Artificial Intelligence Lecture Notes

Definitions

AI is the study of symbol systems for the purpose of understanding and implementing intelligent search.

Also AI can be defined as "Design computer that thinks as people do"

"Weak AI"
- **Machines can possibly act intelligently.**

"Strong AI"
- **Machines can actually think intelligently.**

Most AI researchers take the weak hypothesis for granted, and don't care about the strong AI hypothesis.

A.I Principles:
1- The data structures used in knowledge representation.
2- the algorithms needed to apply that knowledge.
3- the language and programming techniques used their implementation.

A.I Application:
1- Game application.
2-Automated reasoning and theorem proving.
3-Perception.
4-Expert Systems
5- Natural language understanding and semantic modeling.
6- Planning and robotics
machine learning
language & environment for A.I
Pattern recognition
A.I & Philosophy.

A.I Branches:
1- Logical A.I
2-Search
3- Representation
4-Inference
5-knowledge & Reasoning
6- Planning
7- Epistemology.
8- Ontology.
9- Heuristics.
10-Genetig Programming.

Turing Test:

• **The test is conducted with two people and a machine.**
• **One person plays the role of an interrogator and is in a separate room from the machine and the other person.**
• **The interrogator only knows the person and machine as A and B. The interrogator does not know which is the person and which is the machine.**
• **Using a teletype, the interrogator, can ask A and B any question he/she wishes. The aim of the interrogator is to determine which is the person and which is the machine.**
• **The aim of the machine is to fool the interrogator into thinking that it is a person.**
• **If the machine succeeds then we can conclude that machines can think.**
• **Often "forget" the second person.**
• **Informally, the test is whether the "machine" behaves like it is intelligent.**
• **This is a test of behaviour.**
• **It is does not ask "does the machine really think?".**
• **It is too culturally specific?**
• **If B had never heard of "The X-Factor" then does it preclude intelligence?**
• **What if B only speaks Italian?**
• **It tests only behaviour not real intelligence?**

Agent:

An agent is anything that can be viewed as
– perceiving its **environment** through **sensors** and
– **acting** upon that environment through **actuators**
– Assumption: Every agent can perceive its own actions (but not always the effects)
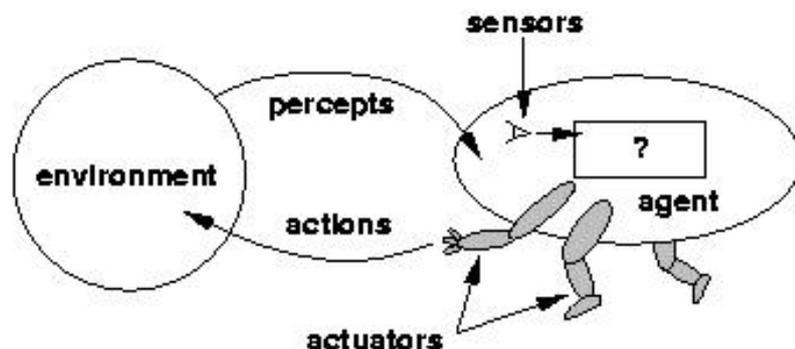
Agents

• **Human agent:**
– eyes, ears, and other organs for sensors;
– hands,legs, mouth, and other body parts for actuators

• **Robotic agent:**
– cameras and infrared range finders for sensors;
– various motors for actuators

• **A software agent:**
– Keystrokes, file contents, received network packages as sensors
– Displays on the screen, files, sent network packets as actuators

Agents and environments

• Percept: agent's perceptual input at any given instant
• Percept sequence: complete history of everything the agent has ever perceived
• An agent's choice of action at any given instant can depend on the entire percept sequence observed to date
• An agent's behavior is described by the **agent function** which maps from percept histories to actions:**[f: P* → A]**
• Internally, the agent function will be implemented by an **agent program** which runs on the physical architecture to produce f
• **agent = architecture + program**

Rational agents

For each possible percept sequence, a rational agent should select an action that is expected to maximize its performance measure, given the evidence provided by the percept sequence and whatever built-in knowledge the agent has.
• An agent should strive to "**do the right thing**", based on what it can perceive and the actions it can perform.
• The right action is the one that will cause the agent to be most successful
• Performance measure: An objective criterion for success of an agent's behavior

• **PEAS** specifies the setting of an intelligent agent:
- **P**: The **p**erformance measure defines degree of success.
- **E**: What does the agent know about the **e**nvironment?
- **A**: The **a**ctions that the agent can perform.
- **S**: Everything that an agent as perceived so far through its **s**ensors.

• For Example: **Automated Taxi Driver**
  - <u>Performance measure</u>: Safe, fast, legal, comfortable trip, maximize profits.
  - <u>Environment</u>: Roads, other traffic, pedestrians, customers.
  - <u>Actuators</u>: Steering wheel, accelerator, brake, signal horn.
  - <u>Sensors</u>: Cameras, sonar, speedometer, GPS, odometer, engine sensors, keyboard.

• For Example: **Medical Diagnosis System**
  - <u>Performance measure</u>: Healthy patient, minimize costs, minimize lawsuits.
  - <u>Environment</u>: Patient, hospital, staff.
  - <u>Actuators</u>: Screen display (questions, tests, diagnoses, treatments, referrals.
  - <u>Sensors</u>: Keyboard (… MISSING)

Environment types

• **Fully Observable** (vs. partially observable): An agent's sensors give it access to the complete state of the environment at each point in time.
• **Deterministic** (vs. stochastic): The next state of the environment is completely determined by the current state and the action executed by the agent.
  - Strategic: the environment is deterministic except for the actions of other agents.

• **Episodic** (vs. sequential): The agent's experience is divided into atomic "episodes" (each episode consists of the agent perceiving and then performing a single action), and the choice of action in each episode depends only on the episode itself.
• **Static** (vs. dynamic): The environment is unchanged while an agent is deliberating.
  - Semi-dynamic: the environment itself does not change with the passage of time, but the agent's performance score does.
• **Discrete** (vs. continuous): A limited number of distinct, clearly defined percepts and actions.
• **Single agent** (vs. multi agent): An agent operating alone in an environment. (When there are multiple agents, agents can work co-operatively or competitively.)

• **Difficulty of agent design:**
  - *Simplest*: fully observable, deterministic, episodic, static, discrete and single agent.
  - *Hardest*: partially observable, stochastic, sequential, dynamic, continuous and multi-agent.
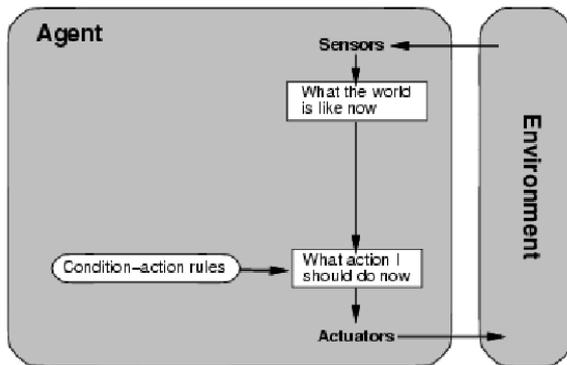
Agent types

Table-based Look-up Agent
• Keep all percept sequences in memory and associate them with the appropriate actions.
• Drawbacks:
- Huge table (e.g., 35 100 entries for chess)
- Take a long time to build the table
- No autonomy
- Even with learning, need a long time to learn the table entries
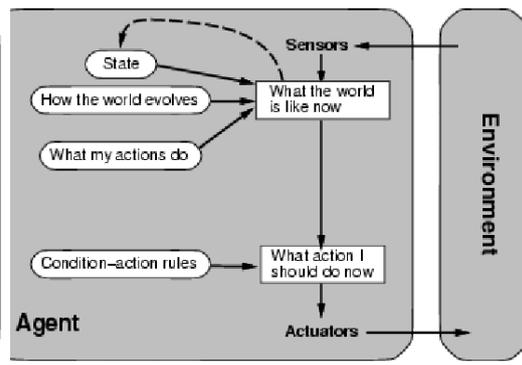• Easy to implement, but not realistic. Store and Retrieve.

