5 Control and Implementation of State Space Search

Chapter Objectives

- Compare the recursive and iterative implementations of the depth-first search algorithm
- Learn about pattern-directed search as a basis for production systems
- Learn the basics of production systems
- The agent model: Has a problem, searches for a solution, has different ways to model the search

Summary of previous chapters

- **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**

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.
```

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

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
```


Pattern-directed search

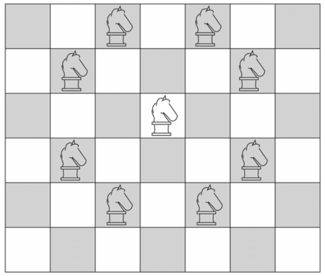
- use modus ponens on rules such as $q(X) \rightarrow p(X)$
- if p(a) is the original goal, after unification on the above rule, the new subgoal is q(a)


```
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 ∧ ...):
               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 case
      end;
      return FAIL
   end.
```

A chess knight's tour problem

Legal moves of a knight



Move rules

move(1,8) move(1,6) move(2,9) move(2,7) move(3,4) move(3,8) move(4,9) move(4,3)

move(6,1) move(6,7) move(7,2) move(7,6) move(8,3) move(8,1) move(9,2) move(9,4)

1	2	3
4	5	6
7	8	9

Examples

- **Is there a move from 1 to 8?**
  Pattern_search(move(1,8)) success
- **Where can the knight move from 2?**
  Pattern_search(move(2,X)) {7/X}, {9/X}
- **Can the knight move from 2 to 3?**
  Pattern_search(move(2,3)) fail
- **Where can the knight move from 5?**
  Pattern_search(move(5,X)) fail

2 step moves

- **$\forall X,Y$ [path2(X,Y) $\leftarrow \exists Z$ [move(X,Z) $\wedge$ move(Z,Y)]]**
- **path2(1,3)?**
- **path2(2,Y)?**

3 step moves

- **$\forall X,Y$ [path3(X,Y) $\leftarrow \exists Z,W$ [move(X,Z) $\wedge$ move(Z,W) $\wedge$ move(W,Y)]]**
- **path3(1,2)?**
- **path3(1,X)?**
- **path3(X,Y)?**

General recursive rules

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

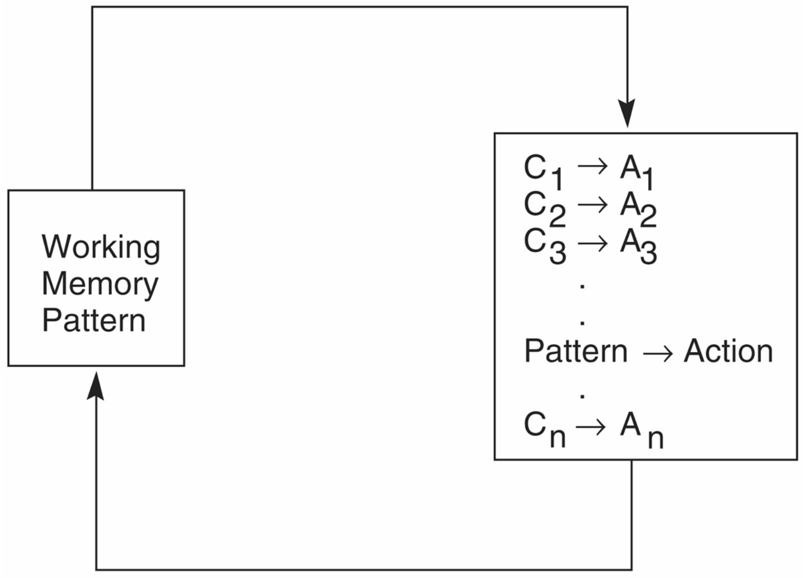
Generalized pattern_search

- **if the current goal is negated**
  call pattern_search with the goal and return success if the call returns failure
- **if the current goal is a conjunction**
  call pattern_search for all the conjuncts
- **if the current goal is a disjunction**
  call pattern_search for all the disjuncts until one returns success

A *production system* is defined by:

- **A set of *production rules* (aka *productions*): condition-action pairs.**
- ***Working memory*: the current state of the world**
- **The *recognize-act cycle*: the control structure for a production system**
  - **Initialize working memory**
  - **Match patterns to get the *conflict set* (*enabled rules*)**
  - **Select a rule from the conflict set (*conflict resolution*)**
  - ***Fire* the rule**

A production system



Trace of a simple production system

Production set:

1. ba → ab
2. ca → ac
3. cb → 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	Ø	Halt

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:

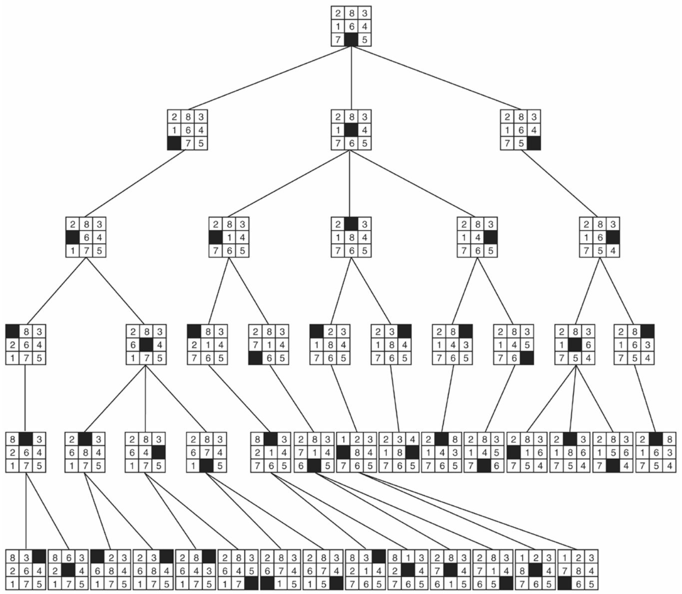
Condition		Action
goal state in working memory	→	halt
blank is not on the left edge	→	move the blank left
blank is not on the top edge	→	move the blank up
blank is not on the right edge	→	move the blank right
blank is not on the bottomedge	→	move the blank down

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.

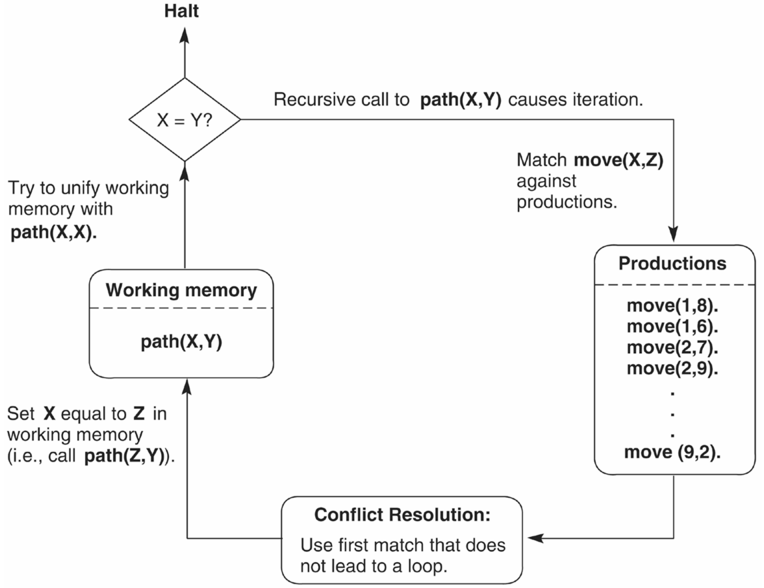
Production system search with loop detection & depth bound 5 (Nilsson, 1971)



A production system solution to the 3 × 3 knight's tour problem

Iteration #	Working memory: Current square	Working memory: Goal square	Conflict set (rule #'s)	Fire rule
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

The recursive path algorithm: a production system



Data-driven search in a production system

Production set:

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

Goal-driven search in a production system

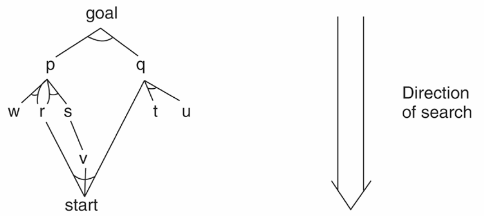
Production set:

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 $\rightarrow v \wedge r \wedge q$

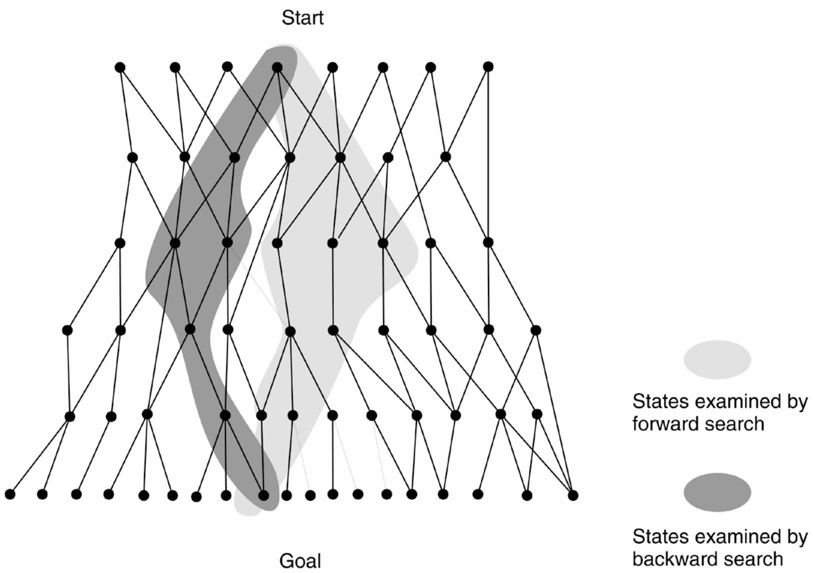
Trace of execution:

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

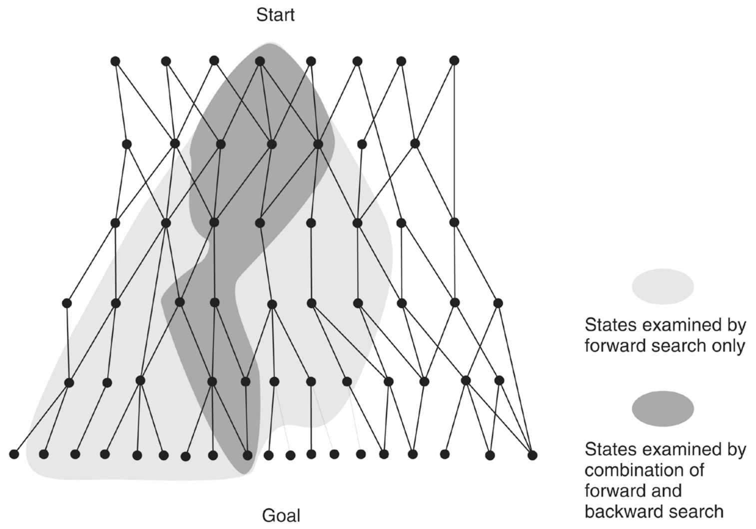
Space searched by execution:



Bidirectional search misses in both directions: excessive search



Bidirectional search meets in the middle



Advantages of production systems

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

Comparing search models

Given a start state and a goal state

- **State space search keeps the "current state" in a "node". Children of a node are all the possible ways an operator can be applied to a node**

- **Pattern-directed search keeps the all the states (start, goal, and current) as logic expressions. Children of a node are all the possible ways of using modus ponens.**

- **Production systems keep the "current state" in "working memory." Children of the current state are the results of all applicable productions.**

Variations on a search theme

- **Bidirectional search: Start from both ends, check for intersection (Sec. 5.3.3).**

- **Depth-first with iterative deepening: implement depth first search using a *depth-bound*. Iteratively increase this bound (Sec. 3.2.4).**

- **Beam search: keep only the "best" states in OPEN in an attempt to control the space requirements (Sec. 4.4).**

- **Branch and bound search: Generate paths one at a time, use the best cost as a "bound" on future paths, i.e., do not pursue a path if its cost exceeds the best cost so far (Sec. 3.1.2).**