Topics discussed

the history and applications of AI,
logical reasoning
state space searching
heuristic searching
game playing
expert systems
problem solving methods
Reasoning with uncertainty

Introduction to AI

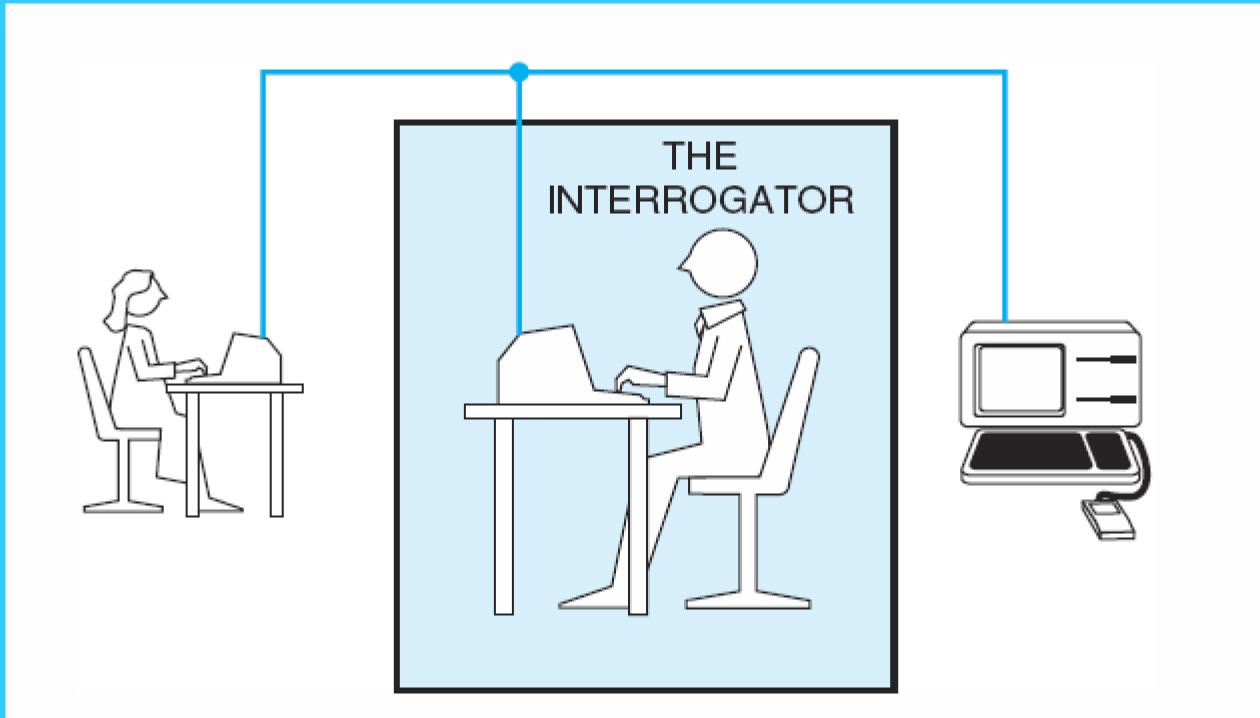
Intelligence means:

a system can adapt itself to novel situations, has the capacity to reason, to understand the relationships between facts, to discover meanings, to recognize truth, and to learn.

Artificial intelligence is:

the science of making machines do things that would require intelligence if done by men.

Fig 1.1 The Turing test.



Two most fundamental problems of AI

- knowledge representation
- problem-solving technique, such as search

Important Research and Application Areas

1.2.1 Game Playing

1.2.2 Automated Reasoning and Theorem Proving

1.2.3 Expert Systems

1.2.4 Natural Language Understanding and Semantic Modelling

1.2.5 Modelling Human Performance

1.2.6 Planning and Robotics

1.2.7 Languages and Environments for AI

1.2.8 Machine Learning

1.2.9 Alternative Representations: Neural Nets and Genetic Algorithms

A representation scheme should:

a) Be adequate to express all of the necessary information.
b) Support efficient execution of the resulting code.
c) Provide a natural scheme for expressing the required knowledge.

For propositional expressions **P**, **Q** and **R**:

$\neg(\neg P) \equiv P$

$(P \vee Q) \equiv (\neg P \rightarrow Q)$

The contrapositive law: $(P \rightarrow Q) \equiv (\neg Q \rightarrow \neg P)$

De Morgan's law: $\neg(P \vee Q) \equiv (\neg P \vee \neg Q)$ and $\neg(P \wedge Q) \equiv (\neg P \vee \neg Q)$

The commutative laws: $(P \wedge Q) \equiv (Q \wedge P)$ and $(P \vee Q) \equiv (Q \vee P)$

The associative law: $((P \wedge Q) \wedge R) \equiv (P \wedge (Q \wedge R))$

The associative law: $((P \vee Q) \vee R) \equiv (P \vee (Q \vee R))$

The distributive law: $P \vee (Q \wedge R) \equiv (P \vee Q) \wedge (P \vee R)$

The distributive law: $P \wedge (Q \vee R) \equiv (P \wedge Q) \vee (P \wedge R)$

DEFINITION

PREDICATE CALCULUS SYMBOLS

The alphabet that makes up the symbols of the predicate calculus consists of:

1. The set of letters, both upper- and lowercase, of the English alphabet.
2. The set of digits, 0, 1, …, 9.
3. The underscore, _.

Symbols in the predicate calculus begin with a letter and are followed by any sequence of these legal characters.

Legitimate characters in the alphabet of predicate calculus symbols include

a R 6 9 p _ z

Examples of characters not in the alphabet include

% @ / & “ ”

Legitimate predicate calculus symbols include

George fire3 tom_and_jerry bill XXXX friends_of

Examples of strings that are not legal symbols are

3jack “no blanks allowed” ab%cd ***71 duck!!!

DEFINITION

SYMBOLS and TERMS

Predicate calculus symbols include:

1. *Truth symbols* true and false (these are reserved symbols).
2. *Constant symbols* are symbol expressions having the first character lowercase.
3. *Variable symbols* are symbol expressions beginning with an uppercase character.
4. *Function symbols* are symbol expressions having the first character lowercase. Functions have an attached arity indicating the number of elements of the domain mapped onto each element of the range.

A *function expression* consists of a function constant of arity n, followed by n terms, $t_1, t_2, \ldots, t_n$, enclosed in parentheses and separated by commas.

A predicate calculus *term* is either a constant, variable, or function expression.

DEFINITION

PREDICATES and ATOMIC SENTENCES

Predicate symbols are symbols beginning with a lowercase letter.

Predicates have an associated positive integer referred to as the *arity* or "argument number" for the predicate. Predicates with the same name but different arities are considered distinct.

An atomic sentence is a predicate constant of arity n, followed by n terms, $t_1, t_2, \ldots, t_n$, enclosed in parentheses and separated by commas.

The truth values, true and false, are also atomic sentences.

DEFINITION

PREDICATE CALCULUS SENTENCES

Every atomic sentence is a sentence.

1. If s is a sentence, then so is its negation, $\neg$ s.
2. If s_1 and s_2 are sentences, then so is their conjunction, $s_1 \wedge s_2$.
3. If s_1 and s_2 are sentences, then so is their disjunction, $s_1 \vee s_2$.
4. If s_1 and s_2 are sentences, then so is their implication, $s_1 \rightarrow s_2$.
5. If s_1 and s_2 are sentences, then so is their equivalence, $s_1 \equiv s_2$.
6. If X is a variable and s a sentence, then $\forall$ X s is a sentence.
7. If X is a variable and s a sentence, then $\exists$ X s is a sentence.

DEFINITION

INTERPRETATION

Let the domain D be a nonempty set.

An *interpretation* over D is an assignment of the entities of D to each of the constant, variable, predicate, and function symbols of a predicate calculus expression, such that:

1. Each constant is assigned an element of D.
2. Each variable is assigned to a nonempty subset of D; these are the allowable substitutions for that variable.
3. Each function f of arity m is defined on m arguments of D and defines a mapping from D^m into D.
4. Each predicate p of arity n is defined on n arguments from D and defines a mapping from D^n into {T, F}.

DEFINITION

TRUTH VALUE OF PREDICATE CALCULUS EXPRESSIONS

Assume an expression E and an interpretation I for E over a nonempty domain D. The truth value for E is determined by:

1. The value of a constant is the element of D it is assigned to by I.
2. The value of a variable is the set of elements of D it is assigned to by I.
3. The value of a function expression is that element of D obtained by evaluating the function for the parameter values assigned by the interpretation.
4. The value of truth symbol "true" is T and "false" is F.
5. The value of an atomic sentence is either T or F, as determined by the
6. The value of the negation of a sentence is T if the value of the sentence is F and is F if the value of the sentence is T.
7. The value of the conjunction of two sentences is T if the value of both sentences is T and is F otherwise.

8.–10. The truth value of expressions using $\vee$, $\rightarrow$, and $\equiv$ is determined from the value of their operands as defined in Section 2.1.2.

Finally, for a variable X and a sentence S containing X:

11. The value of $\forall$ X S is T if S is T for all assignments to X under I, and it is F otherwise.
12. The value of $\exists$ X S is T if there is an assignment to X in the interpretation under which S is T; otherwise it is F.

- First-order predicate calculus allows quantified variables to refer to objects in the domain of discourse, and not to predicate or functions.
- Almost any grammatically correct English sentence may be represented in first-order predicate calculus.
- The limitation of the predicate calculus is that it is difficult to represent possibility, time, and belief.

DEFINITION

PROOF PROCEDURE

A *proof procedure* is a combination of an inference rule and an algorithm for applying that rule to a set of logical expressions to generate new sentences.

We present proof procedures for the *resolution* inference rule in Chapter 12.

DEFINITION

LOGICALLY FOLLOWS, SOUND, and COMPLETE

A predicate calculus expression X *logically follows* from a set S of predicate calculus expressions if every interpretation and variable assignment that satisfies S also satisfies X.

An inference rule is *sound* if every predicate calculus expression produced by the rule from a set S of predicate calculus expressions also logically follows from S.

An inference rule is *complete* if, given a set S of predicate calculus expressions, the rule can infer every expression that logically follows from S.

DEFINITION

MODUS PONENS, MODUS TOLLENS, AND ELIMINATION, AND INTRODUCTION, and UNIVERSAL INSTANTIATION

If the sentences P and $P \rightarrow Q$ are known to be true, then *modus ponens* lets us infer Q.

Under the inference rule *modus tollens*, if $P \rightarrow Q$ is known to be true and Q is known to be false, we can infer $\neg P$.

And elimination allows us to infer the truth of either of the conjuncts from the truth of a conjunctive sentence. For instance, $P \wedge Q$ lets us conclude P and Q are true.

And introduction lets us infer the truth of a conjunction from the truth of its conjuncts. For instance, if P and Q are true, then $P \wedge Q$ is true.

Universal instantiation states that if any universally quantified variable in a true sentence is replaced by any appropriate term from the domain, the result is a true sentence. Thus, if a is from the domain of X, $\forall X\, p(X)$ lets us infer p(a).

Unification

- It is an algorithm for determining the substitutions needed to make two predicate calculus expressions match.
- A variable cannot be unified with a term containing that variable. The test for it is called the occurs check.

```
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 lists are of different sizes
         otherwise:                                                  %both E1 and E2 are lists
            begin
                  HE1 := first element of E1;
                  HE2 := first element of E2;
                  SUBS1 := unify(HE1,HE2);
                  if SUBS1 : = FAIL then return FAIL;
                  TE1 := apply(SUBS1, rest of E1);
                  TE2 : = apply (SUBS1, rest of E2);
                  SUBS2 : = unify(TE1, TE2);
                  if SUBS2 = FAIL then return FAIL;
                        else return composition(SUBS1,SUBS2)
            end
   end                                                                          %end case
end
```

Resolution refutation proofs involve the following steps:

1. Put the premises or axioms into *clause form* (13.2.2).
2. Add the negation of what is to be proved, in clause form, to the set of axioms.
3. *Resolve* these clauses together, producing new clauses that logically follow from them (13.2.3).
4. Produce a contradiction by generating the empty clause.
5. The substitutions used to produce the empty clause are those under which the opposite of the negated goal is true (13.2.4).