Simple Reflex Agent   Model-based Reflex Agent
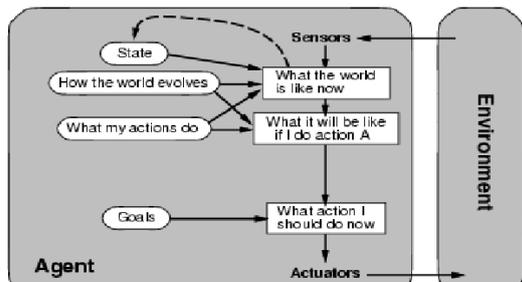• No memory of the past.   • A model of the world to conjecture about what may happen
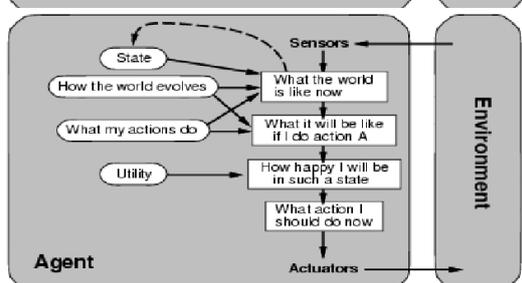

Goal-based Agent
• With a range of actions available, we favour actions which help us arrive at the goal to find the solution efficiently.
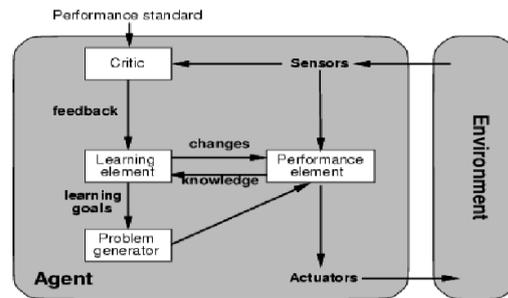

Utility-based Agent
• Focus on the performance measure, the optimal solution – not just any solution.

Learning Agent

* <u>Search Techniques</u>
There are two types of searches, these are:
1- Blind search
- Depth First Search.
- Breadth First Search.
- Hybrid Search (Iterative deepening)

2- Heuristic search
- Hill Climbing.
- Best First Search.
- A* algorithm.

## * <u>Structures and Strategies for State Space Search</u>
To successfully design and implement search algorithms, a programmer must be able to analyze and predict their behavior. Questions that need to be answered include:

- Is the problem solver guaranteed to find a solution?
- Will the problem solver always terminate, or can it become caught in an infinite loop?
- When a solution is found, it is guaranteed to be optimal?
- What is the complexity of the search process in terms of time usage? Space usage?
- How can the interpreter most effectively reduce search complexity?
- How can an interpreter be designed to most effectively utilize a representation language?

The theory of **state space search** is the primary tool for answering these questions. By representing a problem as a **state space** graph, the graph theory can be used to analyze the structure and complexity of both the problem and the procedures used to solve it.
If a graph is used, the problem of cycles will occur so the best structure to represent a problem is a **tree structure** (there is a unique path between every two nodes)**,** which can be defined as a tree with no cycles. It is important to distinguish between problems whose state space is a tree and those that may contain loops (graph). General graph search algorithms must detect and eliminate loops from potential solution paths, while tree searches may gain efficiency by eliminating this test and its overhead. So convert graph representation to a tree representation. To convert a graph to a tree, remove any nodes that may cause a cycle.

Example: convert a graph to a tree
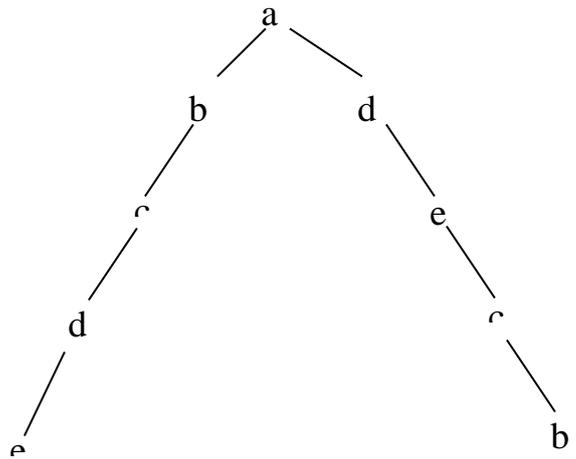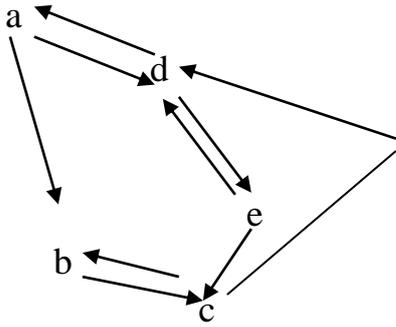
* **Depth First Search Algorithm**

Initialize: open=[start]; closed=[ ]; parent[start]=null; found=no;
While open<>[ ] do
   Begin
- Remove the first state from left of open, call it X;
- If X is a goal then {found=yes; break}
- Generate all possible children of x and put them in list L;
- Put X on closed;
- Eliminate from L any child already on closed;
- Eliminate from open any child in L;
- For each child y in L set parent[y]=X;
- Add L to the left of open;

  End.
 If found =yes then
   Generate and return the solution path.
 Else
   Output no solution.

Example: use depth first search to find the path between A and L for the following search space:
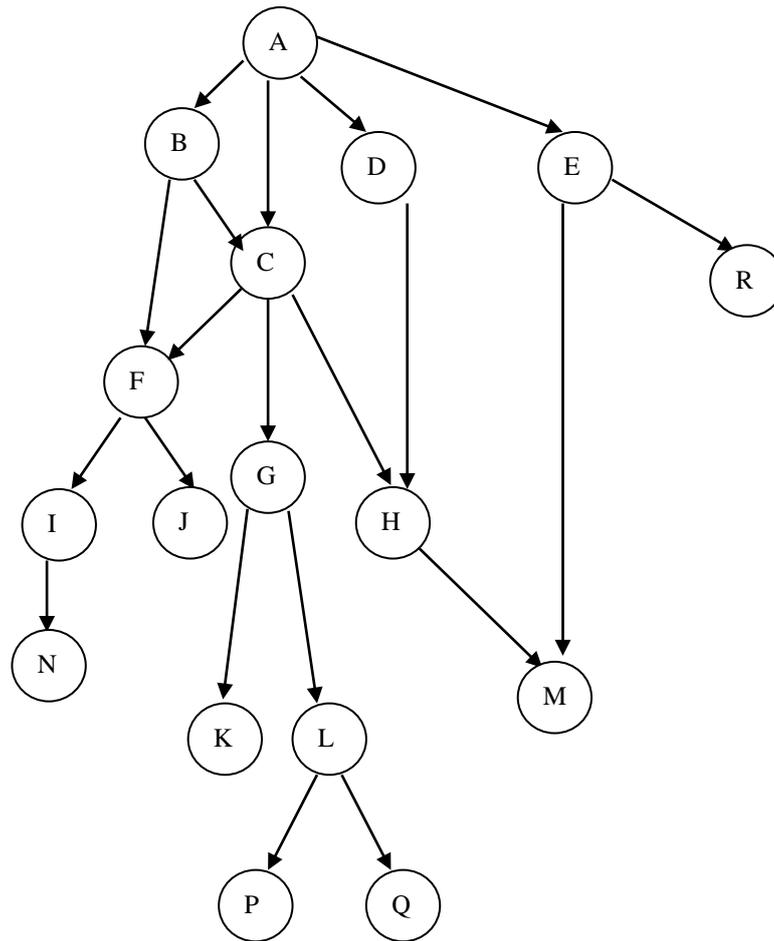
Solution:
Start=[A], goal=L.

Iteration #0:
Open=[A], closed=[ ], parent[A]=null.

Iteration #1:
X=A, L=[B,C,D,E], open=[B,C,D,E], closed=[A].
Parent[B]=A, parent[C]=A, parent[D]=A, parent[E]=A.

Iteration #2:
X=B, L=[F,C], open=[F,C,D,E], closed=[A,B].
Parent[F]=B, parent[C]=B.

Iteration #3:
X=F, L=[I,J], open=[I,J,C,D,E], closed=[A,B,F].
Parent[I]=F, parent[J]=F.

Iteration #4:
X=I, L=[N], open=[N,J,C,D,E], closed=[A,B,F,I].
Parent[N]=I.

Iteration #5:
X=N, L=[ ], open=[J,C,D,E], closed=[A,B,F,I,N].

Iteration #6:
X=J, L=[ ], open=[C,D,E], closed=[A,B,F,I,N,J].

Iteration #7:
X=C, L=[F,G,H], open=[G,H,D,E], closed=[A,B,F,I,N,J,C].
Parent[G]=C, parent[H]=C.

Iteration #8:
X=G, L=[K,L], open=[K,L,H,D,E], closed=[A,B,F,I,N,J,C,G].
Parent[K]=G, parent[L]=G.

Iteration #9:
X=K, L=[ ], open=[L,H,D,E], closed=[A,B,F,I,N,J,C,G,K].

Iteration #10:
X=L
Since L is a goal, stop and find path.

Path: A ⟶ B ⟶ C ⟶ G ⟶ L

* __Breadth First Search Algorithm__

Initialize: open=[start]; closed=[ ]; parent[start]=null; found=no;
While open<>[ ] do
   Begin
-   Remove the first state from left of open, call it X;
-   If X is a goal then {found=yes; break}
-   Generate all possible children of x and put them in list L;
-   Put X on closed;
-   Eliminate from L any child already on closed;
-   Eliminate from L any child in open;
-   For each child y in L set parent[y]=X;
-   Add L to the right of open;
  End.
 If found =yes then
   Generate and return the solution path.
 Else
   Output no solution.

