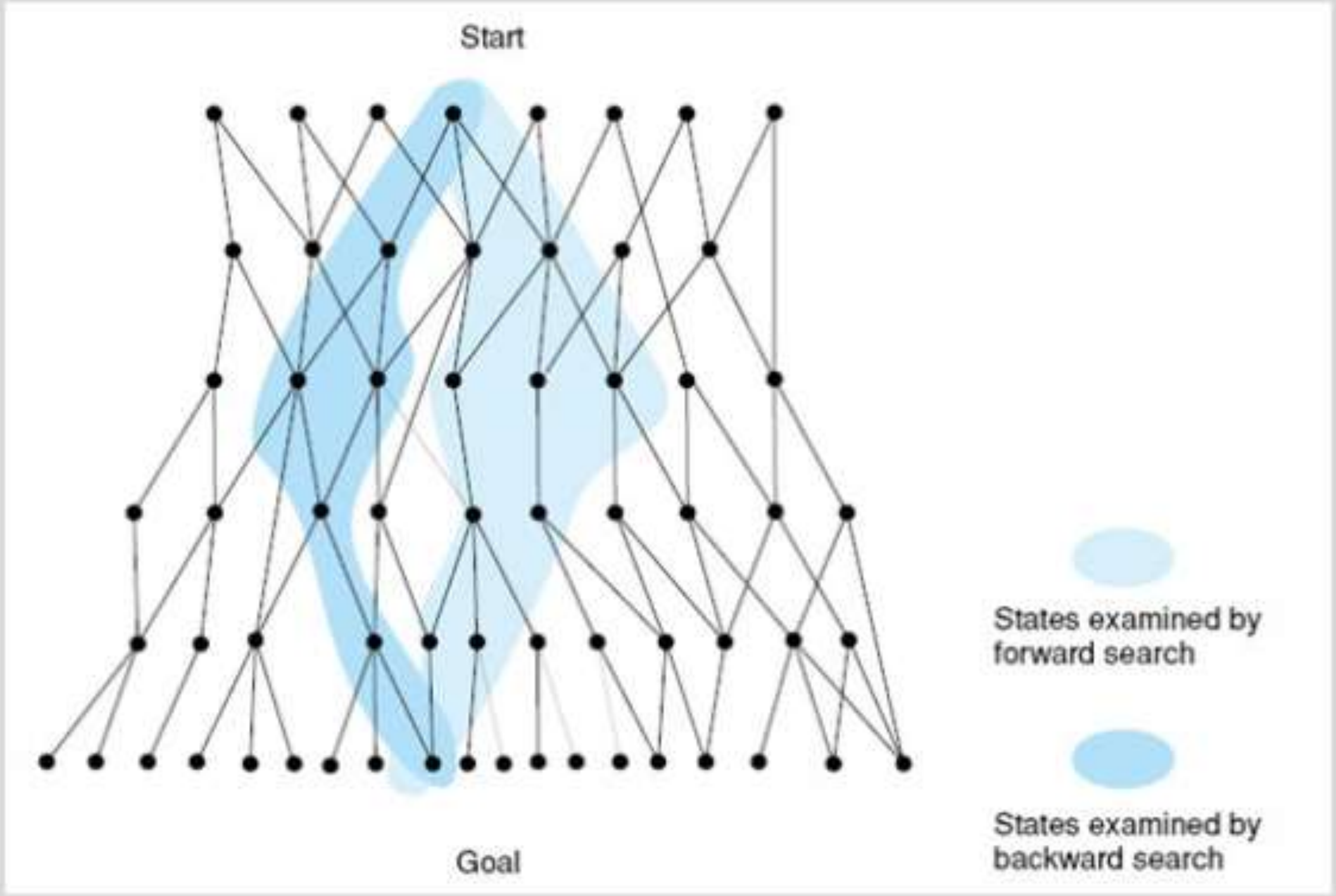
Trace of execution:

