CONTROL AND IMPLEMENTATION OF STATE SPACE SEARCH

5.0 Introduction
5.1 Recursion-Based Search
5.2 Pattern-Directed Search
5.3 Production Systems
5.4 The Blackboard Architecture for Problem Solving
5.5 Epilogue and References
5.7 Exercises

George F Luger

ARTIFICIAL INTELLIGENCE
Structure and Strategies for Complex Problem Solving
Introduction

• Chapter 3 and 4 represented problem solving as search through a set of problem situations or states. This state space approach to problem solving allows graph theory to be used as a tool for designing and analyzing programs.

• Chapter 3 defined a general backtracking graph search algorithm as well as algorithms for both depth-first and breadth-first search.

• Chapter 4 presented algorithms for heuristic search.
The following definitions characterize the data and control structures used to implement state space search:

- Representation of a problem solutions as a path from a start state to a goal
- Search to test systematically alternative paths to a goal
- Backtracking or some other mechanism to allow an algorithm to recover from paths that fail to find a goal
- Lists to keep explicit records of states under consideration
  - The open list allows the algorithms to explore untried states if necessary
  - The closed list of visited states allows the algorithm to implement loop detection and avoid repeating fruitless paths
- Implementation of the open list as a stack for depth-first search, a queue for breadth-first search and a priority queue for best-first search
In this chapter 5, it 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.
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)
Example recursive algorithm

function member(item, list)
begin
    if list is empty
        then return FAIL % terminate
    else
        if item = first element of list
            then return SUCCESS % terminate
        else
            begin
                tail := list with its first item removed;
                member(item, tail) % recurse
            end
    end
end
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
                    then add the child to the front of open
            end;
            depthsearch
        end;
end.  % recur
• The previous algorithm does not utilize the full power of recursive
• it is possible to simplify the procedure further by using recursion itself (rather than an explicit open list) to organize states and paths through the state space
• a global closed list is used to detect duplicate states and prevent loops and the open list is implicit in the activation records of the recursive environment
• rather than generating all children of a state and placing them on an open list, the following algorithm produces the child states one at a time and recursively searches the descendants of each child before generating its sibling
• In recursive searching a child state, if some descendant of that state is a goal, the recursive call returns success and the algorithm ignores the siblings
• If the recursive call on the child state fails to find a goal, the next sibling is generated and all of its descendants are searched
• In this fashion, the algorithm searches the entire graph in a depth-first order
function depthsearch (current_state) algorithm

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
end
Pattern-Direct Search

- An algorithm that determines whether a predicate calculus expression is a logical consequence of some set of assertions.
- This suggests a goal directed search with the initial query forming the goal, and the modus ponens defining the transitions between states.
- Given a goal, the algorithm uses unification to select the implication whose conclusions match the goal.
- If goal is \( p(a) \), find implication \( q(X) \rightarrow p(X) \).
- The algorithm treats implications as potential rules for solving the query, called rules.
- After unifying the goal with the conclusion of the implication (or rule) and applying the resulting substitutions throughout the rule.
- The rule premise becomes a new goal \( q(a) \), called subgoal.
- The algorithm then recurs on the subgoal.
- If a subgoal matches a fact in the knowledge base, search terminates.
- The series of inferences that led from the initial goal to the given facts prove the truth of the original goal.
• apply recursive search to a space of logical inferences; the result is a general search procedure for predicate calculus
• a modified version of the recursive search algorithm that uses unification in chapter 2, to determine when two expressions match and modus ponens to generate the children of states
• the current focus variable “current_goal”
• If current_goal matches with a fact, the algorithm returns success, otherwise
• the algorithm attempts to match current_goal with the conclusion of some rules, recursively attempting to solve the premise
• If current_goal does not match any of the given assertions, the algorithm returns fail
function pattern_search (current_goal) algorithm
begin
    if current_goal is a member of closed
        then return FAIL % test for loops
    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 \ldots):
        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 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;
return FAIL % end case
end.
The Knight’s Tour Problem

- A knight can move two squares either horizontally or vertically followed by square in an orthogonal as long as it does not move off the board
- most eight possible moves that the knight may make
- attempts to finds a series of legal moves in which the knight lands on each square of the chessboard exactly once
Figure 5.1: Legal moves of a chess knight.