Example: use breadth first search to find the path between A and L for the search space in the previous example:

Solution:
Start=[A], goal=L.

Iteration #0:
Open=[A], closed=[ ], parent[A]=null.

Iteration #1:
X=A, L=[B,C,D,E], open=[B,C,D,E], closed=[A].
Parent[B]=A, parent[C]=A, parent[D]=A, parent[E]=A.

Iteration #2:
X=B, L=[F,C̶], open=[C,D,E,F], closed=[A,B].
Parent[F]=B.

Iteration #3:
X=C, L=[F̶,G,H], open=[D,E,G,H], closed=[A,B,C].
Parent[G]=C, parent[H]=C.

Iteration #4:
X=D, L=[H̶], open=[E,G,H], closed=[A,B,C,D].

Iteration #5:
X=E, L=[M,R], open=[G,H,M,R], closed=[A,B,C,D,E].
Parent[M]=E, parent[R]=E.

Iteration #6:
X=G, L=[K,L], open=[H,M,R,K,L], closed=[A,B,C,D,E,G].
Parent[K]=G, parent[L]=G.

Iteration #7:
X=H, L=[M̶], open=[M,R,K,L], closed=[A,B,C,D,E,G,H].

Iteration #8:
X=M, L=[ ], open=[R,K,L], closed=[A,B,C,D,E,G,H,M].

Iteration #9:
X=R, L=[ ], open=[K,L], closed=[A,B,C,D,E,G,H,M,R].

Iteration #10:
X=K, L=[ ], open=[L], closed=[A,B,C,D,E,G,H,M,R,K].

Iteration #11:
X=L
Since L is a goal, stop and find path.

Path: A ⟶ C ⟶ G ⟶ L

* <u>Hybrid Search Algorithm</u>

Apply depth first iteratively increasing the level of the state space by one until the goal is found or the whole space is searched with no success.

When you reapply depth first, you start from the start state and ignore any information from previous runs.

* <u>Heuristics</u>

May be defined as the study of methods and rules of discovery and invention.

In state space search heuristics are formalized as rules for choosing those branches in the state space that are most likely to lead to an acceptable solution.

AI problem solvers employ heuristic in two basic situations:

1- A problem may not have an <u>exact solution</u> because of inherent ambiguities in the problem statement or available data.

<u>Example:</u>
In vision, vision scenes are often ambiguous allowing multiple interpretations of objects. Heuristics are used to select the most likely interpretation.

2- A problem may have an exact solution. But the <u>computational cost</u> of finding it may be prohibitive.

<u>Example:</u> bark of the cars.

A heuristics can lead a search algorithm to a sub optimal solution or fail to find any solution at all.

* <u>Heuristic Algorithms:</u>

It is useful to think of heuristic algorithms as consisting of two parts:
1- A heuristic measure function (is a measure to determine <u>which path is the best</u>).
2- An algorithm that uses that the measure to search the state space.

* <u>Heuristic Functions:</u>

There are heuristics of a very general applicability and ones that represent specific knowledge that is relevent to the solution of a particular problem. One example of a good general purpose heuristic is the algorithm (nearest neighbor algorithm), which works by selecting the locally superior alternative at each step.

General purpose heuristics may be coupled with some special purpose heuristics so as to work well for the specific domain.

A Heuristic function is a function that maps from a problem state description to measure of desirability usually represented as numbers.

Sometimes a high value of the heuristic function indicates a relatively good position while at other times, a low value indicates an advantageous situation.

The purpose of the heuristics functions is to guide the search process in the most profitable direction by suggesting, which path to follow first when more than one path is available.

In general there is a trade off between the cost of evaluating a heuristic function and the savings in search time that the function provides.

However, the most accurately the heuristic function estimated the true merits of each node in the tree or graph, the more direct will be the solution process.

Examples:
1- 8-puzzle, the number of tiles that are in the place they belong to.
2- traveling salesman, the sum of the distances.

* **Search Algorithms**

The strategy used for controlling a search is often critical in determining how effective the heuristics functions; here are some general-purpose control strategies:
1- Generate and test.
2- Hill climbing.
3- Best first search.
4- A* .

Romania Map Problem

Straight−line distance to Bucharest	
Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	178
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	98
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374
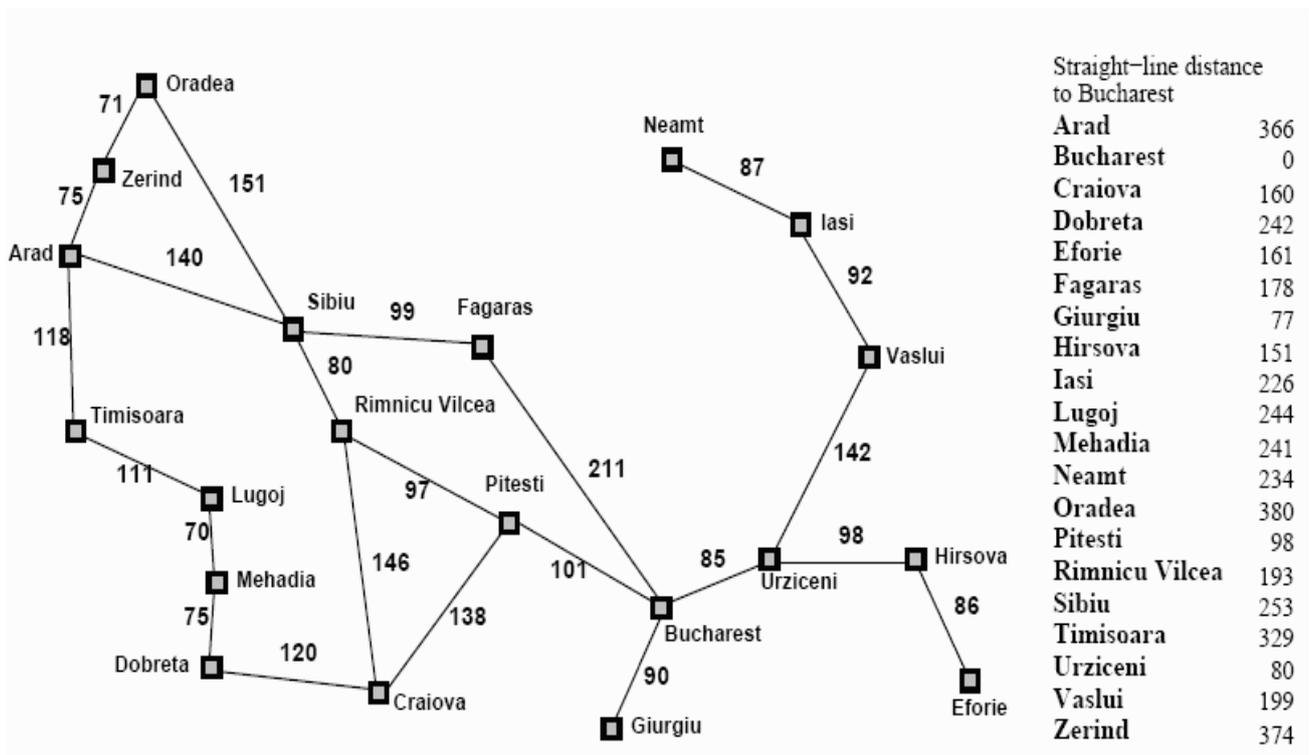
* __Hill Climbing:__

The strategy consists of the following steps:

1- Generate a possible solution for some problems, (this means generating particular point in the problem space. For others, it means generating a path from a start state). See if it is a solution if so quit, else continue.

2- From this solution apply a number of applicable rules to generate a new set of proposed solutions.

3- For each statement of the set, do the following:-
   - a- Send it to the test function, if it is a solution, quit.
   - b- If not, see if it is closest to a solution of any of the elements tested so far. If it is, remember it, if it is not, forget it.

4- Take the best element found above and uses it as the next proposed solution.

5- Go back to step 2.

To see how hill climbing works consider the following problem:

Four cubes each of those sides is painted by one of 4 colors. A solution to the puzzle consists of an arrangement of the cubes in a row such that on all four sides of the row, one block face of each color is showing.

To solve the problem, we first need to define a heuristic function that describes how close a particular configuration is to being a solution. One such solution is simply the sum of the number of different colors on each of the four sides. Next we need to define a set of rules that describe ways of transforming one configuration into another. One rule will suffice. It says simply pick a block and rotate it at 90º in any direction.

The next step is to generate a starting state this can be done at random. Now hill climbing can begin. We could try all possible rotations of all four blocks and see which leads to the greatest improvement. Another strategy may be to try some of the possible moves. This can be done by picking one block and see if there is any way to rotate it to improve the situations. If there is, perform that rotation and continue. But what if more of the possible rotations produce desirable state.