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

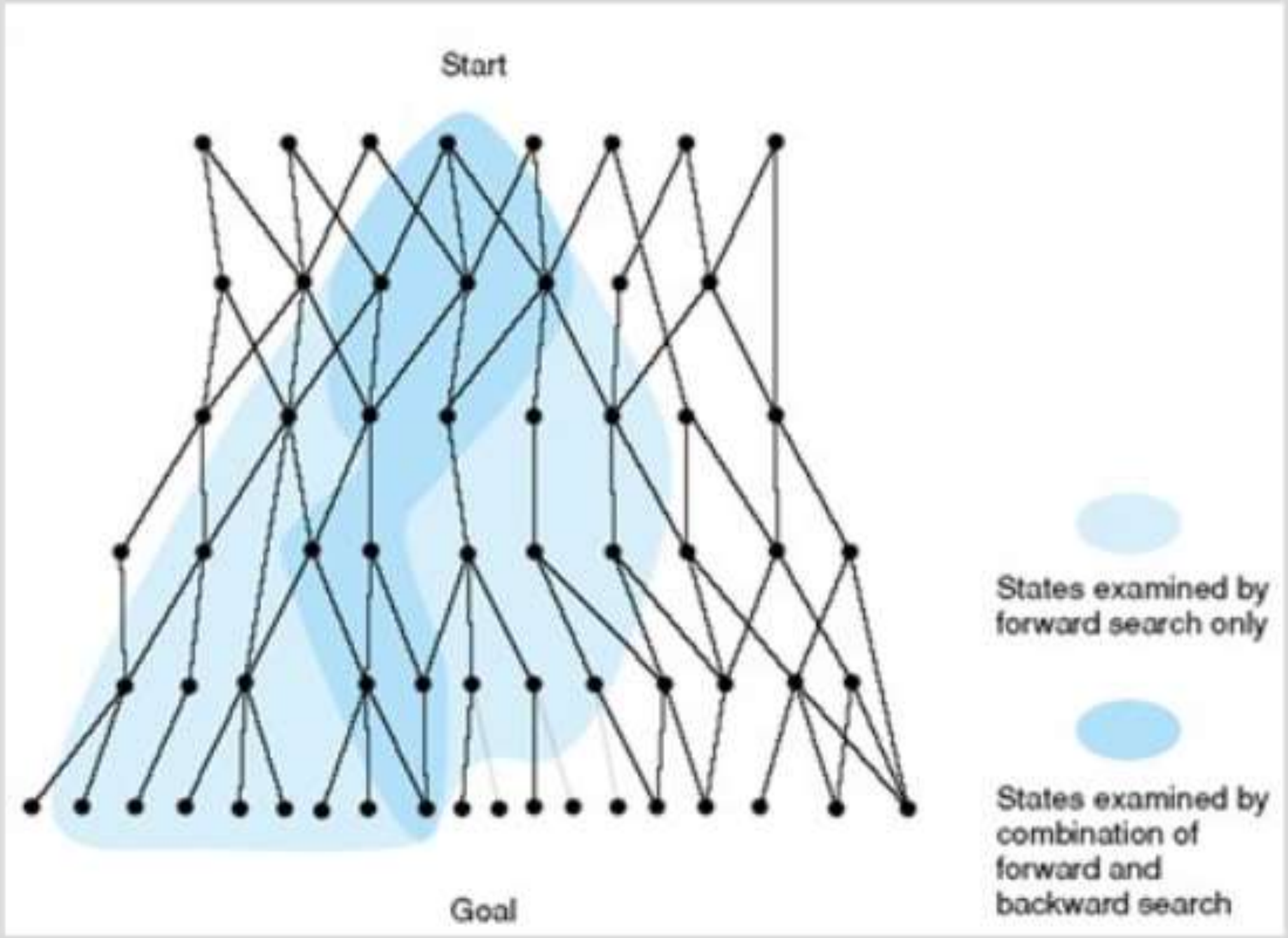


- Backward chaining is the best choice if:
  - The goal is given in the problem statement, or can sensibly be guessed at the beginning of the consultation; or
  - The system has been built so that it sometimes asks for pieces of data rather than expecting all the facts to be presented to it.
  - e.g. "please now do the gram test on the patient's blood, and tell me the result“.
  - This is because (especially in the medical domain) the test may be expensive, or unpleasant, or dangerous for the human participant so one would want to avoid doing such a test unless there was a good reason for it.
- If it is important that the system should seem to behave like a human expert, backward chaining is the best choice.