Algorithm to convert to clausal form (1)

1. Eliminate conditionals $\rightarrow$, using the equivalence

   $P \rightarrow Q = \neg P \vee Q$

   e.g, $(\exists X)\ (p(X) \wedge (\forall Y)\ (f(Y) \rightarrow h(X,Y)))$ becomes

   $(\exists X)\ (p(X) \wedge (\forall Y)\ (\neg f(Y) \vee h(X,Y)))$

2. Eliminate negations or reduce the scope of negation to one atom.

   e.g., $\neg\neg P = P$

   $\neg(P \wedge Q) = \neg P \vee \neg Q$

   $\neg (\exists X)\ p(X) = (\forall X) \neg p(X)$

   $\neg (\forall X)\ p(X) = (\exists X) \neg p(X)$

3. Standardize variables within a WFF so that the bound or dummy variables of each quantifier have unique names.

   e.g., $(\exists X) \neg p(X) \vee (\forall X)\ p(X)$ is replaced by

   $(\exists X) \neg p(X) \vee (\forall Y)\ p(Y)$

Algorithm to convert to clausal form (2)

4. Eliminate existential quantifiers, by using Skolem functions, named after the Norwegian logician Thoralf Skolem.

   e.g., $(\exists X)\ m(X)$ is replaced by $m(a)$

   $(\forall X)\ (\exists Y)\ k(X, Y)$ is replaced by

   $(\forall X)\ k(X, f(X))$

5. Convert the WFF to prenex form which is a sequence of quantifiers followed by a matrix.

   e.g., $(\exists X)\ (p(X) \wedge (\forall Y)\ (\neg f(Y) \vee h(X,Y)))$ becomes

   $(\forall Y)\ (p(a) \wedge (\neg f(Y) \vee h(a,Y)))$

6. Convert the matrix to conjunctive normal form, which is a conjunctive of clauses. Each clause is a disjunction.

   e.g., $P \vee (Q \wedge R) = (P \vee Q) \wedge (P \vee R)$

7. Drop the universal quantifiers.

   e.g.,WFF(*) becomes $p(a) \wedge (\neg f(Y) \vee h(a,Y))$

8. Eliminate the conjunctive signs by writing the WFF as a set of clauses

   e.g., WFF(*) becomes $p(a)$

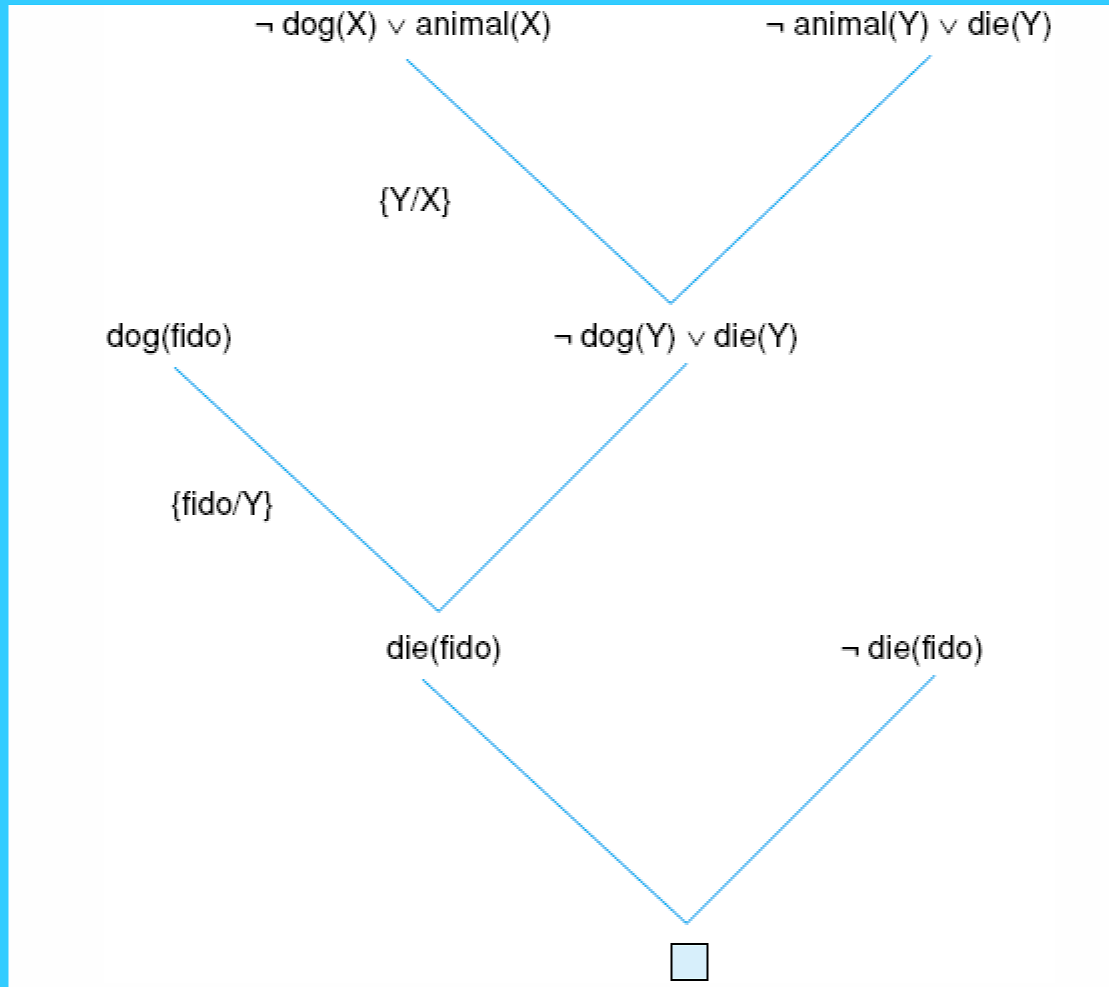
   $(\neg f(Y) \vee h(a,Y))$

9. Rename variables in clauses, if necessary, so that the same variable name is only used in one clause.

   e.g., $p(X) \vee q(X) \vee k(X,Y)$ and $\neg p(X) \vee q(Y)$ become

   $p(X) \vee q(X) \vee k(X,Y)$ and $\neg p(X1) \vee q(Y1)$

Fig 13.3 Resolution proof for the “dead dog” problem.



Anyone passing his history exams and winning the lottery is happy.

$\forall$ **X (pass (X,history) $\wedge$ win (X,lottery) $\rightarrow$ happy (X))**

Anyone who studies or is lucky can pass all his exams.

$\forall$ **X** $\forall$ **Y (study (X) $\vee$ lucky (X) $\rightarrow$ pass (X,Y))**

John did not study but he is lucky.

$\neg$ **study (john) $\wedge$ lucky (john)**

Anyone who is lucky wins the lottery.

$\forall$ **X (lucky (X) $\rightarrow$ win (X,lottery))**

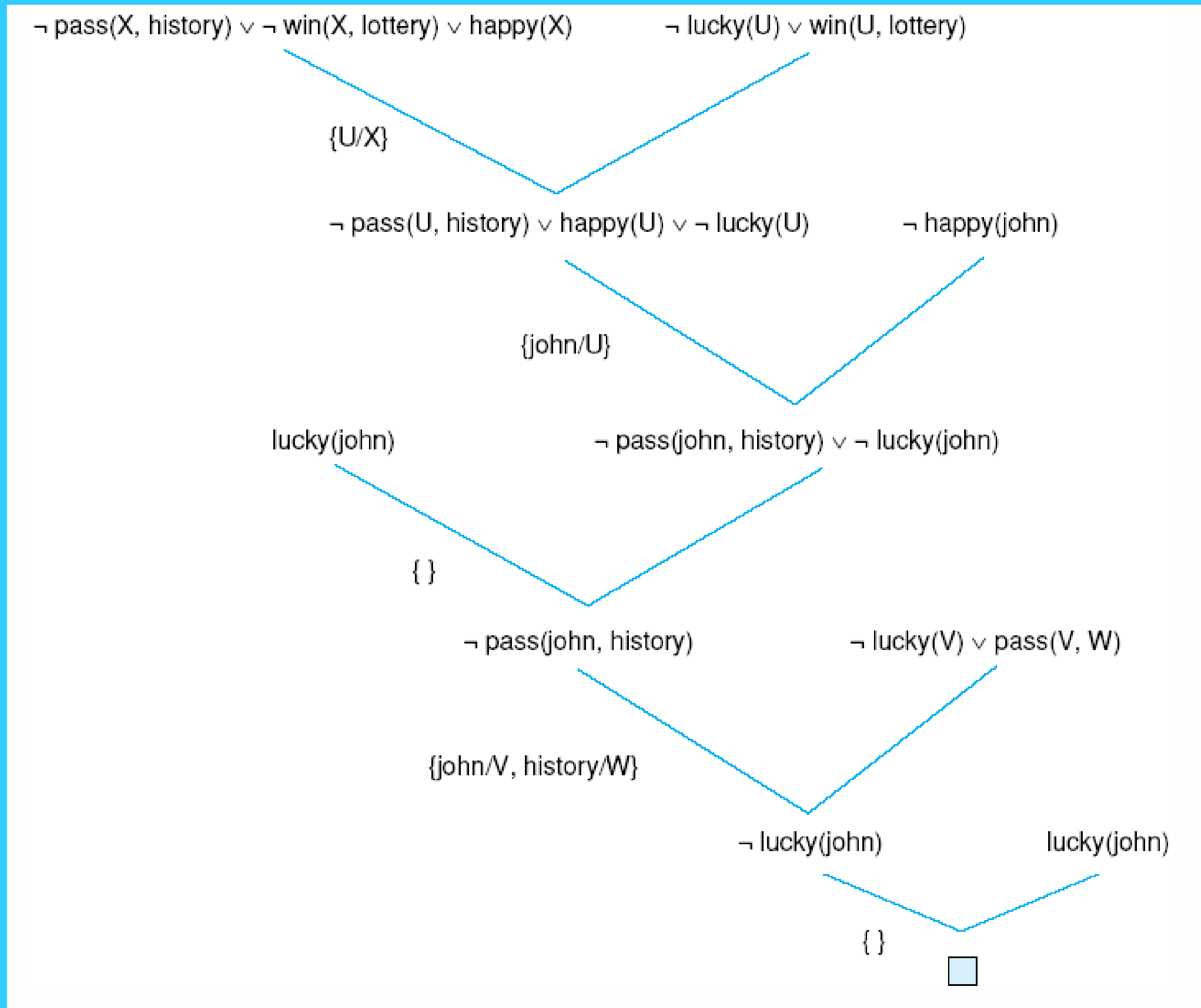
These four predicate statements are now changed to clause form (Section 12.2.2):

1. $\neg$ **pass (X, history) $\vee$ $\neg$ win (X, lottery) $\vee$ happy (X)**
2. $\neg$ **study (Y) $\vee$ pass (Y, Z)**
3. $\neg$ **lucky (W) $\vee$ pass (W, V)**
4. $\neg$ **study (john)**
5. **lucky (john)**
6. $\neg$ **lucky (U) $\vee$ win (U, lottery)**

Into these clauses is entered, in clause form, the negation of the conclusion:

7. $\neg$ **happy (john)**

Fig 13.5 One refutation for the “happy student” problem.