Figure 5.2: A $3 \times 3$ 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)
• move(1,8) takes the knight from the upper left-hand corner to the middle of the bottom row in 3X3 chessboard
• To determine whether there is a move from 1 to 8, called pattern_search(move(1,8))
• because this goal unifies with move(1,8) in the knowledge base, the result is success with no variable substitutions required
• to find where the knight can move from a particular location such as square 2,
• the goal move(2,X) unifies two different predicates in the knowledge base, with the substitutions of {7/X} and {9/X}
• move(2,3) and move(5,Y) will fail since there is no assertions exist that define a move in the knowledge base
• to devise a general definition for a path of successive moves around the board
• this can be done through the use the predicate calculus implications
• these are added to the knowledge as rules for creating paths of successive moves
• The rules are written as Conclusion $\leftarrow$ Premise
• For example, two-move path could be formulated as

$$\forall X,Y \ [path_2(X,Y) \leftarrow \exists Z [move(X,Z) \land move(Z,Y)]]$$

• It says that for all locations X and Y, a two-move path exists between them if there exists a location Z such that the knight can move from X to Z and then Z to Y
• for a goal path2(1,3), a specific rule defines

$$path_2(1,3) \leftarrow \exists Z [move(1,Z) \land move(Z,3)]$$

• pattern_search then calls itself on this premise
• pattern_search will attempt to solve each subgoal separately
• requires that not only both subgoal succeed but also any variables bindings be consistent across subgoals
• To find all locations that can be reached
• For the goal path\(2(2,Y)\)
• found substitution \(\{6/Y\}\) and \(\{2/Y\}\) with intermediate \(Z\) being 7
• found substitution \(\{2/Y\}\) and \(\{4/Y\}\) with intermediate being 9

• To find a two-move path from a number to itself, from any number to 5 and so on
• one of the advantages of pattern-driven control:
• a variety of queries may be taken as the initial goal
• A three-move path can be defined as

\[ \forall X, Y \ [ path_3(X, Y) \leftarrow \exists Z, W \ [ move(X, Z) \land move(Z, W) \land move(W, Y) ]] \]

• this clause can solve such goals as path3(1,2), path3(1,X0 and path3(X,Y)

• the path moves are the same for a path of any length such as

\[ \forall X, Y \ [ path_3(X, Y) \leftarrow \exists Z \ [ move(X, Z) \land path_2(Z, Y) ]] \]

• the single, general recursive rule:

\[ \forall X, Y \ [ path(X, Y) \leftarrow \exists Z \ [ move(X, Z) \land path(Z, Y) ]] \]
• In “path” rule is incomplete in that it includes no terminating condition

• Any attempt to solve a goal involving the path predicate would fail to halt because each attempt to solve the rule premise would lead to another recursive call on path(Z,Y)

• There is no test in the rule to determine if the desired goal state is ever reached

• The general recursive path is defined as follow:

\[
\forall X \ path(X,X)
\]

\[
\forall X, Y [ path(X,Y) \leftarrow \exists Z [move(X,Z) \land path(Z,Y)] ]
\]
Production Systems

- A model of computation that has proved particularly important in AI both for implementing search algorithms and for modeling human problem solving

- Provide pattern-directed control of a problem-solving process and consists of *a set production rules*, *a working memory*, and *a recognize-act control cycle*
A Production System is defined by

- **The set of production rules**
  - condition-action pair, problem-solving knowledge

- **Working memory**
  - description of the current state of the world, pattern to match

- **The recognize-act cycle**
  - working memory is initialized with beginning problem description
  - a set of pattern, current state, is maintained in working memory
  - patterns in working memory are matched against the conditions
  - produces a subset of the production rules, called conflict set, whose conditions match the patterns in working memory
  - productions in conflict set are said to be enabled
  - select one production and it is fired, i.e. action is performed
  - changes the contents of working memory
  - control cycle repeat until no conditions in working memory are matched with any rule conditions
DEFINITION

PRODUCTION SYSTEM

A production system is defined by:

1. *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.

2. *Working memory* contains a description of the *current state of the world* in a reasoning process.

3. *The recognize–act cycle*. The control structure for a production system is simple: *working memory* is initialized with the beginning problem description. The current state of the problem-solving 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*, whose conditions match the patterns in working memory. The productions in the conflict set are said to be *enabled*. One of the productions in the conflict set is then selected (*conflict resolution*) and the production is *fired*. 
Sorting a String Using Production System

- To sort a string ‘cbaca’ to ‘aabcc’
- production rules:
  
  \[ ba \rightarrow ab \]
  
  \[ ca \rightarrow ac \]
  
  \[ cb \rightarrow bc \]

- 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 rules
Figure 5.3: A production system. Control loops until working memory pattern no longer matches the conditions of any productions.
Figure 5.4: Trace of a simple production system.