* __Hill Climbing Algorithm:__

Begin
  CS=start, found=false, Path=[]
  While (not found) do
      Begin
          Add CS to Path
          If CS=goal then return (path)
          Else generate all child states of CS
              Remove any child states of CS already on Path
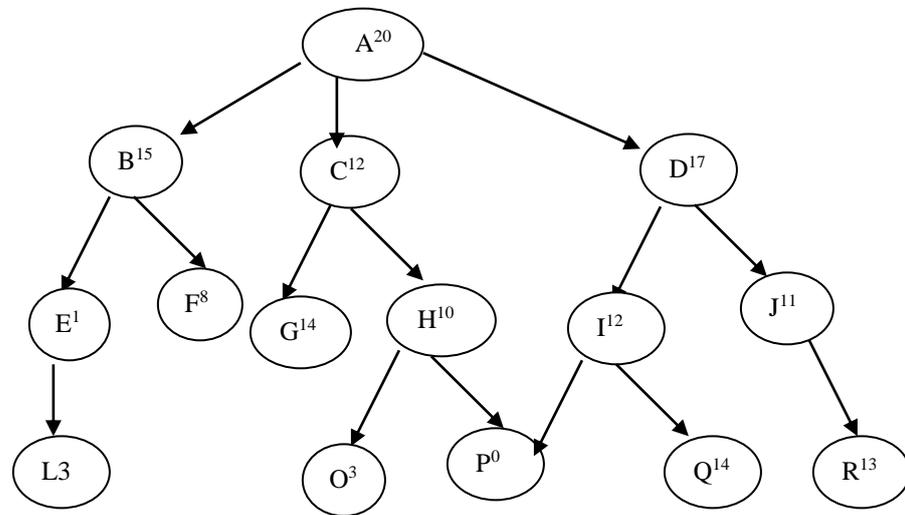   If CS has no remaining child states then return (fail)
              Else compute the heuristic value of all remaining child states

Choose a child state with the <u>best</u> heuristic value call it B
If B is not better than CS then return (fail)
Else CS=B
    End
End

Example: use hill climbing search to find the path between A and P for the following search space:

Solution:

Iteration	CS	Path
0	A	[]
1	C	[A]
2	H	[A,C]
3	P	[A,C,H]
4		[A,C,H,P]

With hill climbing can arrive at the following situations:

1- a local maximum is a state that is better than all its neighbors but is not better than other states further away. At local maximum, all moves appear to made things worse.
2- A plateau is a flat area of the search space in which a whole set of neighboring states have the same value on a plateau. it is not possible to determine the best direction in which to move by making local comparisons.
3- A ridge is an area of the search space that is higher than surrounding areas but that cannot be <u>traversed</u> by single moves in any one direction. There are some ways of dealing with these problems. Although the methods are by no means guaranteed.

The solution:

1- Backtrack to some earlier node and try going in a different direction. This is reasonable if at that node there was another direction that looked as promising or almost as promising as one that was chosen. To adopt this strategy, maintain a list of paths almost taken and go back to one of them, if the path that was taken leads to a dead end.
2- Make a big jump in one direction to try to get to a new section of the search space. This strategy may be used with plateau. If the only rules available describe single small steps apply them several times in the same direction.
3- Apply two or more rules before doing the test. This corresponds to moving in several directions at once. This is a good strategy for dealing with ridges.

* **Best First Search (BFS):**
In general, heuristic search requires a more informed algorithm this is provided by BFS.

BFS uses lists to maintain states:

1- Open: to keep track of the current scope of the search.
2- Closed: to record states already visited.

An added step in the algorithm orders the states on open according to some heuristic estimated of their closeness to the goal. Thus each iteration of the loop considers the most promising state on the open list:

The algorithm for BFS:

Initialize: open=[start], closed=[], parent[s]=null.
     While open <> [] do
       Begin
         Remove the next state form open, call it X.
         If X is a goal then return the solution path that led to X.
         Else
           Generate all possible children of X and put them in list L.
           For each child Y of X do
             Case
               The child Y is not on open or closed:
                 Begin
                   Assign the child Y a heuristic value.
                   Add the child Y to open.
                   Set parent[Y]=X.
                 End;
               The child is already on open:
                 Begin
                   If the child was reached alone a shorter path than the state currently on open, then give the state on open this shorter value.
                 End;
               The child already on closed:

Begin
    If the child was reached alone a shorter path than the
    state currently on closed, then give the state on closed
    this shorter value and move this state from closed to
    open.
End;
    End; {case}
    Put X on closed
    Reorder states on open according to heuristic value (best values first)
End; {while}
Return (failure)
End.

Example: use best first search to find the path between A and P for the following
search space:
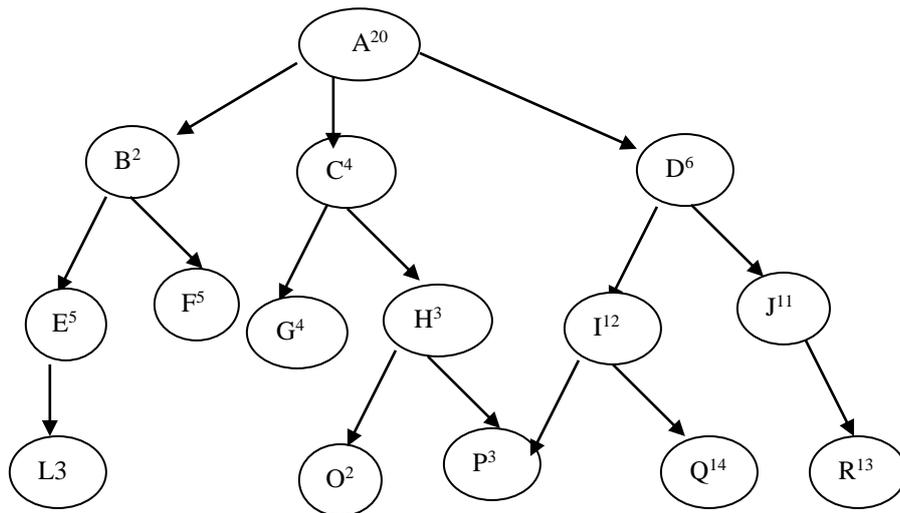
Solution:
Start=[A], goal=P.

Iteration #0:
open=[A], closed=[ ], parent[A]=null.

Iteration #1:
X=A, L=[B,C,D], open=[B^2,C^4, ,D^6], closed=[A].
parent[B]=A, parent[C]=A, parent[D]=A.

Iteration #2:
X=B, L=[E,F], open=[C^4,E^5,F^5,D^6], closed=[A,B].
parent[E]=B, parent[F]=B.

Iteration #3:
X=C, L=[G,H], open=[H^3,G^4,E^5,F^5,D^6], closed=[A,B,C].
parent[G]=C, parent[H]=C.

Iteration #4:

X=H, L=[O,P], open=[O^2,P^3,G^4,E^5,F^5,D^6], closed=[A,B,C,H].
parent[O]=H, parent[P]=H.

Iteration #5:
X=O, L=[], open=[P^3,G^4,E^5,F^5,D^6], closed=[A,B,C,H,O].

Iteration #6:
X=P=goal then stop

Path: A $\longrightarrow$ C $\longrightarrow$ H $\longrightarrow$P

* __Implementing heuristic evaluation functions:__
Since heuristics are fallible, it is possible that a search algorithm can be misled down some path that fails to lead to a goal. To overcome this problem, if two states have the same or nearly the same heuristic evaluations, it is generally preferable to examine the state that is nearest to the root of the graph.

This state will have greater probability of being on the shortest path to the goal. The distance from the starting state to its descendants can be measured by maintaining a depth count for each state. This count is (0) for the beginning state and is increment by (1) for each level of the search. It records the actual number of moves that have been used to go from the starting state to each descendant. This can be added to the heuristic evaluation of each state to bias search in favor of states found shallower in the graph.

This makes the evaluation function the sum of two components:
F(n)=h(n) + g(n)   where:
- g(n) measures the actual length of the path from any state n to the start state and
- h(n) is the heuristic measure at state n.

* __Behavior of heuristic search:__
In some problems, the objective is not only to find the solution, but many require the algorithm to find the shortest path to the goal. This can be important when an application might have excessive cost for extra solution steps. __Heuristics that find the shortest path to a goal whenever it exists are said to be admissible__ (definition of admissibility). In other applications a minimal path might not be as important as the overall problem efficiency.

In what sense is one heuristic better than another? This is the informedness of heuristic.

When a state is discovered using heuristic search is there any guarantee that the same state won't be found later in the search at a cheaper cost (with a shorter path from the start state?) this is property of <u>monotoncity</u>.

* **Admissibility measures:**

An algorithm is admissible if it is guaranteed to find a minimal path to a solution whenever such a path exists.

Breadth first search is an admissible search strategy, since it looks at every state at level n of the graph before considering any state at level n+1 so any goal nodes are found along the shortest possible path, however, it is often too inefficient for practical use.

* **Definitions:**

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 goal.

If this evaluation function is used with the BFS algorithm, the result is called algorithm A.