State Space Search

The goal states are described by

- a measurable property of the states, or
- a property of the path developed in the search.

To design a search algorithm, we must consider

- Is it guaranteed to find a solution?
- Is it guaranteed to find an optimal solution?
- What is its complexity?
- Whether the complexity can be reduced?

Search algorithms must detect and eliminate loops from potential solution paths.

Fig 3.12 State space in which goal-directed search effectively prunes extraneous search paths.

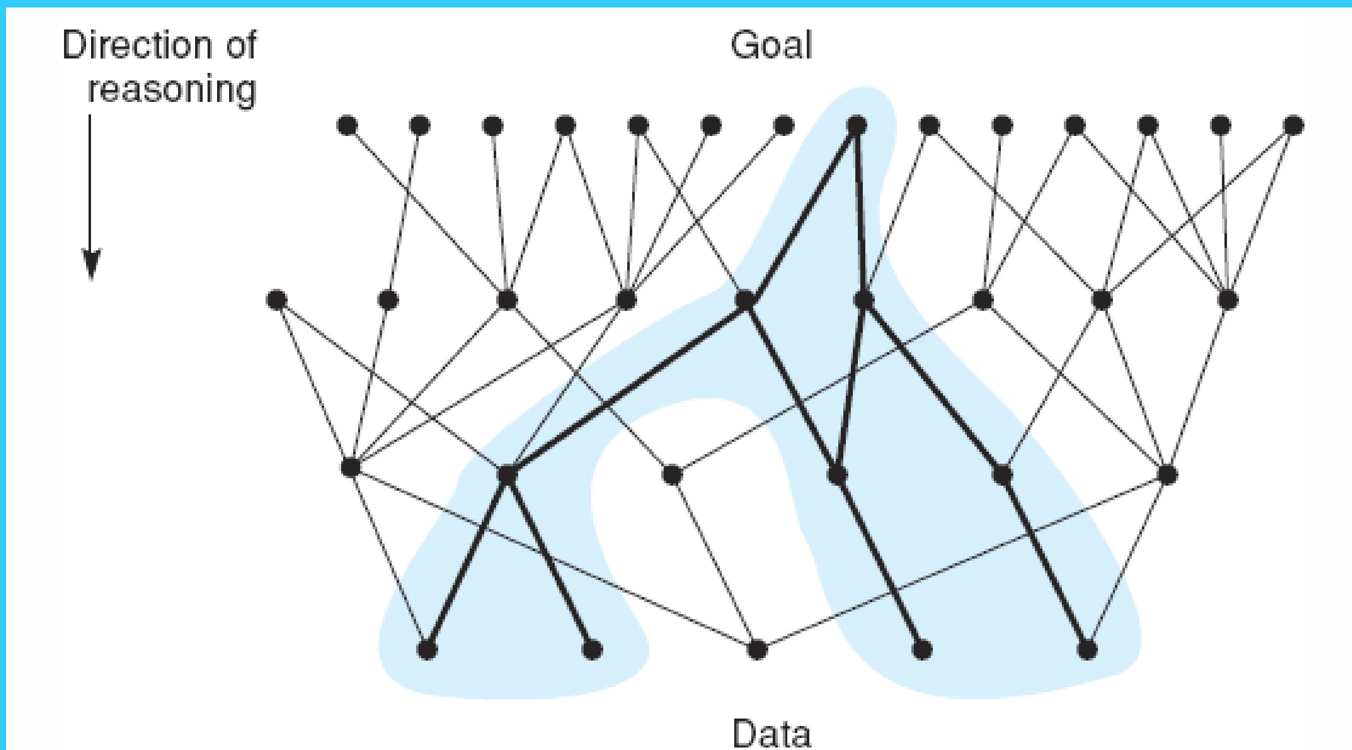
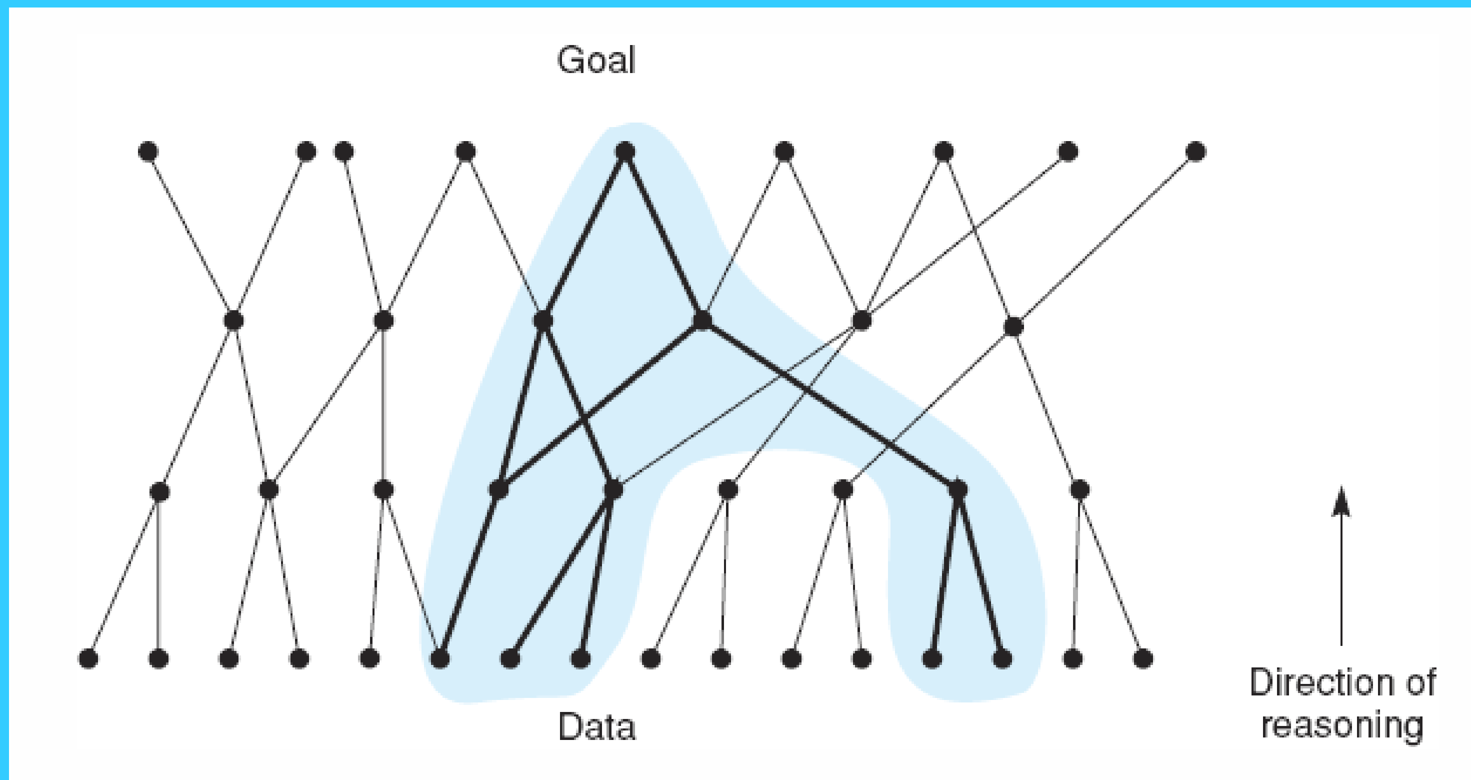
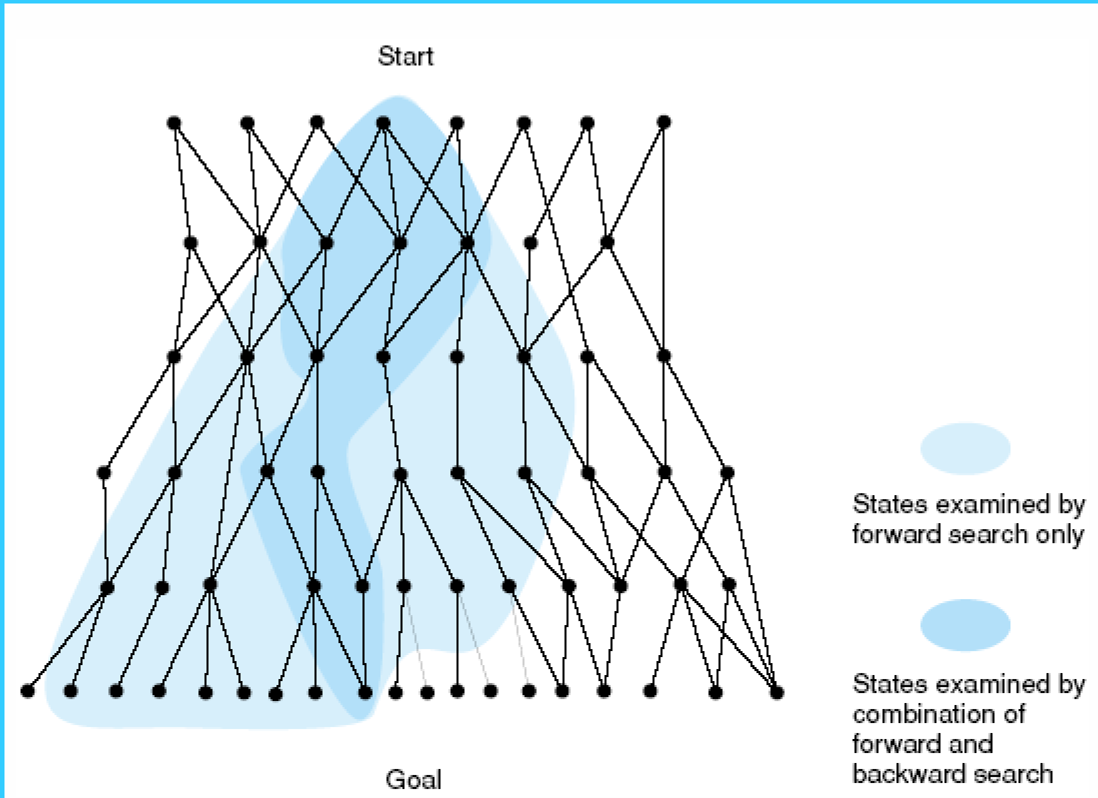


Fig 3.13 State space in which data-directed search prunes irrelevant data and their consequents and determines one of a number of possible goals.



- Data-driven and goal-driven searches search the same state space graph; but, the order and actual number of state searched can be differ.
- The preferred strategy is determined by the properties of the problem itself:
  – the complexity of the rules used for changing states
  – the shape of the state space
  – the nature and availability of the data

Fig 6.12 Bidirectional search meeting in the middle, eliminating much of the space examined by unidirectional search.



- Backtracking is a technique for systematically trying all different paths through a state space
  - For a data-driven search, it begins at the start state and pursues a path until it reaches either a goal or a "dead end".
    - If it finds a goal, it quits and returns the solution path.
    - Otherwise, it "backtracks" to the most recent node on the path having unexamined siblings and continues down one of these branches.
  - For a goal-driven search, it takes a goal be the root and evaluates descendants back in an attempt to find a start state.
  - The depth-first and breadth-first searches exploit the ideas used in backtrack.

Function backtrack algorithm

```
function backtrack;

begin
   SL := [Start];  NSL := [Start];  DE := [ ];  CS := Start;                    % initialize:
   while NSL ≠ [ ] do                                  % while there are states to be tried
      begin
         if CS = goal (or meets goal description)
            then return SL;                        % on success, return list of states in path.
         if CS has no children (excluding nodes already on DE, SL, and NSL)
            then begin
               while SL is not empty and CS = the first element of SL do
                  begin
                     add CS to DE;                               % record state as dead end
                     remove first element from SL;                              %backtrack
                     remove first element from NSL;
                     CS := first element of NSL;
                  end
               add CS to SL;
            end
            else begin
               place children of CS (except nodes already on DE, SL, or NSL) on NSL;
               CS := first element of NSL;
               add CS to SL
            end
      end;
      return FAIL;
end.
```

Fig 3.14 Backtracking search of a hypothetical state space space.

