Building Control Algorithms For State Space Search.

5.0 Introduction
5.1 Recursion-Based Search
5.2 Production Systems

5.3 The Blackboard Architecture for Problem Solving
5.4 Epilogue and References
5.5 Exercises
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 % build stack
            end;
            depthsearch % recur
        end;
    depthsearch
end.
function depthsearch (current_state);
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
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 database unifying facts or rules do
  begin
    case
      current_goal unifies with a fact:
        return SUCCESS;
      current_goal is a conjunction (p ∧ ...):
        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 → 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;

return FAIL
end.
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. 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.

2. Working memory contains a description of the current state of the world 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 element 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.

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. 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. The process terminates when the contents of working memory do not match any rule conditions.

Conflict resolution 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 state of the world, or may involve complex rule selection heuristics. This is an important way in which a production system allows the addition of heuristic control to a search algorithm.

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. Many practical implementations of production systems allow backtracking to a previous state of working memory in such situations.

A schematic drawing of a production system is presented in Figure 6.1.
Fig 6.1 A production system. Control loops until working memory pattern no longer matches the conditions of any productions.
Fig 6.2  Trace of a simple production system.

<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 set:
1. ba → ab
2. ca → ac
3. cb → bc
Fig 6.3  The 8-puzzle as a production system.

<table>
<thead>
<tr>
<th>Start state:</th>
<th>Goal state:</th>
</tr>
</thead>
</table>
| \[
| 2 & 8 & 3  \\
| 1 & 6 & 4  \\
| 7 & 5      |
|\]           | \[
| 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.
Fig 6.4  The 8-puzzle searched by a production system with loop detection and depth-bound, from Nilsson (1971).
Fig 6.5  Legal moves of a chess knight.
Fig 6.6 a 3 x 3 chessboard with move rules for the simplified knight tour problem.
Table 6.1 Production rules for the 3 x 3 knight problem.

<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>
Fig6.7  A production system solution to the 3 x 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>Halt</td>
</tr>
</tbody>
</table>
Fig 6.8  The recursive path algorithm as production system.

Recursive call to path(X,Y) causes iteration.

X = Y?

Try to unify working memory with path(X,X).

Working memory

path(X,Y)

Set X equal to Z in working memory (i.e., call path(Z,Y)).

Match move(X,Z) against productions.

Productions

move(1,8).
move(1,6).
move(2,7).
move(2,9).

Conflict Resolution:
Use first match that does not lead to a loop.
Fig 6.9  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:

Direction of search
Fig 6.10 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</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, start</td>
<td>1, 2, 3, 4, 5, 6</td>
<td>halt</td>
</tr>
</tbody>
</table>

Space searched by execution:

```
     goal
    /   \
  p     q
 /     /\    \
 w   /  s \t   u
 /\     \   \
 v   /    \    \
 \    /      \   \
 start
```
Fig 6.11 Bidirectional search missing in both directions, resulting in excessive search.
Fig 6.12 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
Fig 6.13 Blackboard architecture