Define the function f*(n)=g*(n)+h*(n), where:

- g*n) is the cost of the shortest path from the start node to node n.
- h*(n) is the actual cost of the shortest path from n to the goal.

It follows that f*(n) is the actual cost of the optimal path from a start node to a goal node that passes through n.

If BFS is used with the evaluation function f*, the resulting search strategy is admissible. For most real problems, f* does not exist.

In algorithm A, g(n) is a reasonable estimate of g* but they may not be equal. g(n)>=g*(n). These are equal if the graph has discovered the optimal path to state n.

Similarly if we replace h*(n) with h(n) which is a heuristic estimate, if algorithm A uses an evaluation function f in which h(n)<=h*(n) the resulted algorithm will be called A*.

* **Algorithm A:**

Begin
   Input the start node s and the goal node.
   Open=[s]; closed=[], g[s]=0; pred[s]=null.
   While open<>[] do
     Begin
       Remove the first element form open, call it X.
       If X is a goal then return the solution path that led to X.
      Else
        Generate all possible children of X and put them in list L.
        For each child Y of X do
          Case
            The child Y is not on open or closed:
             Begin
               g[Y]=g[X]+cost(X,Y).

$$f[Y]=g[Y]+h[Y].$$
Set pred[Y]=X.
Add the child Y to open.
End;
Else
The child Y in open or in close:
Begin
temp=f[Y]-g[Y]+g[X]+cost(X,Y).
If temp<f[Y] then
Begin
g[Y]=g[X]+cost(X,Y).
f[Y]=temp.
pred[Y]=X.
If Y is on close then insert Yin open and remove it from close
End;
End;
End; {case}
Put X on closed
Reorder states on open according to heuristic value (best values first)
End; {while}
Return (failure)
End.

Example: use A algorithm to find the path between A and K for the following search space
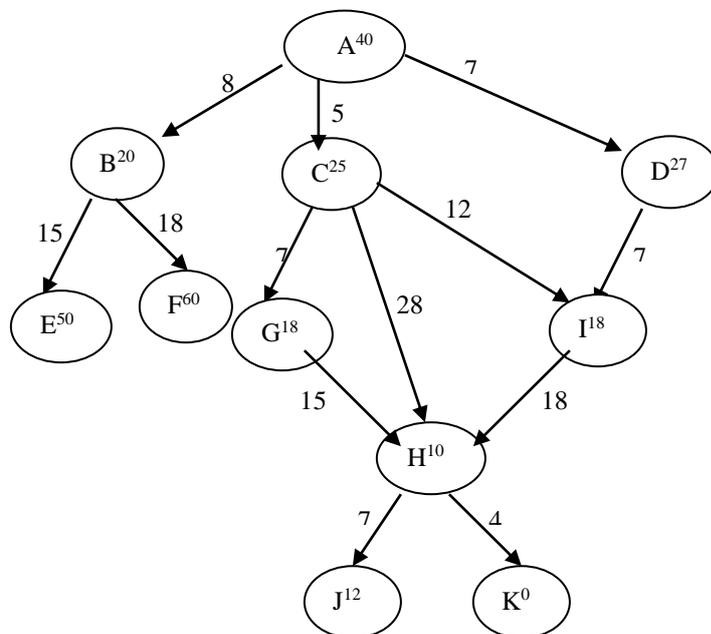
Solution:
   Start=[A], goal=K.

 Iteration #0:
Open=[A], closed=[ ], G[A]=0, pred[A]=null.

Iteration #1:
X=A, L=[B,C,D], G[B]=8, F[B]=28, pred[B]=A, G[C]=5 F[C]=30, pred[C]=A, G[D]=7, F[D]=34, pred[D]=A, closed=[A], open=[B28,C30,D34].

Iteration #2:
X=B, L=[E,F], G[E]=23, F[E]=73, pred[E}=B, G[F]=26, F[F]=86, pred[F]=B, closed=[A,B], open=[C30,D34,E73,F86].

Iteration #3:
X=C, L=[G,H,I], G[G]=12, F[G]=30, pred[G]=C, G[H]=33, F[H]=43, pred[H]=C, G[I]=17, F[I]=35, pred[I]=C, closed=[A,B,C], open=[G30,D34,I35,H43,E73,F86].

Iteration #4:
X=G, L=[H], temp=37, G[H]=27, F[H]=37, pred[H]=G, closed=[A,B,C,G], open=[D34,I35,H37,E73,F86].

Iteration #5:
X=D, L=[I], temp=32, G[I]=14, F[I]=32, pred[I]=D, closed=[A,B,C,G,D], open=[I32,H37,E73,F86].

Iteration #6:
X=I, L=[H], temp=42, closed=[A,B,C,G,D,I], open=[H37,E73,F86].

Iteration #7:
X=H, L=[J,K], G[J]=34, F[J]=46, pred[J]=H, G[K]=31, F[K]=31, pred[K]=H, closed=[A,B,C,G,D,I,H], open=[K31,J46].

Iteration #8:
X=K
Since K is a goal, stop and find path.

Path: A $\longrightarrow$ C $\longrightarrow$ G $\rightarrow$ H $\longrightarrow$ K

        All A* algorithms are admissable: This theorm says that any A* algorithm (i.e that uses a heuristic h(n) such that h(n)≤h*(n) for all n), is guaranteed to find the minimal path from n to the goal, if such a path exists, Breadth first search may be characterized as an A* algorithm in which f(n)= g(n)+0. (i.e h(n) =zero).

        In the 8-puzzle, the heuristic h(n) used of counting the number of tiles out of place is counting less than or equal to the actual number of moves required to move them to their goal position. Hence this heuristic is admissable and guarantees the optimal or shortest path solution**.**

* <u>Monotonucity</u>

The definition of the A* algorithm did not require that g(n)=g*(n), this means that admissable heuristics may initially reach non-goal states along a suboptimal path.

As long as the algorithm eventually finds an optimal path to all states on the path to the goal. A heuristic function h is monotonic if:-

1- for all states n_i and n_j, where n_j is a descendant of n_i:-
    $h(n_i) - h(n_j) \leq cost(n_i,n_j)$, where cost (n_i,n_j) is the actual cost (in number of moves) of going from state n_i to state n_j.

2- The heuristic evaluation of the goal state is zero , h(goal)=0

This is to describe the monotone property that is to say that the heuristic is everywhere admissable reaching each state along the shortest path from its ancestors. If the graph search algorithm for BFS is used with a monotonic heuristic, an important step may be omitted, since the heuristic finds the shortest path to any state the first time it is discovered, when a state is rediscoverd, it is not necessary to check if the new path is shorter.

Any monotonic heuristic is admissible. This can be shown as follows:-

If we consider a path in the space as a sequence of states $S_1,S_2,…,S_g$, where S_1 is the start state and S_g is the goal state. Then :-

S_1 to S_2 $h(S_1)-h(S_2) \leq cost(S_1,S_2)$
S_2 to S_3 $h(S_2)-h(S_3) \leq cost(S_2,S_3)$
S_3 to S_4 $h(S_3)-h(S_4) \leq cost(S_3,S_4)$
.
.
.
S_{g-1} to S_g $h(S_{g-1})-h(S_g) \leq cost(S_{g-1},S_g)$

Summing up each column and using the monotone property $h(S_g)=0$, path from S_1 to S_g, $h(S_1) \leq cost(S_1,S_g)$.

This means that monotone heuristic h is A* and admissable.

*<u>Informedness</u>

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

In the 8-puzzle, Breadth first search is equivalent to the A* algorithm with heuristic h, such that $h_1(x)=0$, for all states x.

This is trivially less than h*. Also h_2, the number of tiles out of place with respect to the goal state is alower bound for h*. In this case:-
$$h_1 \leq h_2 \leq h*$$

It follows that the number of tiles out of place heuristic is more informed than Breadth first search, both h_1 and h_2 find the optimal path, but h_2 evalutes many fewer states.

Similary we can argue that the heuristic that calculates the sum of direct distances by which all the tiles are out of place is gain more informed than that of the number of tiles out of place.

This can visualize a sequence of search spaces, each smaller than the previous one. Converging on the optimal path solution.

If aheuristic h_2 is more informed than h_1 then the set of states examined by h_2 is a subset of those expanded by h_1 , however one must be careful that the computations necessary to employ the more informed heuristic are not so inefficient as the offset the gains from reducing the number of states searched.