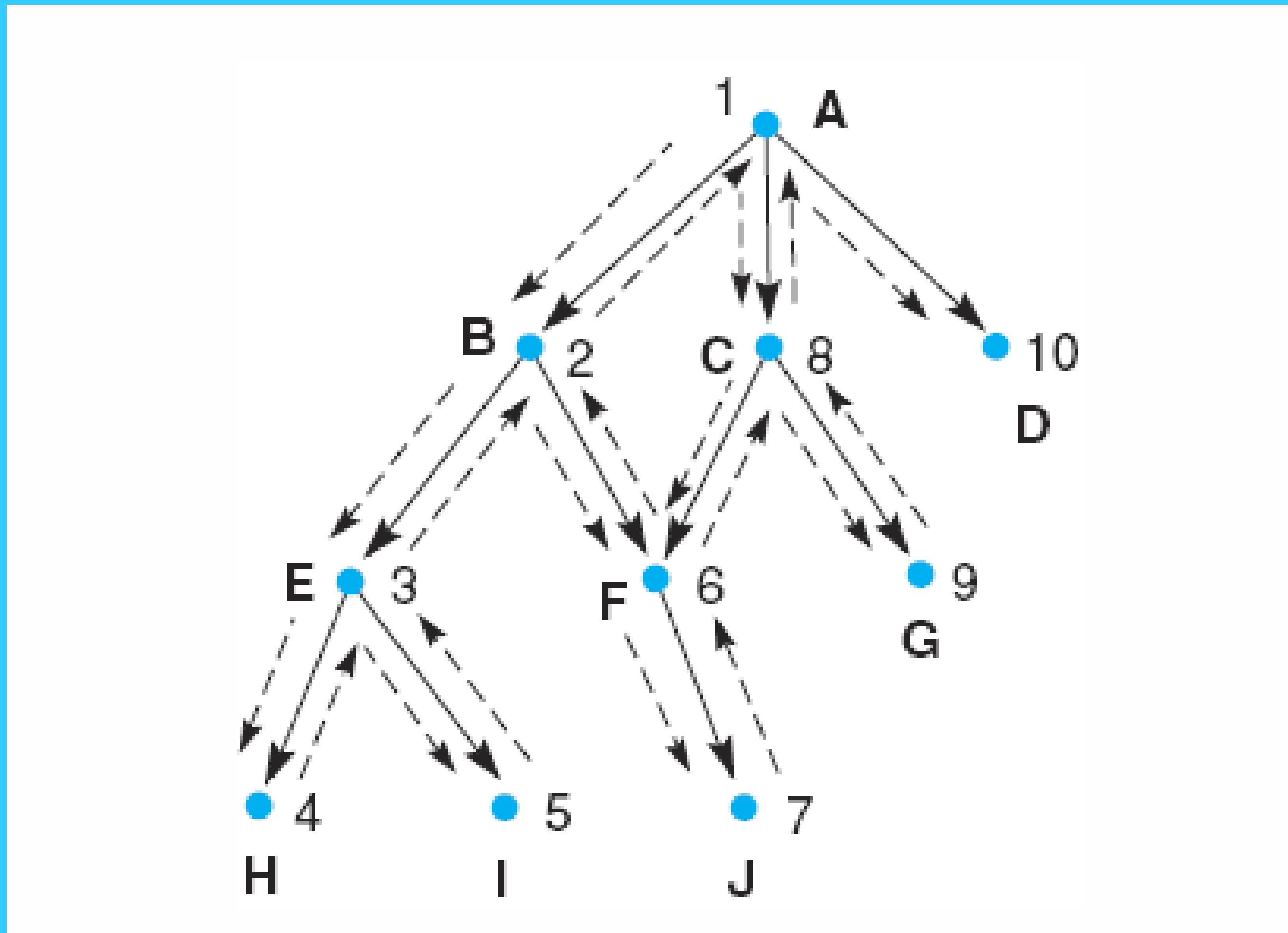
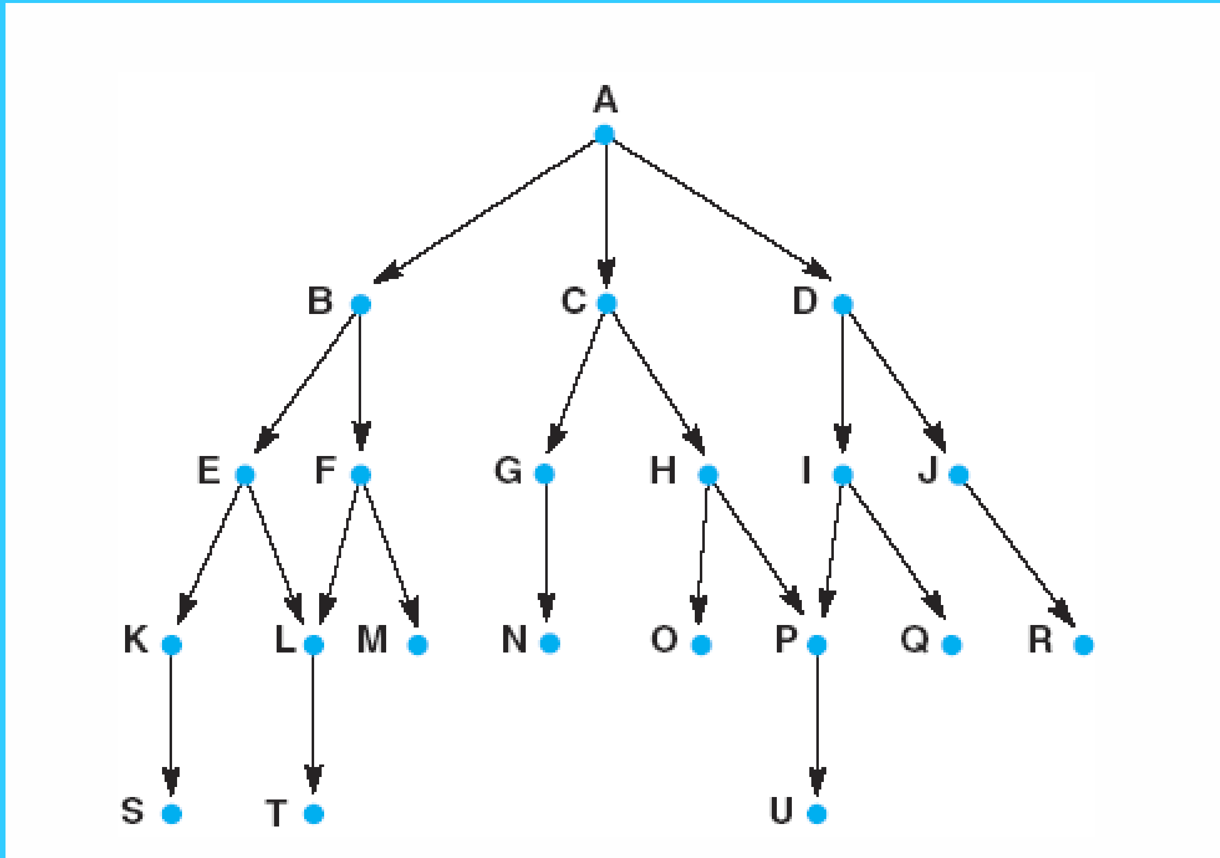


Fig 3.15 Graph for breadth - first search example



Function depth_first_search algorithm

```
begin
   open := [Start];                                              % initialize
   closed := [ ];
   while open ≠ [ ] do                                           % states remain
      begin
         remove leftmost state from open, call it X;
         if X is a goal then return SUCCESS                      % goal found
            else begin
               generate children of X;
               put X on closed;
               discard children of X if already on open or closed;   % loop check
               put remaining children on left end of open            % stack
            end
      end;
   return FAIL                                                   % no states left
end.
```

The choice of depth-first or breadth-first search depends on the problem

- the importance of finding the optimal path to a goal
- the branching of the state space
- the available time and space resources
- the average length of paths to a goal node

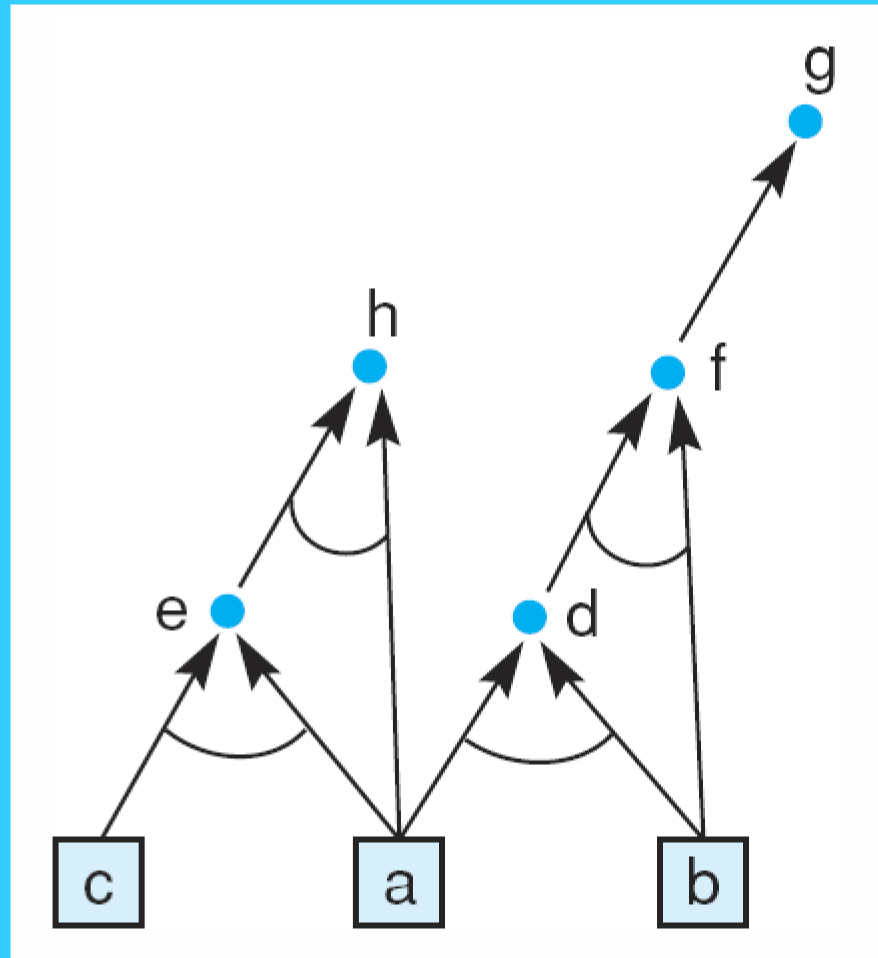
A trade-off – Depth-first search with a depth bound

- The depth bound forces a failure on a search path once it gets below a certain level. This causes a breadth like sweep of the space at that depth level.
- It is preferred when time constraints exist or a solution is known within a certain depth.
- It is guaranteed to find a shortest path to a goal.
- Depth-first iterative deepening (Korf 1987) -- depth bound increases one at each iteration

- For all uniformed search algorithms, the worst-case time complexity is exponential.

- The reasoning with the predicate calculus can be represented by and/or graphs.
  - The and/or graph is an extension of the basic state space model.
  - They are used for many AI problems, such as theorem proving and expert systems.