Production set:

1. ba → ab
2. ca → ac
3. cb → bc

<table>
<thead>
<tr>
<th>Iteration #</th>
<th>Working memory</th>
<th>Conflict set</th>
<th>Rule fired</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>cbaca</td>
<td>1, 2, 3</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>cabca</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>acbca</td>
<td>2, 3</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>acbac</td>
<td>1, 3</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>acabc</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>5</td>
<td>aacbc</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>6</td>
<td>aabcc</td>
<td>Ø</td>
<td>Halt</td>
</tr>
</tbody>
</table>
Production Systems and Expert Systems

• Production system provides a model for encoding human expertise in the form of rules and designing pattern-driven search algorithms, tasks that are central to the design of the rule-based expert system.

• In expert systems, the production system is not only necessarily assumed to actually model human problem-solving behavior, however, the aspects of production systems that make them useful as a potential model of human problem solving.
Examples of Production Systems

• The 8-Puzzle

• Representation
  – each board configuration with a “state” predicate with nine parameter (locations of eight tiles and the blank)

• Rules
  – implications whose premise performs the required condition check
  – alternatively, arrays or list structures could be used for board states

• A depth bound
  – need to keep track of the length of the current path
  – numbers of possible states of working memory grows exponentially with the depth of the search
Figure 5.5: 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:

<table>
<thead>
<tr>
<th>Condition</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>goal state in working memory</td>
<td>→ halt</td>
</tr>
<tr>
<td>blank is not on the left edge</td>
<td>→ move the blank left</td>
</tr>
<tr>
<td>blank is not on the top edge</td>
<td>→ move the blank up</td>
</tr>
<tr>
<td>blank is not on the right edge</td>
<td>→ move the blank right</td>
</tr>
<tr>
<td>blank is not on the bottom edge</td>
<td>→ move the blank down</td>
</tr>
</tbody>
</table>

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.
Figure 5.6: The 8-puzzle searched by a production system with loop detection and depth bound 5, from Nilsson (1971).
The knight’s Tour Problem on 3X3

• Rules
  – each move represents as a rule whose condition is the location of the knight on a particular square and whose actions moves the knight to another square
  – sixteen productions for all possible moves of the knight

• Working memory
  – contains both the current and goal state

• Control
  – applies rules until the current state equals the goal and then halts

• Conflict resolution
  – fire the first rule that did not cause the search to loop
  – allow backtracking
<table>
<thead>
<tr>
<th>Rule #</th>
<th>Condition</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>knight on square 1</td>
<td>move knight to square 8</td>
</tr>
<tr>
<td>2</td>
<td>knight on square 1</td>
<td>move knight to square 6</td>
</tr>
<tr>
<td>3</td>
<td>knight on square 2</td>
<td>move knight to square 9</td>
</tr>
<tr>
<td>4</td>
<td>knight on square 2</td>
<td>move knight to square 7</td>
</tr>
<tr>
<td>5</td>
<td>knight on square 3</td>
<td>move knight to square 4</td>
</tr>
<tr>
<td>6</td>
<td>knight on square 3</td>
<td>move knight to square 8</td>
</tr>
<tr>
<td>7</td>
<td>knight on square 4</td>
<td>move knight to square 9</td>
</tr>
<tr>
<td>8</td>
<td>knight on square 4</td>
<td>move knight to square 3</td>
</tr>
<tr>
<td>9</td>
<td>knight on square 6</td>
<td>move knight to square 1</td>
</tr>
<tr>
<td>10</td>
<td>knight on square 6</td>
<td>move knight to square 7</td>
</tr>
<tr>
<td>11</td>
<td>knight on square 7</td>
<td>move knight to square 2</td>
</tr>
<tr>
<td>12</td>
<td>knight on square 7</td>
<td>move knight to square 6</td>
</tr>
<tr>
<td>13</td>
<td>knight on square 8</td>
<td>move knight to square 3</td>
</tr>
<tr>
<td>14</td>
<td>knight on square 8</td>
<td>move knight to square 1</td>
</tr>
<tr>
<td>15</td>
<td>knight on square 9</td>
<td>move knight to square 2</td>
</tr>
<tr>
<td>16</td>
<td>knight on square 9</td>
<td>move knight to square 4</td>
</tr>
</tbody>
</table>
**Figure 5.7:** A production system solution to the $3 \times 3$ knight’s tour problem.