Bidirectional search missing in both directions, resulting in excessive search

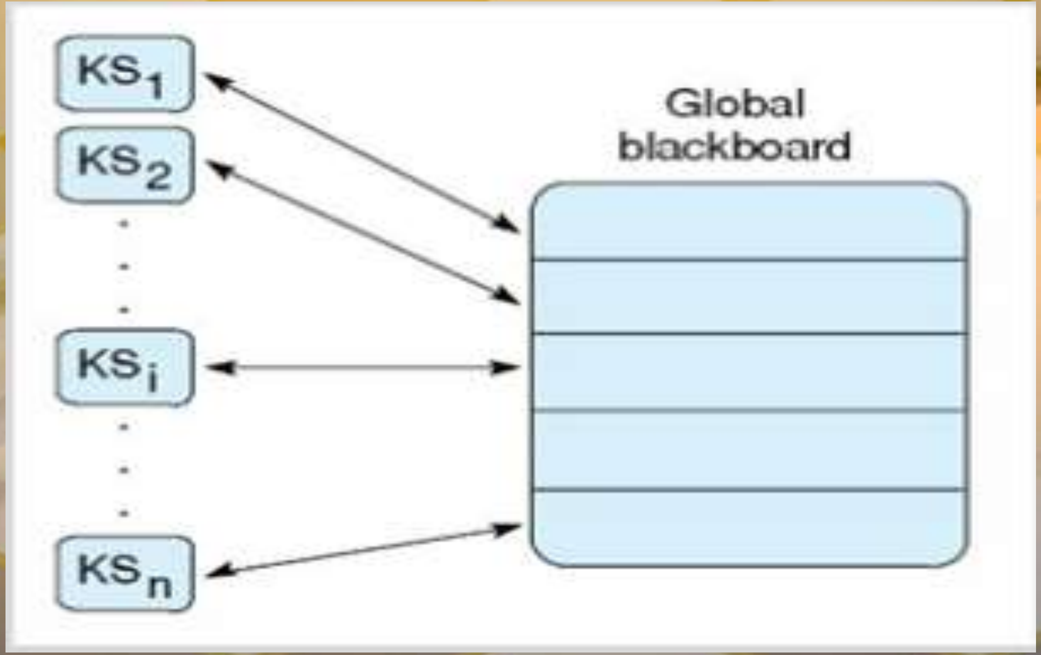

Start
Goal
States examined by forward search
States examined by backward search

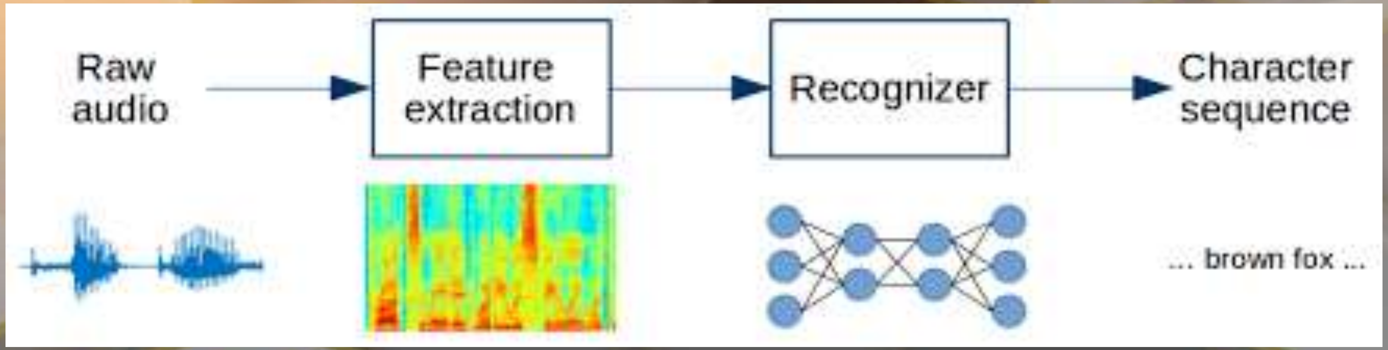
Bidirectional search meeting in the middle, eliminating much of the space examined by unidirectional search.


Start
Goal
States examined by forward search only
States examined by combination of forward and backward search

Blackboard architecture

The blackboard architecture is a model of control that has been applied to problems requiring the coordination of multiple processes or knowledge sources.
A blackboard is a central global database for the communication of asynchronous knowledge sources focusing on related aspects of a particular problem.





introduces higher-level techniques
for implementing search algorithms such as:

- **recursive search** implements depth-first search with backtracking in a more concise,natural fashion than in chapter 3 and form the basis for many of the algorithms; recursive search is augmented through the use of ***unification*** to search the state space generated by predicate calculus assertions
- **pattern-directed search** algorithm is the basis of PROLOG and many of the expert system shells discussed in chapter 7
- **production systems** is a general architecture for pattern-defined problem solving that has been used extensively both to model human solving and to build expert systems and other AI applications
- **blackboard** representation is for another AI problem-solving