Search in games
*The minimax procedure:-

Games are an important application area for heuristic algorithms. Two person games   are   more   complicated   than   simple   puzzles (i.e. the existance of unpredictable opponent).

ex. nim game
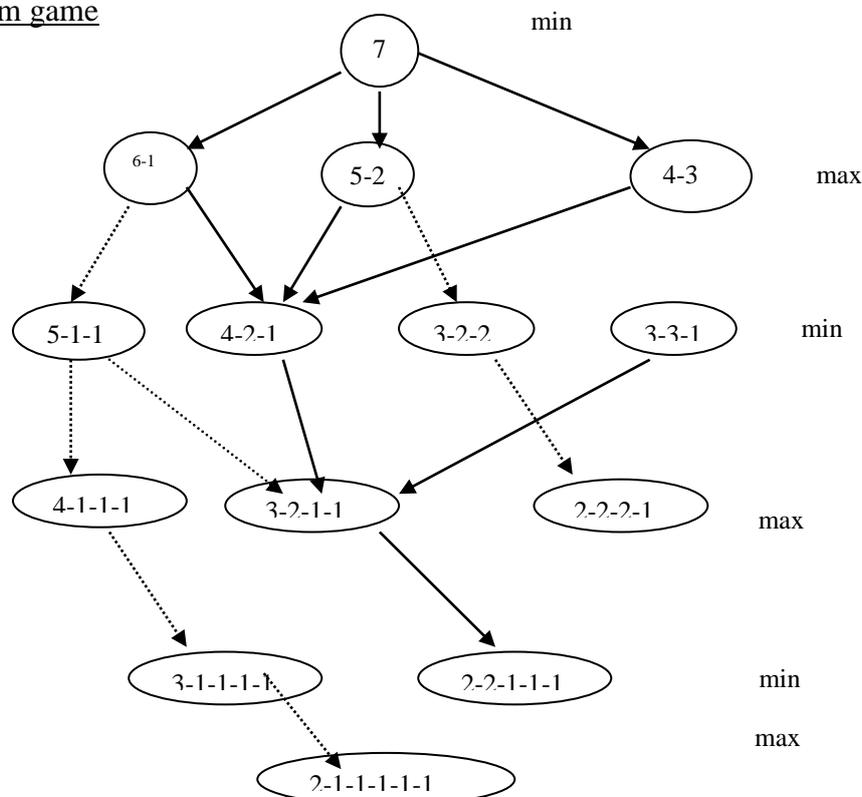
To play nim, a number of matches are placed on a table. At each move, the player must divide a pile of matches into two non-empty piles with different number of matches in each pile. The first player who can no longer make a move losses the game.

Each level in the search space is labelled according to whose move it is at the point in the game, (i.e min or max). Each leaf node is given a value of (0) or (1) depending on wether it is a win for max or for min. Minimax propagates these values up the graph through successive parent nodes according to the rule:-

- If the parent state is a max node, give it the maximum value among its children.
- If the parent state is a min node, give it the minimum value among its children.

The value that is assigned to each state indicates the value of the best state that player can hope to achieve.

The values of the leaf nodes are propagated up the graph using minimax.

Minimax search procedure is a <u>depth first</u>, depth limited search procedure, in applying minimax to more complicated games, it is seldom possible to expand the state space graph out to the leaf nodes, instead the state space is searched to a predefined number of levels as determined by available resources of time and money.

This strategy is called an **n-move look ahead**, where n is the number of levels explored, since the leaves of this subgraph are not final states of the game, it is not possible to give them values that reflect a win or a loss instead each node is given a value according to some heuristic evaluation function.The value that is propagated back to the root node is not an indication of whether or not a win can be achieved but is simply the heuristic value of the best state that can be reached in n moves from this start node .

Game graphs are searched by level or ply. Each move by a player define a new ply of the graph. Game-playing programs typically look ahead a fixed ply depth.

The states on that ply are measured heuristically and the values are propagated back up the graph using minimax. The search algorithm then uses these derived values to select among possible next moves .

In the example opposite, since our goal is to maximize the value of the heuristic function, we choose to move to B. Hence A's value is 8 since we can move to aposition with avalue of 8.
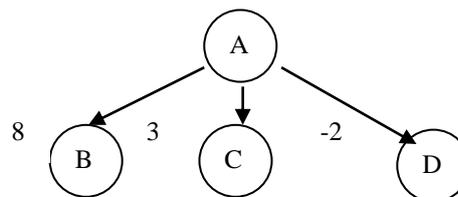
If we stop at two-ply look ahead, we have now the situation opposite. Taking into account that the opponent gets to choose which of the successor moves will be made, and this which terminal value should be backed up to the next level. Suppose we make move B. The opponent's goal is to minimize the value of the evalution function and hence is expected to choose move F. This means that we will end up in a very bad position (-6) even through there is position E (9) which is very good. Once

the values from the second ply are backed up, it is clear that the correct move for us at the first level is C. This process can be reapeted for as many ply as time allows, and the more accurate evaluations that are produced can be used to choose the correct move at the top level. This alternation of maxmizing and minimizing at alternate ply corrospondes to the opposing strategies of the two players and gives this method its name **minimax** .
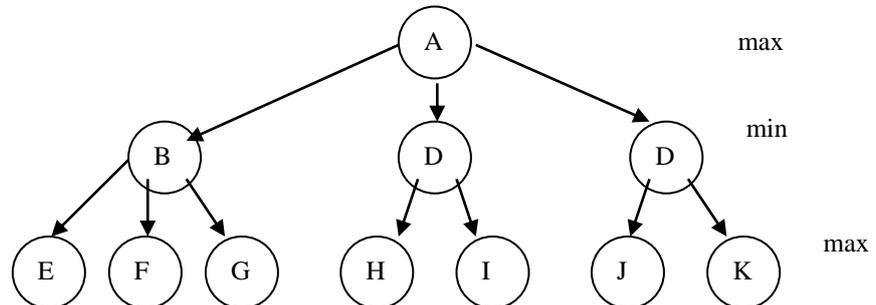
* <u>Minimax Procedure</u>

The recursive procedure for the minimax operation is assumed to return two results.
- The backed up value of the path it chooses (value).
- The path it self.

It is called to compute the best move from the current position (CURRENT) and takes two arguments: a position and current depth of the search, (e.x. minimax (CURRENT,0)).

1- If the node is on the final level explored (i.e search should be stopped at the current level) then the value of this mode is that determined by the static evaluation function and the path is nil.

2- Otherwise generate one more ply of the tree and return a list of nodes (successors) representing the moves that could be mode starting in the current node.

3- If no successors, then there are no moves to be made, then do the same as if the node is on the final level to be explored .

4- If successors, then examine each element and keep track of the best one. This is done as follows:-

5- Initilize BS (best score) to the minimum value that the static function can return it will be updated to reflect the best score that can be achieved by an element of the successors.

6- For each element of the successors (to be called succ) do the following:-

a- Set result-succ to minimax (succ,depth+1). This recursive call to minimax will actually do the exploration of succ.

b- Set new value to minus value (result succ). This will cause it to reflect the merits of the position from the opposite side from that of the next lower level.

c- If new value > BS, then we have found a successor that is better than any that have benn examined so far. record this by:

   1- Set BS to new value.
   2- The best known path is now from current to succ, and then on the appropriate path from succ as determined by the recursive call to minimax, so set the best path to the result of appending succ to path. (Result succ).

After all successors have been examined, the value of node=BS

*<u>**Alpha – Beta cutoff**</u>

The minimax procedure is a depth first process, it requires a two pass analysis of the search space (the first to descend down to the ply depth and then apply the heuristic, the second to propagate values back up the tree, minimax follows all branches in the space including many that could be ignored or pruned by a more intelligent algorithm.
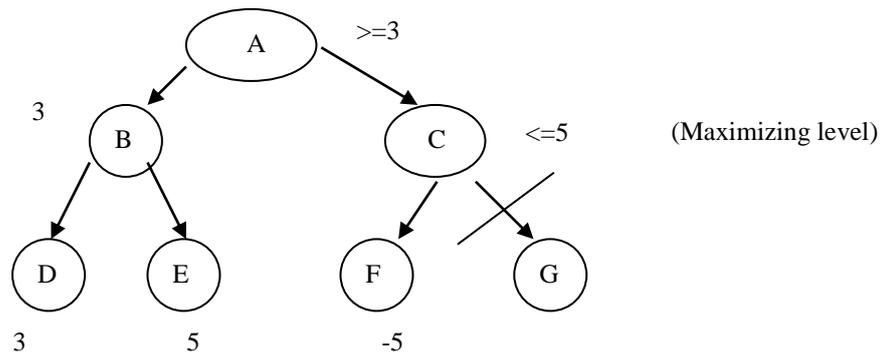
In the example above, A is guaranteed a score of (3) or greater if we move to B, any move that produces ascore of less than 3 is worse than the move B, after examining node F, we know that the opponent is guaranteed a score of (-5) or less at C, so after examining only F we are sure that a move to C is worse regaredless of the score of node G, thus we need not explore node G at all.However cutting out one node may not appear effective but the process can be cost – effective.

If we were to eliminate not a single node but an entire tree three ply deep.