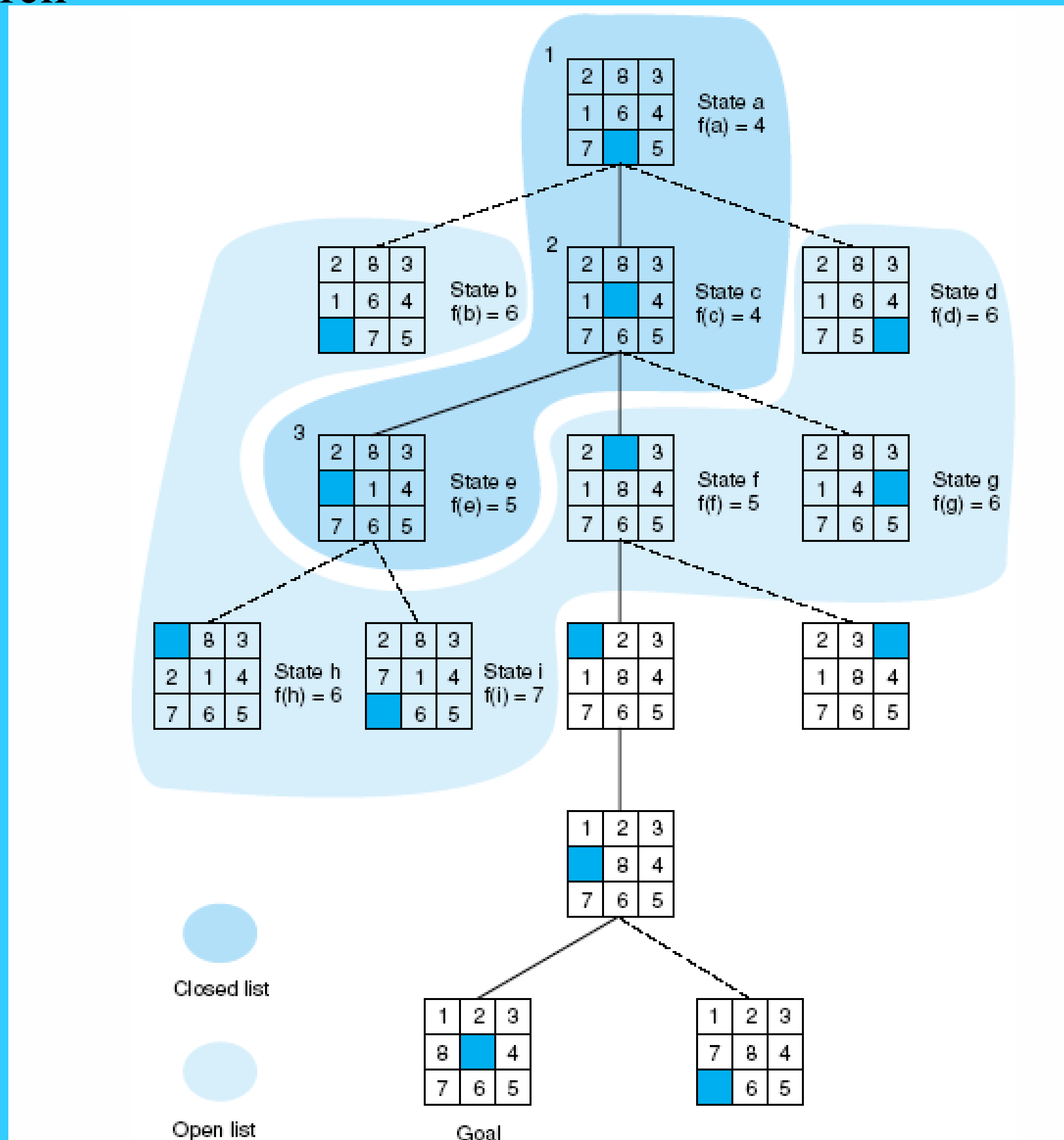
Fig 3.23 And/or graph of a set of propositional calculus expressions.



```
function best_first_search;

begin
   open := [Start];                                                  % initialize
   closed := [ ];
   while open ≠ [ ] do                                               % states remain
      begin
         remove the leftmost state from open, call it X;
         if X = goal then return the path from Start to X
         else begin
               generate children of X;
               for each child of X do
               case
                  the child is not on open or closed:
                     begin
                        assign the child a heuristic value;
                        add the child to open
                     end;
                  the child is already on open:
                     if the child was reached by a shorter path
                     then give the state on open the shorter path
                  the child is already on closed:
                     if the child was reached by a shorter path then
                        begin
                           remove the state from closed;
                           add the child to open
                        end;
               end;                                                  % case
               put X on closed;
               re-order states on open by heuristic merit (best leftmost)
            end;
return FAIL                                                          % open is empty
end.
```

Fig 4.17 Open and closed as they appear after the 3rd iteration of heuristic search



DEFINITION

ALGORITHM A, ADMISSIBILITY, ALGORITHM A*

Consider the evaluation function $f(n) = g(n) + h(n)$, where

n is any state encountered in the search.

g(n) is the cost of n from the start state.

h(n) is the heuristic estimate of the cost of going from n to a goal.

If this evaluation function is used with the best_first_search algorithm of Section 4.1, the result is called *algorithm A*.

A search algorithm is *admissible* if, for any graph, it always terminates in the optimal solution path whenever a path from the start to a goal state exists.

If algorithm A is used with an evaluation function in which h(n) is less than or equal to the cost of the minimal path from n to the goal, the resulting search algorithm is called *algorithm* A* (pronounced "A STAR").

It is now possible to state a property of A* algorithms:

All A* algorithms are admissible.

DEFINITION

INFORMEDNESS

For two A* heuristics h_1 and h_2, if $h_1(n) \le h_2(n)$, for all states n in the search space, heuristic h_2 is said to be *more informed* than h_1.

Fig 4.18 Comparison of state space searched using heuristic search with space searched by breadth-first search. The proportion of the graph searched heuristically is shaded. The optimal search selection is in bold. Heuristic used is $f(n) = g(n) + h(n)$ where $h(n)$ is tiles out of place.

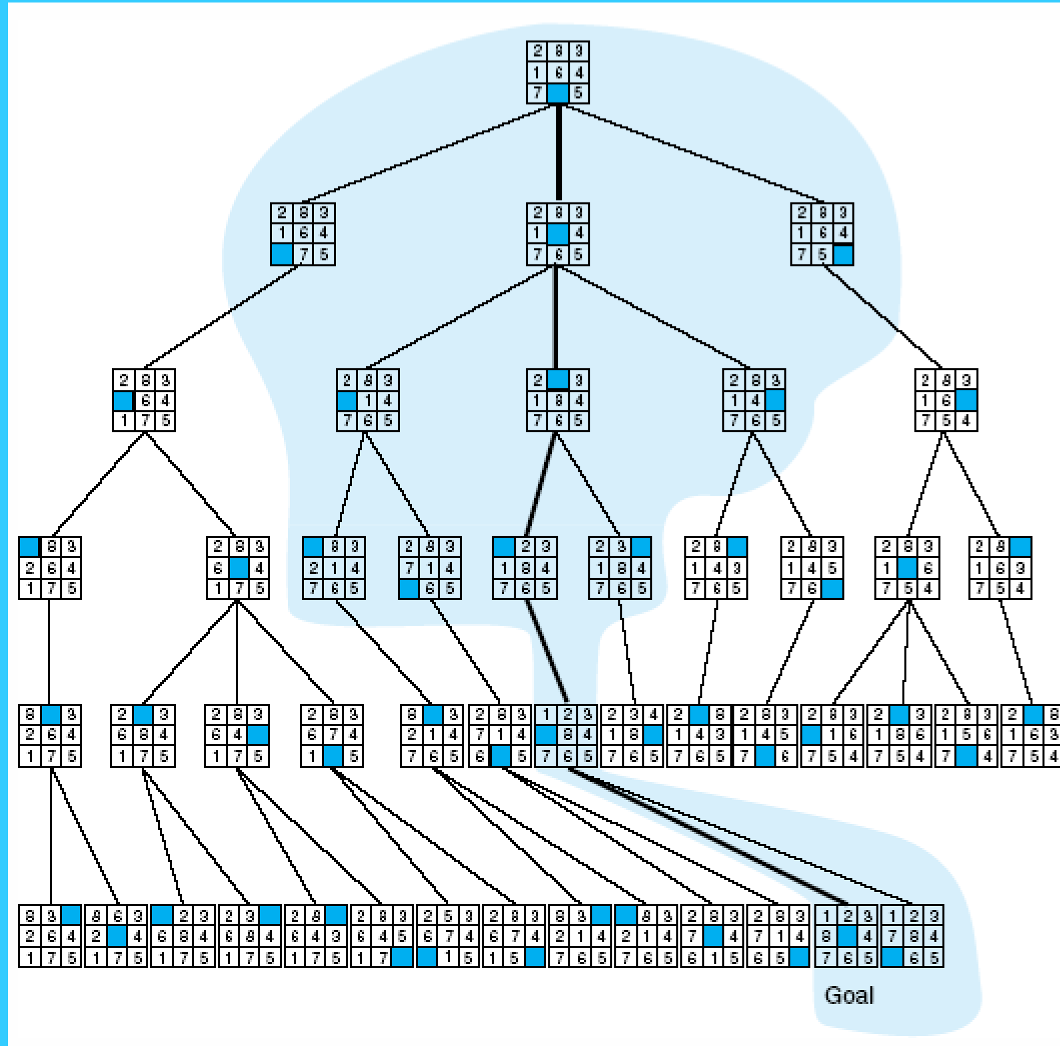


Fig 4.20 Exhaustive minimax for the game of nim. Bold lines indicate forced win for MAX. Each node is marked with its derived value (0 or 1) under minimax.

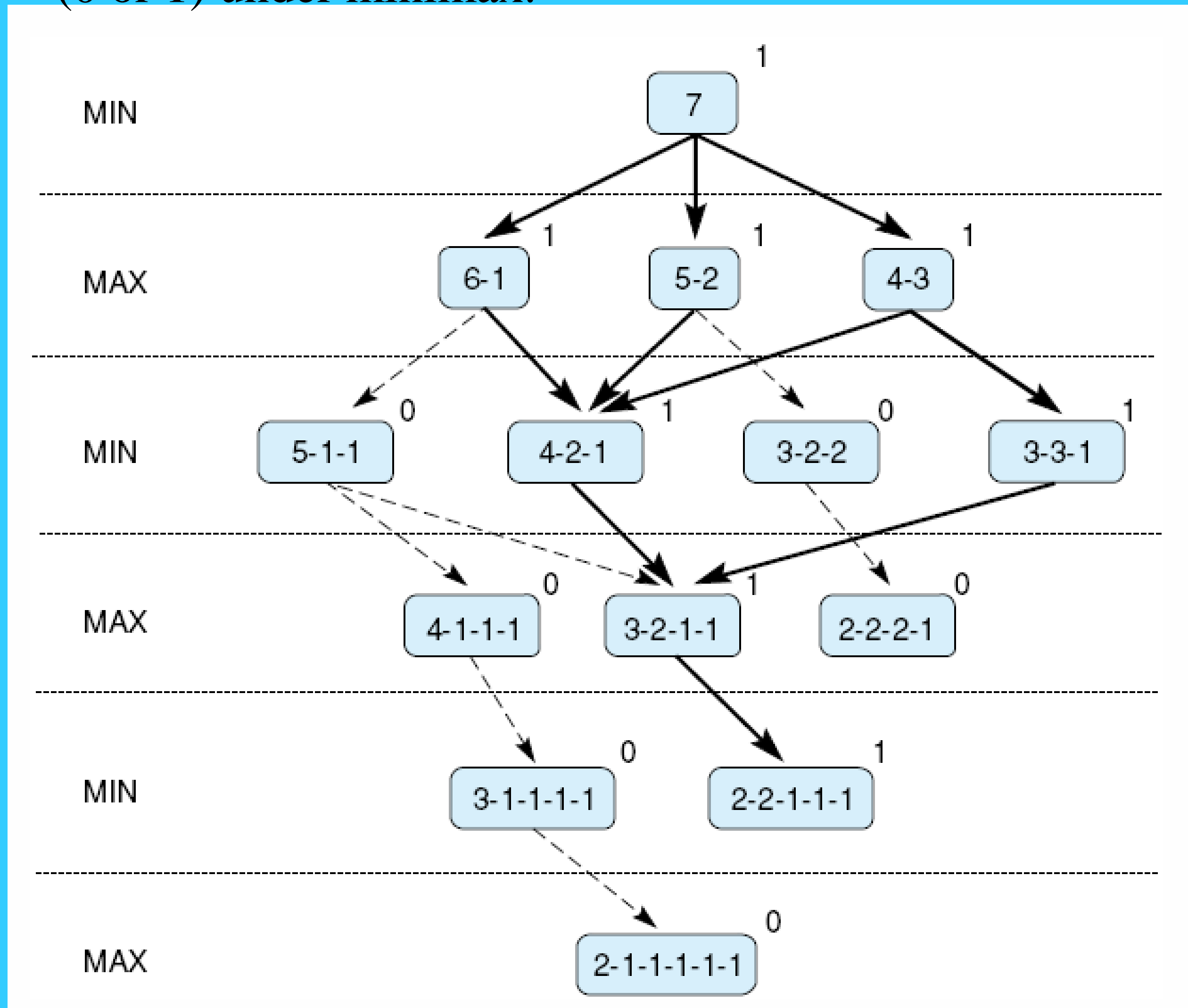


Fig 4.21 Minimax to a hypothetical state space. Leafstates show heuristic values; internal states show backed-up values.