<table>
<thead>
<tr>
<th>Iteration #</th>
<th>Working memory</th>
<th>Conflict set (rule #’s)</th>
<th>Fire rule</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Current square</td>
<td>Goal square</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>1, 2</td>
</tr>
<tr>
<td>1</td>
<td>8</td>
<td>2</td>
<td>13, 14</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>2</td>
<td>5, 6</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td>2</td>
<td>7, 8</td>
</tr>
<tr>
<td>4</td>
<td>9</td>
<td>2</td>
<td>15, 16</td>
</tr>
<tr>
<td>5</td>
<td>2</td>
<td>2</td>
<td></td>
</tr>
</tbody>
</table>
Preventing Loop

- pattern_search checked a goal list (closed) of visited states
- the actual conflict resolution strategy was therefore: select the first matching move that leads to an unvisited state
- previously visited are not in global closed list
- so we can use working memory for loop detection
- use assert(X) to enter into working memory
- marked the visited state using been(X) predicate

\[ \forall X \: path(X, X) \]
\[ \forall X, Y \: path(X, Y) \leftarrow \exists Z \: move(X, Z) \land \neg (been(Z)) \land assert(been(Z)) \land path(Z, Y) \]
The Full Knight’s Tour

• Replace the 16 move facts with a set of 8 rules to generate legal knight moves
• those moves correspond to the 8 possible ways a knight can move
• production rule for moving the knight down two squares and right one square

CONDITION : \( \text{current row} \leq 6 \wedge \text{current column} \leq 7 \)

ACTION : \( \text{new row} = \text{current row} + 2 \wedge \text{new column} = \text{current column} + 1 \)

• defined predicate square(R,C), R for row and C for column
• The above rule in predicate calculus

\[ \text{move}(\text{square}(\text{Row}, \text{Column}), \text{square}(\text{Newrow}, \text{Newcolumn})) \leftarrow \]
\[ \text{less\_than\_or\_equals}(\text{Row}, 6) \land \]
\[ \text{equals}(\text{Newrow}, \text{plus}(\text{Row}, 2)) \land \]
\[ \text{less\_than\_or\_equals}(\text{Column}, 7) \land \]
\[ \text{equals}(\text{Newcolumn}, \text{plus}(\text{Column}, 1)) \]

• plus is a function for addition
• less\_than\_or\_equals and equals are logical predicate for comparison
Figure 5.8: The recursive path algorithm: a production system.
Control of Search in Production Systems

Data-Driven or Goal-Driven

Data-Driven

- begins with a problem description, a set of logical axioms, symptoms of an illness
- a rule forms as \( \text{CONDITION} \rightarrow \text{ACTION} \)
- When CONDITION matches some elements of working memory, its ACTION is performed

- infer new knowledge from the data
- applying rules of inference to the current description of the world and adding the results to that problem description
- closed fit to the production system model of computation
Example of Data-Driven search

• A set of data-driven search on a set of productions expressed as propositional calculus implications
• The conflict resolution strategy is a simple one of choosing the enabled rule that has fired least recently
• in the case of tie, the first rule is chosen
• Execution halts when a goal is reached
**Figure 5.9**: Data-driven search in a production system.

**Production set:**
1. \( p \land q \rightarrow \text{goal} \)
2. \( r \land s \rightarrow p \)
3. \( w \land r \rightarrow q \)
4. \( t \land u \rightarrow q \)
5. \( v \rightarrow s \)
6. start \( \rightarrow v \land r \land q \)

**Trace of execution:**

<table>
<thead>
<tr>
<th>Iteration #</th>
<th>Working memory</th>
<th>Conflict set</th>
<th>Rule fired</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>start</td>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>1</td>
<td>start, v, r, q</td>
<td>6, 5</td>
<td>5</td>
</tr>
<tr>
<td>2</td>
<td>start, v, r, q, s</td>
<td>6, 5, 2</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>start, v, r, q, s, p</td>
<td>6, 5, 2, 1</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>start, v, r, q, s, p, goal</td>
<td>6, 5, 2, 1</td>
<td>halt</td>
</tr>
</tbody>
</table>

**Space searched by execution:**

```
start

v

r

q

s

p

goal
```
Goal-Driven

- begins with a goal
- works backward to the facts of the problem to satisfy that goal
- the goal is placed in working memory and
- matched against the ACTIONs of the production rules
- all production rules whose conclusion ACTIONs match the goal form the conflict set

- when the ACTION of a rule is matched, the CONDITIONs are added to working memory and become the new subgoals (states) of the search
- the new states are then matched to the ACTIONs of other production rules
- The process continues until a fact is found, usually in the problem’s initial description
Figure 5.10: Goal-driven search in a production system.