24
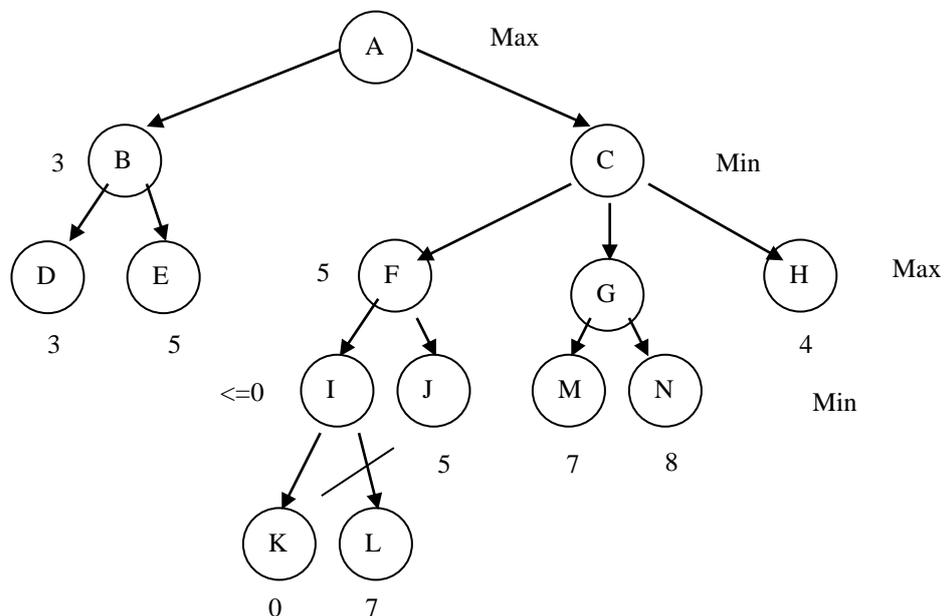
In the figure above, the entire tree headed by B is searched and hence can expect a score of at least 3. At A, when this alpha value is passed to F, it will enable us to skip the exploration of L. This is because after K is examined, I is guaranteed a maximum score of 0 (i.e F is guaranteed aminimum score of 0). But this is less than alphas value of 3, so no more branches of I need to be considerd on examining J, F is assigned a value of 5. This value becomes the value of ß at node C.

The idea of alph-beta search is simple, rather than searching the entire space, alph- beta proceeeds in a depth first fashion, first descent to full ply depth and apply the heuristic evalution function to a state and all its siblings. Suppose these are min nodes, the maximum of these min values is then backed up to the parent (a maximum node). This value is then offered to the grandparent of these mins as a potential ß cutoff.

Next the algorithm descends to other grand children and terminates exploration of their parent if any of their values is equal or larger than this beta value.

Similar procedure may be described for α-pruning:

Two rules for terminating search on alpha-beta values:
1- Search can be stopped below any min node having a beta value less than or equal to the alpha value of any of its max ancesstors.
2- Search can be stopped below any max node having an alpha value greater or equal to the beta value of any of its min node ancesstors.

Note: if the maximizing node is not at the top of the tree, one must also consider the alpha value that has passed down from a higher node. Thus at a maximizing level, α should be set to either the value it had at the next highest maximizing level or the best value found at this level whichever is greater. The corresponding statement can be made about β.

The effectiveness of the $\alpha-\beta$ procedure depends greatly on the order in which paths are examined. If the worst paths are examined first, then no cutoffs at all will occur.

The idea behind the $\alpha-\beta$ procedure can be extended to cutoff additional paths that appear to be at best only slight improvements over paths that have already been explored. The idea is to devote more time exploring other parts of the tree where there may be more gain.

*Other refinements:

These are often modifications to the minimax procedure that can also improve its performance: -

1- One of the factors that should sometimes be considering in determining when to stop going deeper in the search tree is wether or not the situation is relatively stable.

EX.1
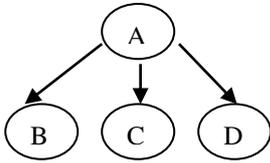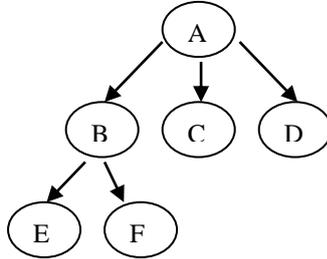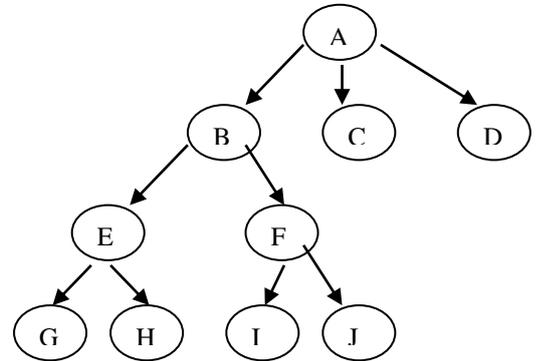
[1]   [2]   [3]

when node B in fig (1) is expanded one more level as in fig (2) the estimate of the worth of B changed greatly, if we stop exploring the tree at this level the value (-4) is assingned to B and we will therefore decide that B is not a good move, hence to make sure of the choice of move, we should continue the search until no such drastic change occures from one level to the next.

2- Secondary search. Another way that the accuracy of the minimax procedure can be improved is to double check a chosen move to make sure that there is not a hidden pitfall a few moves farther away. It is not very expensive to search the single chosen branch an additional level or two to make sure that it still looks good.

3- Using book moves

For some games like in chess, openning sequences and end game sequences are highly stylized. In these situations the performance of a program can often be considerably enhanced if it is provided with a list of moves (called book moves) that should be made. The use of book moves in the openning end game sequences combined with the use of minimax procedure for the midgame provides a good example of the way that knowledge and search can be combined to produce more effective results.

26

* **Logic**
   - Propositional Logic
   - Predicate Logic
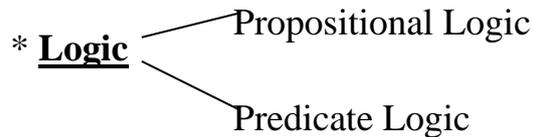
* **Propositional Logic**

"Water is liquid"   true.
"Today is Monday"   true.
"It is raining now"   true.

* **Connectives**

AND	$\wedge$
OR	$\vee$
NOT	$\neg$
IMPLIES	$\Rightarrow$
EQUAL	$=$

P	Q	$P \wedge Q$	$P \vee Q$	$\neg P$	$P \Rightarrow Q$
T	T	T	T	F	T
T	F	F	T	F	F
F	T	F	T	T	T
F	F	F	F	T	T

   - Sentences formed by these connectives are called well formed formulas (wff).
   - Brackets of the form (,), [,], {, } can be used to group symbols into sub expressions and thus control the order of evaluation, for example $(P \vee Q)=R$ and $P \vee (Q=R)$.

* **Common Identities in Propositional Logic**

1- $\neg(\neg P) \equiv P$.
2- $(P \vee Q) \equiv (\neg P \Rightarrow Q)$.
3- $\neg(P \vee Q) \equiv (\neg P \wedge \neg Q)$.
4- $\neg(P \wedge Q) \equiv (\neg P \vee \neg Q)$.
5- $P \vee (Q \wedge R) \equiv (P \vee Q) \wedge (P \vee R)$.
6- $P \wedge (Q \vee R) \equiv (P \wedge Q) \vee (P \wedge R)$.
7- $P \wedge Q \equiv Q \wedge P$.
8- $P \vee Q \equiv Q \vee P$.
9- $(P \wedge Q) \wedge R \equiv P \wedge (Q \wedge R)$.
10- $(P \vee Q) \vee R \equiv P \vee (Q \vee R)$.
11- $P \Rightarrow Q \equiv \neg P \vee Q$.   (important).
12-

* **Predicate Logic**

predicate (arguments).

Relationship between objects.

Property of an object or objects

Examples

1- liquid (water).
   is (water, liquid).

2- Today is Saturday
   day (Saturday).

3- It is raining today.
   weather (today, rain).

4- P1= "it rained on Saturday".
   P2 = " it rained on Sunday".
   P3= "it rained on Monday".
   P4 = " it rained on Tuesday".
   P5= "it rained on Wednesday".
   P6 = " it rained on Thursday".
   P7 = " it rained on Friday".

   weather (X, rain). Where $X \in$ {Saturday, Sunday, ..., Friday).

- Predicate logic allows us to deal with the components of a proposition.
- Predicate logic allows the use of variables, which make the sentence more general.
- The values the variable may assume have to be stated.

* **Quantifiers** $\quad \begin{cases} \forall & \text{For all} \\ \exists & \text{There exists} \end{cases}$

Examples

$\forall X$ weather (X, rain) $\longrightarrow$ true for all values of X.
$\exists X$ weather (X,rain) $\longrightarrow$ true for some values of X

* **First Order Logic**
   The quantifiers are on variables only.

* **Higher Order Logic**
   The quantifiers may be on predicates.
   Example: $\forall X \forall like$ like(X,sports).

* **Definitions**

1- Constant:
   A constant refers to a specific object or to a property of an object. A constant starts with a lower case letter.

2- Variable:
   A variable is used to refer to general cases of objects or properties. Variables start with upper case letter.