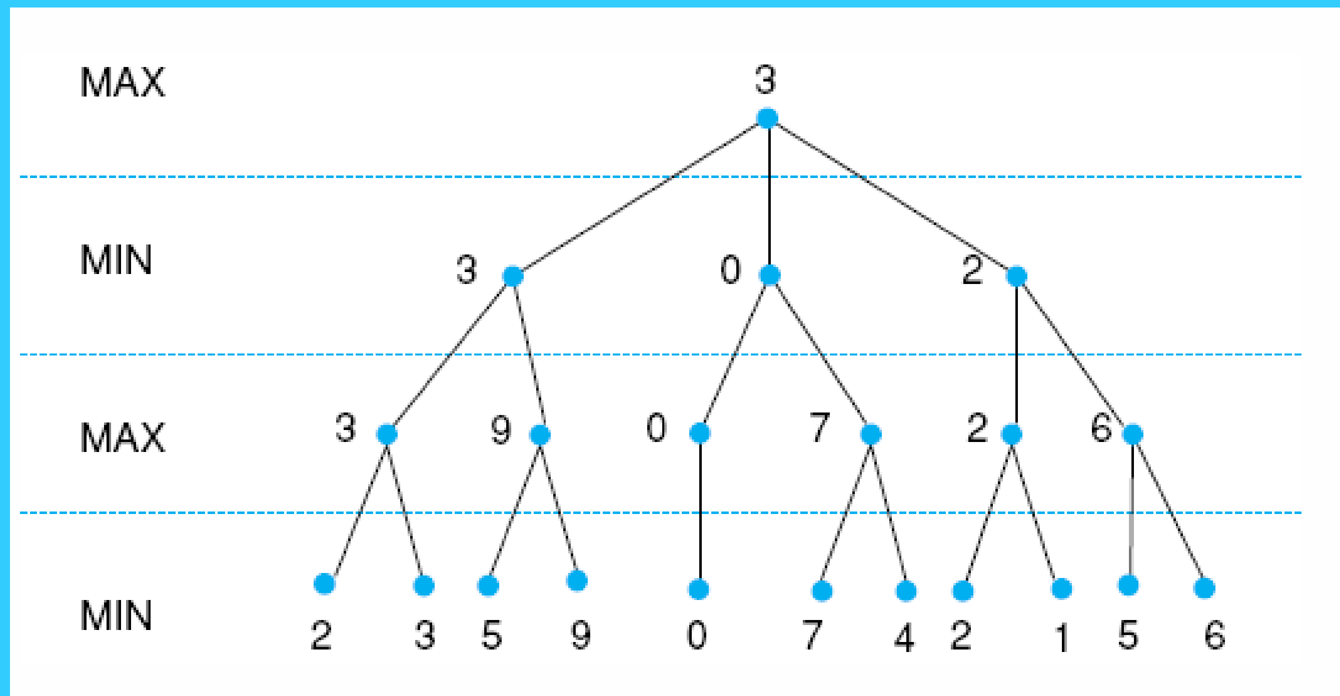


Fig 4.26 Alpha-beta pruning applied to state space of Fig 4.21. States without numbers are not evaluated.

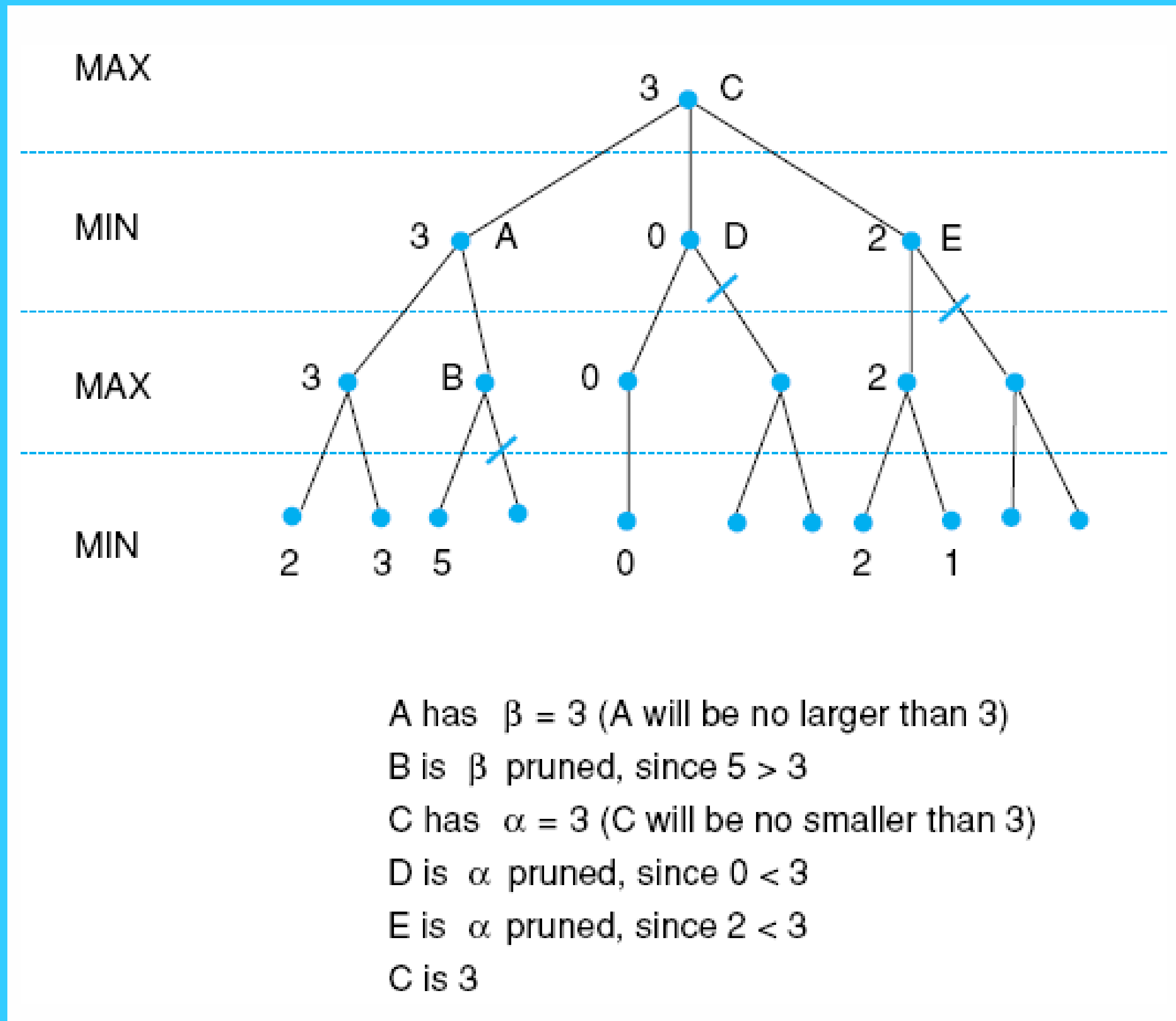


Fig 6.1 A production system. Control loops until working memory pattern no longer matches the conditions of any productions.

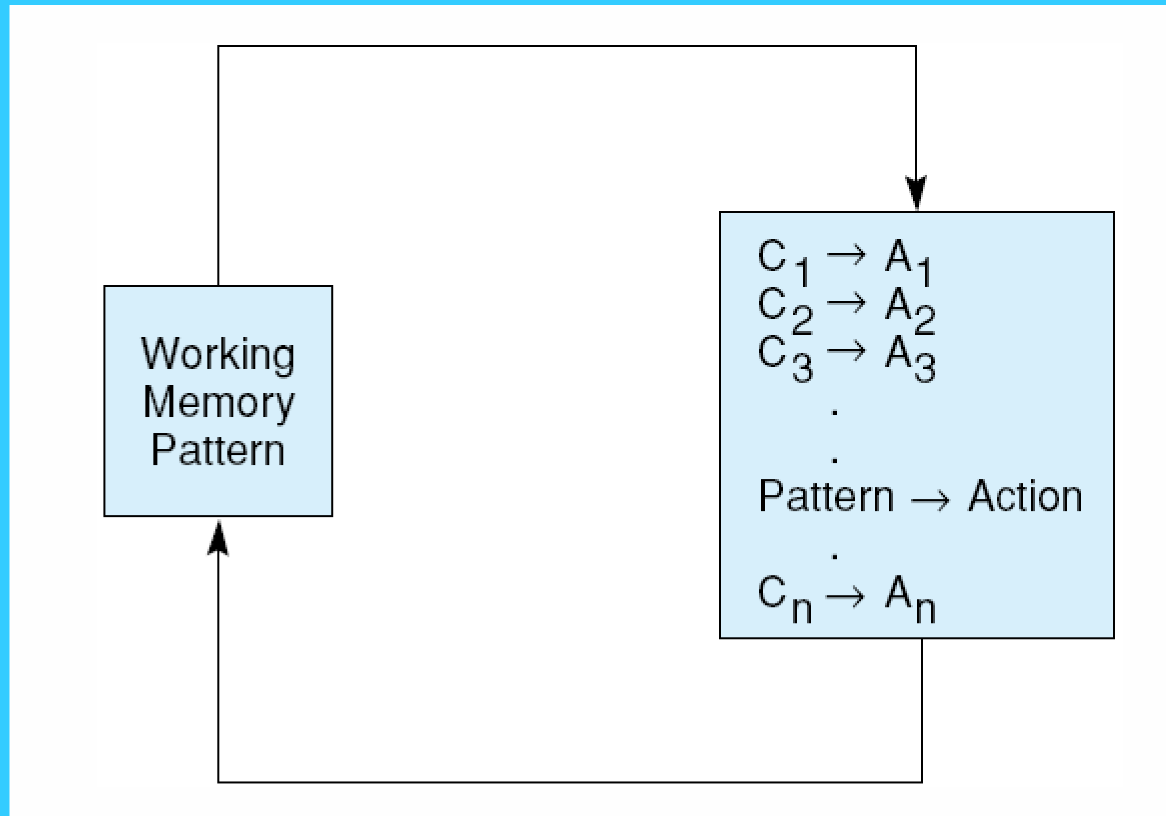


Fig 6.3 The 8-puzzle as a production system.