**Production set:**
1. $p \land q \rightarrow \text{goal}$
2. $r \land s \rightarrow p$
3. $w \land r \rightarrow p$
4. $t \land u \rightarrow q$
5. $v \rightarrow s$
6. start $\rightarrow v \land r \land q$

**Trace of execution:**

<table>
<thead>
<tr>
<th>Iteration #</th>
<th>Working memory</th>
<th>Conflict set</th>
<th>Rule fired</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>goal</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>goal, $p, q$</td>
<td>1, 2, 3, 4, 5</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>goal, $p, q, r, s$</td>
<td>1, 2, 3, 4, 5</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>goal, $p, q, r, s, w$</td>
<td>1, 2, 3, 4, 5</td>
<td>4</td>
</tr>
<tr>
<td>4</td>
<td>goal, $p, q, r, s, w, t, u$</td>
<td>1, 2, 3, 4, 5</td>
<td>5</td>
</tr>
<tr>
<td>5</td>
<td>goal, $p, q, r, s, w, t, u, v$</td>
<td>1, 2, 3, 4, 5, 6</td>
<td>6</td>
</tr>
<tr>
<td>6</td>
<td>goal, $p, q, r, s, w, t, u, v, \text{start}$</td>
<td>1, 2, 3, 4, 5, 6</td>
<td>halt</td>
</tr>
</tbody>
</table>

**Space searched by execution:**

(Space diagram showing the search process with goal as the root and various steps leading to the goal with rules applied in different iterations.)

Direction of search
Figure 5.11: Bidirectional search missing in both directions, resulting in excessive search.
Figure 5.12: Bidirectional search meeting in the middle, eliminating much of the space examined by unidirectional search.
Major advantages of production systems for artificial intelligence

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
Advantages of Production Systems for AI

- Separation of Knowledge and Control
- A Natural Mapping onto State Space Search
- Modularity of Production Rules
- Pattern-Direct Control
- Opportunities for Heuristic Control of Search
- Tracing and Explanation
- Language Independence
- A Plausible Model of Human Problem Solving
Blackboard Architecture for Problem Solving

Blackboards

- Extend productions system by allowing us to organize working memory into separate modules, each of which corresponds to a different subset of production rules
- Integrate these separate sets of production rules and coordinate the actions of these multiple problem solvers within a single structure, the blackboard
- a model of control that has been applied to problems requiring the coordination of multiple process or knowledge sources
- a central global data base for the communication of independent asynchronous knowledge sources focusing on related aspects of a particular problem
- HEARSAY-II research 1976, 1980 speech understanding program used the blackboard approach
Blackboard Architecture for Problem Solving

Blackboards

- Extend productions system by allowing us to organize working memory into separate modules, each of which corresponds to a different subset of production rules
- Integrate these separate sets of production rules and coordinate the actions of these multiple problem solvers within a single structure, the blackboard
- A model of control that has been applied to problems requiring the coordination of multiple process or knowledge sources
- A central global data base for the communication of independent asynchronous knowledge sources focusing on related aspects of a particular problem
- HEARSAY-II research 1976, 1980 speech understanding program used the blackboard approach
• Each knowledge source $KS_i$ gets its data from the blackboard, processes the data, and returns to the blackboard to be used by the other knowledge sources.

• Each $KS_i$ is independent in that it is a separate process operating according to its own specifications and when a multiprocessing or multiprocessor system is used, it is independent of the other processing in the problem.

• It is an asynchronous system in that $KS_i$ begins its operation whenever it finds appropriate input data posted on the blackboard.

• When it finished its processing it posts its results and awaits new input data.
Blackboard Architecture

\[ KS_1 \quad KS_2 \quad KS_i \quad KS_n \quad \text{Global Blackboard} \]

- **KS\(_1\)**: The waveform of the acoustic signal.
- **KS\(_2\)**: The phonemes or possible sound segments of the acoustic signal.
- **KS\(_3\)**: The syllables that the phonemes could produce.
- **KS\(_4\)**: The possible words as analyzed by one KS.
- **KS\(_5\)**: The possible words as analyzed by a second KS (usually considering words from different parts of the data).
- **KS\(_6\)**: A KS to try to generate possible word sequences.
- **KS\(_7\)**: A KS that puts word sequences into possible phrases.