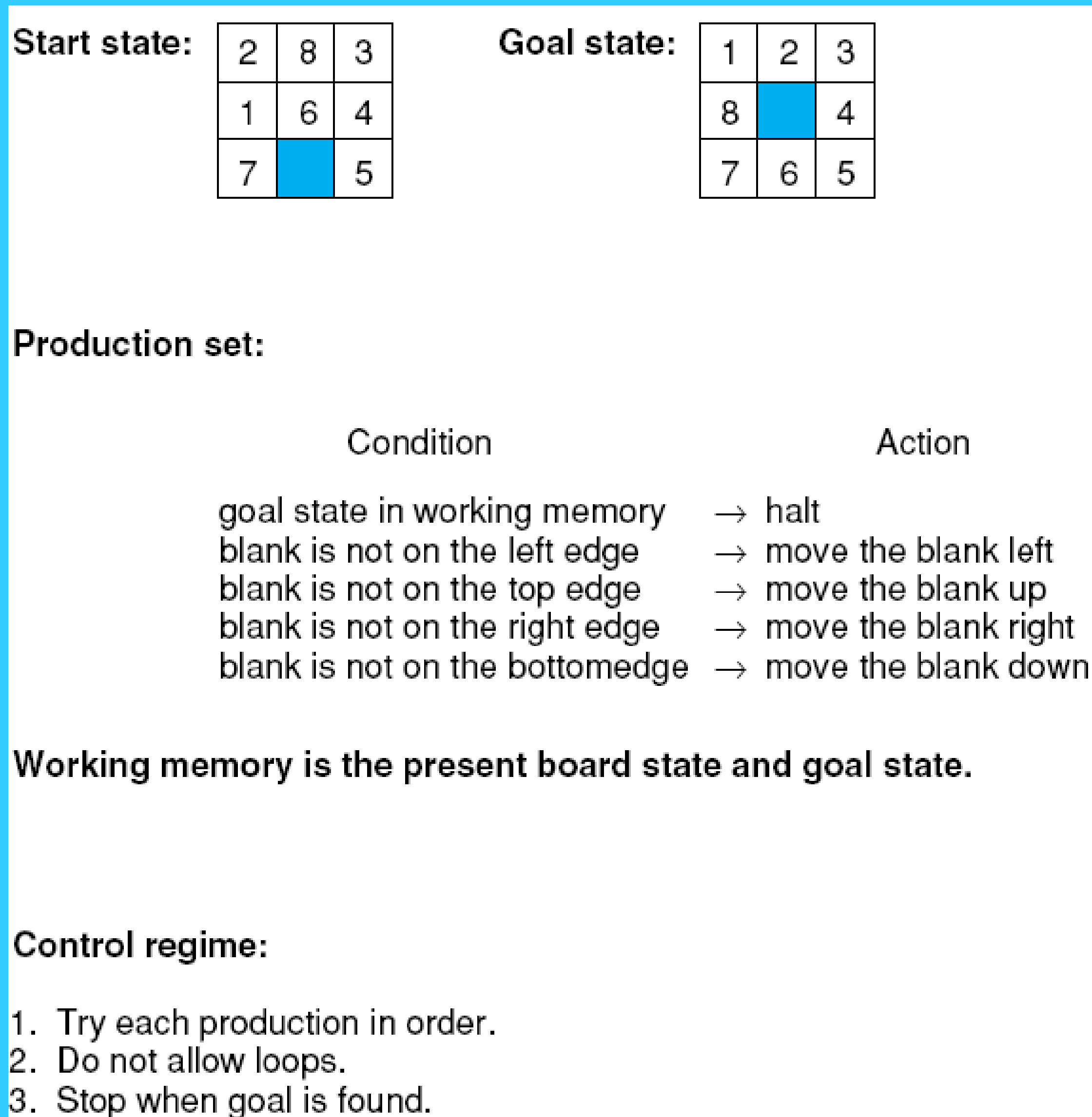
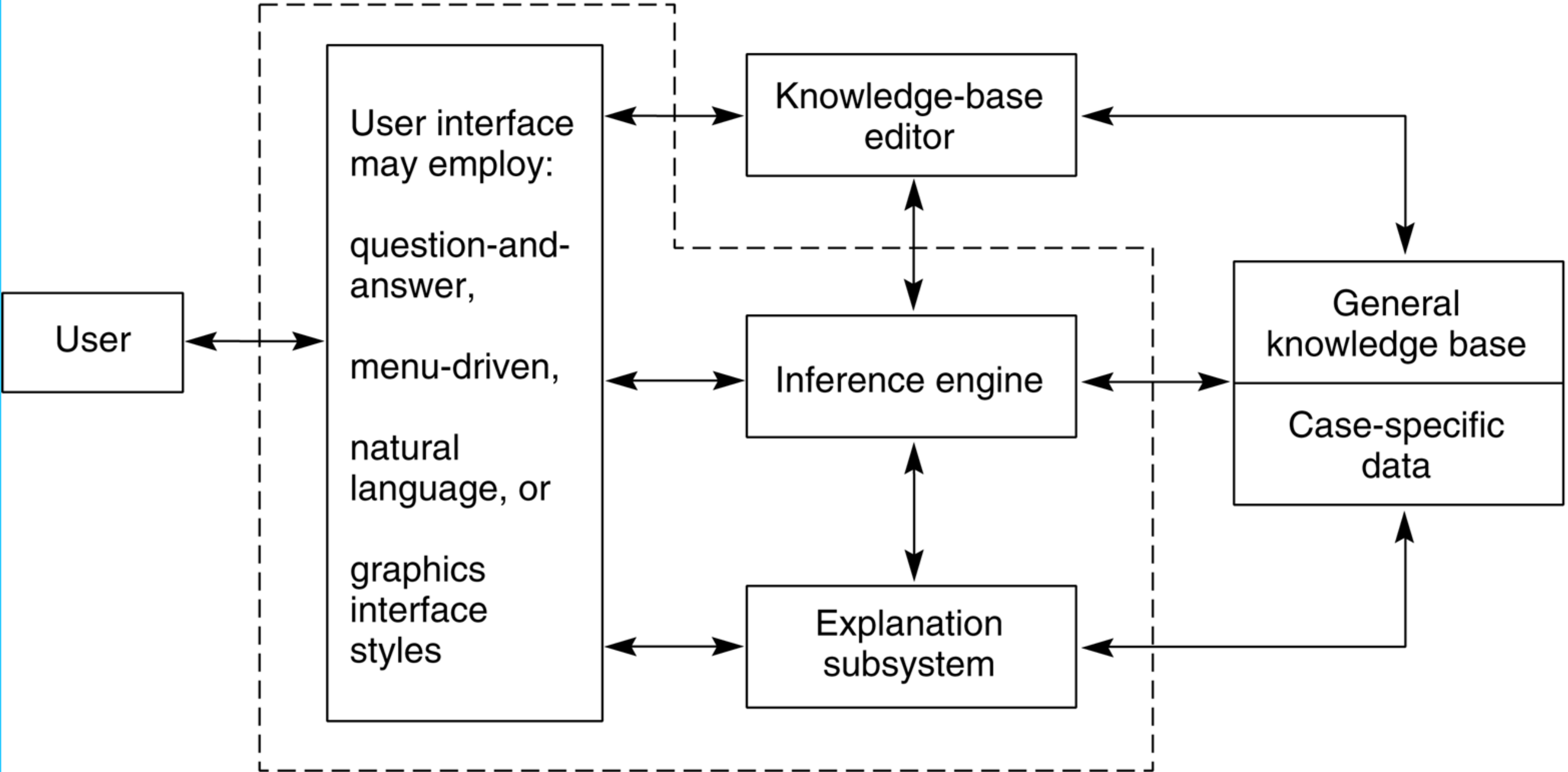


Fig 8.1 architecture of a typical expert system for a particular problem domain.



Guidelines to determine whether a problem is appropriate for expert system solution:

1. The need for the solution justifies the cost and effort of building an expert system.
2. Human expertise is not available in all situations where it is needed.
3. The problem may be solved using symbolic reasoning.
4. The problem domain is well structured and does not require commonsense reasoning.
5. The problem may not be solved using traditional computing methods.
6. Cooperative and articulate experts exist.
7. The problem is of proper size and scope.

A small expert system for analysis of automotive problems.

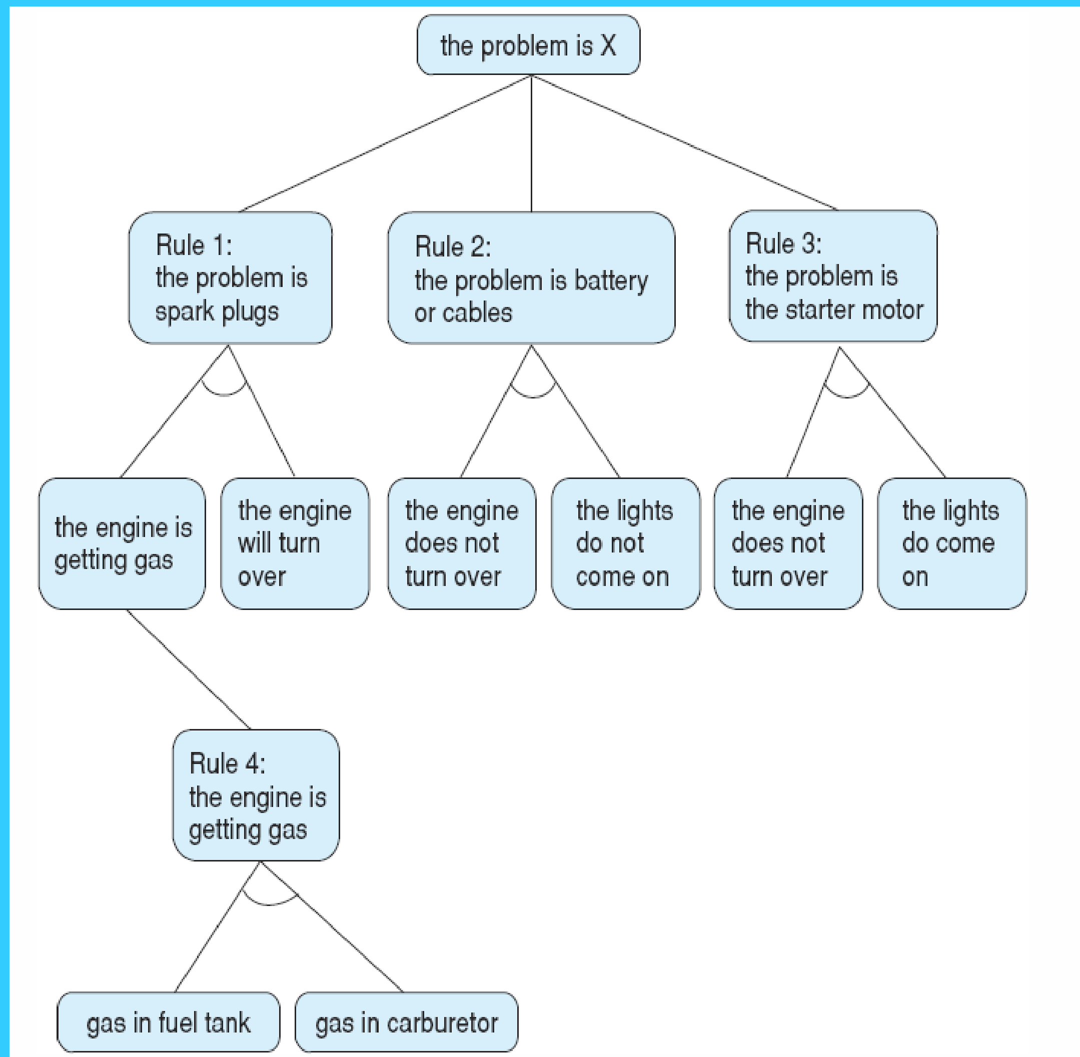
Rule 1: if
the engine is getting gas, and
the engine will turn over,
then
the problem is spark plugs.

Rule 2: if
the engine does not turn over, and
the lights do not come on
then
the problem is battery or cables.

Rule 3: if
the engine does not turn over, and
the lights do come on
then
the problem is the starter motor.

Rule 4: if
there is gas in the fuel tank, and
there is gas in the carburetor
then
the engine is getting gas.

Fig 8.8 The and/or graph searched in the car diagnosis example, with the conclusion of Rule 4 matching the first premise of Rule 1.



The following dialogue begins with the computer asking the user about the goals present in working memory.

Gas in fuel tank?
Yes
Gas in carburetor?
Yes
Engine will turn over?
Why

It has been established that:
1. The engine is getting gas,
2. The engine will turn over,
Then the problem is the spark plugs.

How the engine is getting gas

This follows from rule 4:
if
gas in fuel tank, and
gas in carburetor
then
engine is getting gas.
gas in fuel tank was given by the user
gas in carburetor was given by the user

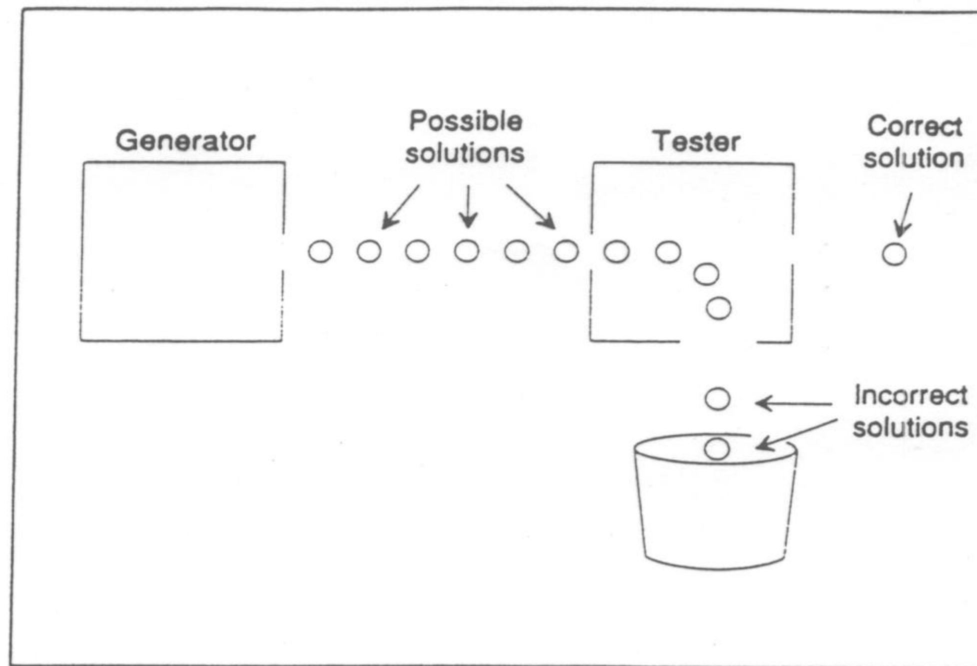
Heuristic and control in expert systems

- the inference engine
- the structure of the rules in the knowledge base, e.g., the order of the premises
- the contents of the rules

Problem Solver

- Problem Solving is the process of developing a sequence of actions to achieve a goal.
- This broad definition admits all goal directed AI programs to the ranks of problem solvers.
- Some more problem solving methods:
  - Generate and Test
  - Means-Ends Analysis
  - Problem Reduction
  - Decision Table
  - Decision Tree
- Few real problems can be solved by a single problem-solving method. Accordingly, it is often seen that problem-solving methods are working together.

Generate and Test



To perform generate and test,

- Until a satisfactory solution is found or no more candidate solutions can be generated.
  - Generate a candidate solution.
  - Test the candidate solution.
- If an acceptable solution is found, announce it: otherwise, announce failure.

Figure 3.3 Means–ends analysis involves states and procedures for reducing differences between states. The current state and goal state are shown solid; other states, not yet encountered, are shown dotted.

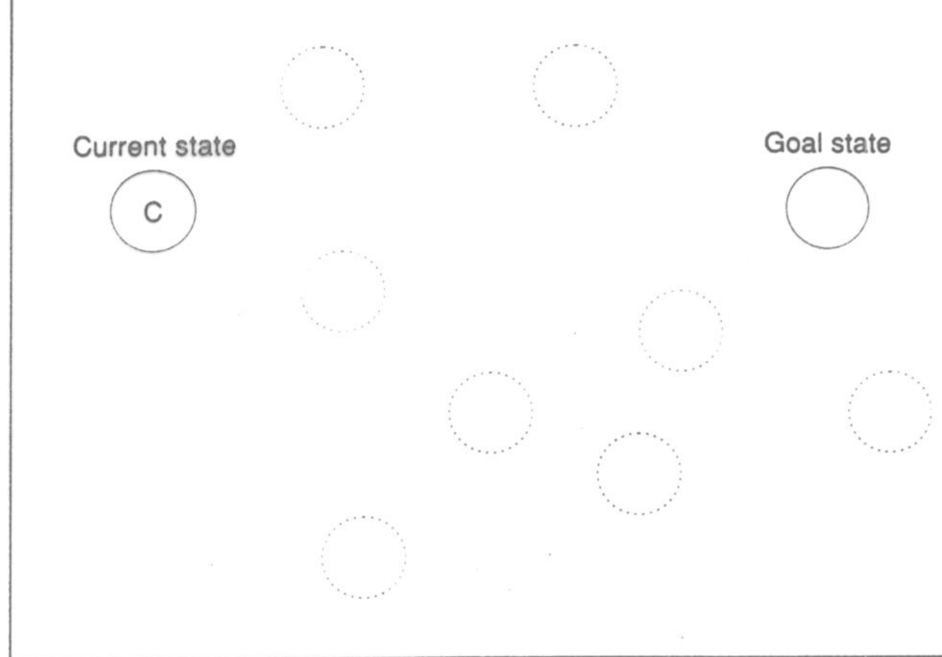
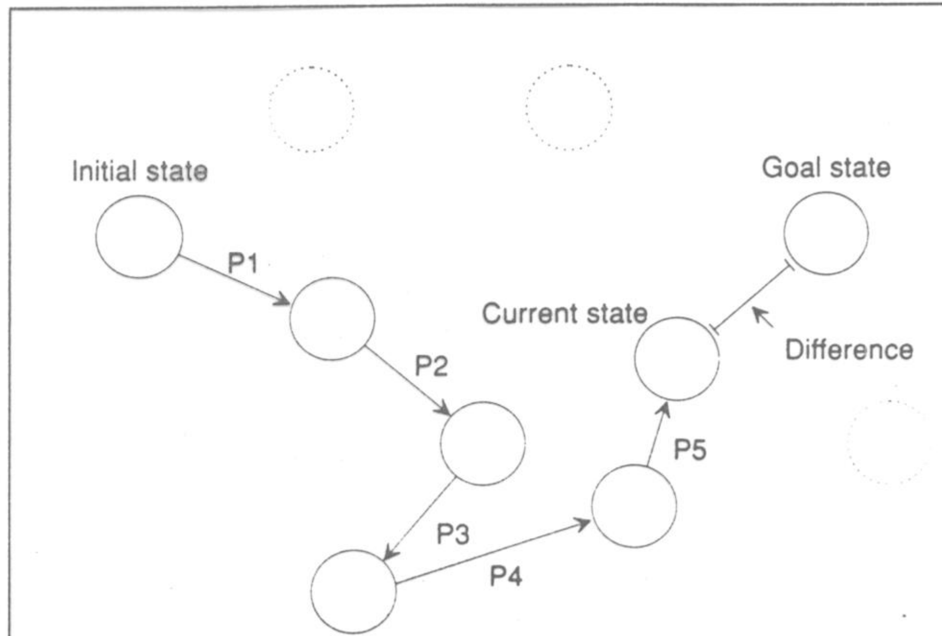


Figure 3.4 Means–ends analysis produces a path through state space. The current state, the goal state, and a description of their difference determine which procedure to try next. Note that the procedures are expected, but not guaranteed, to cause a transition to a state that is nearer the goal state than is the current state.



Difference-Procedure Tables Often Determine the Means

Whenever the description of the difference between the current state and the goal state is the key to which procedure to try next, a simple **difference-procedure table** may suffice to connect difference descriptions to preferred procedures.†

Consider, for example, a travel situation in which the problem is to find a way to get from one city to another. One traveler's preferences might link the preferred transportation procedure to the difference between states, described in terms of the distance between the cities involved, via the following difference-procedure table:

Distance	Airplane	Train	Car
More than 300 miles	√		
Between 100 and 300 miles		√	
Less than 100 miles			√

Problem reduction tries to convert difficult goals into easier-to-achieve subgoals. Each subgoal, in turn, may be divided into lower-level subgoals.

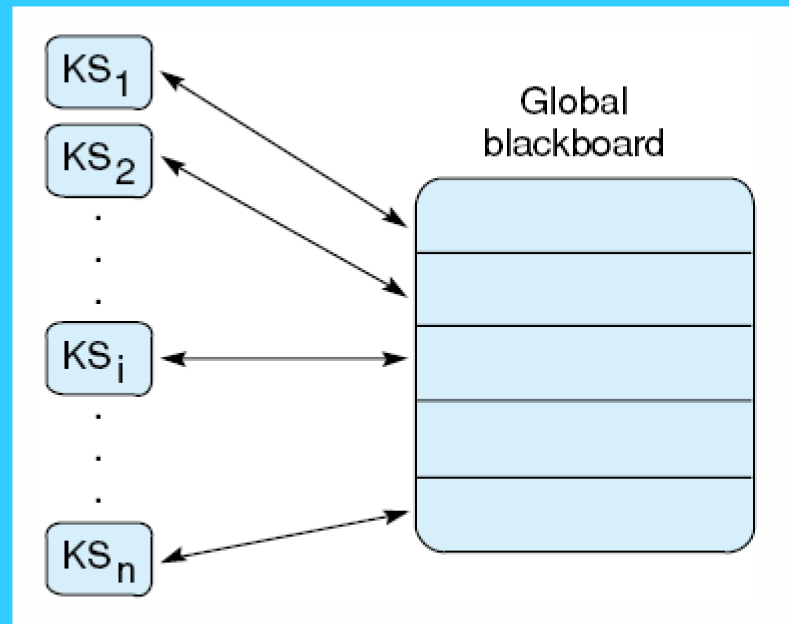
Goal tree is a kind of And-Or tree, in which nodes represent goals and branches indicate how you can achieve goals by solving one or more subgoals. Each node's children correspond to immediate subgoals; each node's parant corresponds to the immediate supergoal. Some goals are satisfied directly, they are called leaf goals.

Goal trees may be used to answer how and why questions.

Planning

- Planning means deciding on a course of actions before acting.
- Failure to plan can result in less than optimal problem solving.
- Common-used approaches to planning:
  - Nonhierarchical planning
  - Hierarchical planning
  - Script-based planning
  - Opportunistic planning

Fig 6.13 Blackboard architecture



9

Reasoning in Uncertain Situations

- Uncertainty results from:
- the use of abductive inference,
- attempts to reason with missing or unreliable data.

Some ways of managing the uncertainty

1. Bayesian probability theory
2. Stanford certainty theory
3. Zadeh's fuzzy set theory
4. nonmonotonic reasoning