3- Function:
   A function name starts with a lower case letter. It has an associated number of arguments. Each argument can be either: constant, variable, or a function.

   Example: product (a, b)
                { return a*b;}

4- Term:
   A term refers either to constant, variable, or a function.

   - The English alphabet and the digits 0, 1, 2, ……, 9 and the underscore " _ " are used to construct a term.

5- Predicate:
   A predicate names a relationship between zero or more objects. It starts with a lower case letter and the name is constructed using the characters used for terms.

6- Atomic Sentence:
   Is a predicate with arity n followed by n terms enclosed in parentheses and separated by commas, predicate logic sentences are delimited by the period character ".".

   Examples:
      -   like (X,Y).
      -   ∀X ∃Y like(X,Y).
      -   ∃X ∀Y like(X,Y).

   Atomic sentence may be combined using the connectives (∧,∨,¬,⇒,=).

7- Literal:
   A literal is an atomic sentence or the negation of an atomic sentence.
8- Clause:
   A clause is one or more literals connected by the connectives (∧,∨,¬,⇒,=).

   A clause with one literal is called a unit clause.

* <u>Horn Clause</u>

A Horn clause has the following form:

$b_1(X,Y) \wedge b_2(X,Y) \wedge \ldots\ldots \wedge b_n(X,Y) \longrightarrow a(X,Y)$. Where the literals $a, b_1, \ldots, b_n$ are all positive.

The $a(X,Y)$ is called the head of the clause and $b_1(X,Y) \wedge \ldots\ldots \wedge b_n(X,Y)$ is called the body of the clause.

These are three cases to consider:

1- The original clause has no head $b_1(X,Y) \wedge b_2(X,Y) \wedge \ldots\ldots \wedge b_n(X,Y)$. This represents a goal to be proved.
2- The clause has no body: $a(X,Y)$. This represents a fact.
3- The clause has a body and a head: $b_1(X,Y) \wedge \ldots \wedge b_n(X,Y) \longrightarrow a(X,Y)$. In this case the clause is called a rule.

A Horn clause may be written in the form: $\neg b_1(X,Y) \vee \neg b_2(X,Y) \vee \ldots \vee \neg b_n(X,Y) \vee a(X,Y)$. Thus a Horn can be defined as a clause with at most one positive literal.

* <u>Common Identities</u>

1- $\neg \exists X\, p(X) \equiv \forall X \neg p(X)$.
2- $\neg \forall X\, p(X) \equiv \exists X \neg p(X)$.
3- $\exists X\, p(X) \equiv \exists Y\, p(Y)$.
4- $\forall X\, p(X) \equiv \forall Y\, p(Y)$.
5- $\forall X\, [p(X) \wedge q(X)] \equiv \forall X\, p(X) \wedge \forall Y\, q(Y)$.
6- $\exists X\, [p(X) \vee q(X)] \equiv \exists X\, p(X) \vee \exists Y\, q(Y)$.

Note that $\forall X\, [p(X) \vee q(X)] \not\equiv \forall X\, p(X) \vee \forall Y\, q(Y)$.

* <u>Examples:</u>

1- If it does not rain tomorrow, Zeki will go to the lake.

Solution\ $\neg$ weather (tomorrow,rain) $\longrightarrow$ go (Zeki,lake).

2- All basketball players are tall.

Solution\ $\forall X$ tall (X). $X \in$ {basketball player}.
       $\forall X$ [basketball_player (X) $\longrightarrow$ tall (X). $X \in$ {players}.
       $\forall X$ [player (X) $\wedge$ play (X,basketball) $\longrightarrow$ tall (X). $X \in$ {all people}.

3- Some students like artificial intelligence, where $X \in$ {set of all things}.

Solution\ $\exists X$ [student (X) $\wedge$ like (X,artificial_intelligence)].

4- Nobody like taxes.

Solution\ $\neg \exists X$ [like (X,taxes)] $X \in$ {set of people}.
       $\neg \exists X$ [person (X) $\wedge$ like (X,taxes)] $X \in$ {set of all things}.

* <u>**Unification**</u>

Unification is the process of making two literals look alike.

Assume that we have a set of literals to be unified: {L1, L2, L3, …., Lk}, we seek a substitution, such that: F= {(t1,v1), (t2,v2), ……, (tn,vn)} such that: L1F=L2F=……=Lkf.

<u>**Example:**</u>

1- Assume that L = p (X,Y,f(Y),b)
        F = {(a,X),(f(Z),Y)}
        LF = p (a,f(Z),f(f(Z)),b).

2- L1 = father (X,Y).
   L2 = father (X1,Y1).
   F = {(X,X1),(Y,Y1)}.
   L1F = father (X,Y).
   L2F = father (X,Y).

   Q = {(ali,X),(ahmed,Y),(ali,X1),(ahmed,Y1)}.
   L1Q = father (ali,ahmed).
   L2Q = father (ali,ahmed).

   R = {(ali,X),(ali,X1),(Y,Y1)}.
   L1R = father (ali,Y).
   L2R = father (ali,Y).

Let F be a substitution, then F is mgu (most general unifier) of s = {L1, L2, …., Lk} provided that for any other unifier Q of {L1, L2, ….., Lk} there exist a unifier R of {L1, L2, …., Lk} such that: Q(S) = R(F(S)).

Unification is done by replacing a variable by a term (important).

* **Procedure mgu**

Begin
- Generate the first disagreement set D.
- Repeat
  While (D is disagreement) do
  If non of the terms in D consists of a variable by itself then
  - stop and report failure becomes the set of literals can not be unified by any substitutions.

  Else
  - Convert variable into a term by adding a pair of the form (t,v) and simultaneously perform the substitution in the literals.
  If v is not in F as a second argument of a pair, then
    - add a pair (tp,vp) to F.
  Else
    - stop and report failure.
  End if
  End if
  End while
    - generate the next disagreement set D;
  Until (no D remain)
  Return (F).
End.

Example:

Let L1 = p (X,f(Y)).
    L2 = p(a,f(g(Z))).
Find the mgu.

Solution\

$F = \{\varnothing\}$.

Step1:
    D = {X,a}.
    F = {(a,X)}.
    L1′ = L1F = p(a,f(Y)).
    L2′ = L2F = p(a,f(g(Z))).

Step2:
    D = {f(Y),f(g(Z))}.
    F′ = {(a,X),(g(Z),Y)}.
    L1″ = L1′F′ = p(a,f(g(Z))).
    L2″ = L2′F′ = p(a,f(g(Z))).
Stop.

* **Skolemization**

Skolemization is the process of replacing existentially quantifier ($\exists$) variables by a constant (skolen constant) or a function (skolen function) of universally quantified ($\forall$) variables under whose scope.

Example:

1- $\exists X$ father (X,ahmed)   X=ali
   father (ali,ahmed).

2- $\forall Y \exists X$ father (X,Y)
   $\forall Y$ father (f(Y),Y).
   f (Y) is a skolen function.

3- $\exists X \ \forall Y$ p (X,Y). Replace X by a constant $\forall Y$ p (a,Y).

4- $\forall X \ \forall Y \exists Z \ \forall W$ p (X,Y,Z,W).
   $\forall X \ \forall Y \ \forall W$ p (X,Y,f(X,Y),W).

5- $\forall X \ \forall Y \exists Z \ \exists W$ p (X,Y,Z,W).
   $\forall X \ \forall Y$ p (X,Y,f(X,Y),g(X,Y)).

* **Clause Form**

Definition: A predicate logic expression WFF (Well Form Formula) is in clause form
          if it consists of a disjunction of literals only.
$P_1 \lor p_2 \lor \ldots \lor p_n$.
$P_1 \lor \neg p_2 \lor \ldots \lor p_n$.
$(p_1 \lor p_2) \land q1$ is not in clause form.

Any WFF can be converted into normal clause form.

Example:

$\forall X \ \{[p(X) \land q(X)] \longrightarrow [r(X,a) \land \exists Y \ (\exists Z \ r(Y,Z) \longrightarrow s(X,Y))]\} \lor \forall X \ t(X)$.

Solution:

1- Eliminate $\longrightarrow$ using the identity p $\longrightarrow$ q $\equiv$ p $\lor$ q.
   $\forall X \ \{\neg \ [p(X) \land q(X)] \lor [r(X,a) \land \exists Y \ (\neg \ \exists Z \ r(Y,Z) \lor s(X,Y))]\} \lor \forall X \ t(X)$.

2- Reduce the scope of negation.
   $\forall X \ \{[\neg \ p(X) \lor \neg \ q(X)] \lor [r(X,a) \land \exists Y \ (\forall Z \ \neg \ r(Y,Z) \lor s(X,Y))]\} \lor \forall X \ t(X)$.

3- Standarize variables, such that each quantifier assumes a different name for the variables.

$\forall X \ \{[\neg\, p(X) \vee \neg\, q(X)] \vee [r(X,a) \wedge \exists Y\ (\forall Z\ \neg\ r(Y,Z) \vee s(X,Y))]\} \vee \forall W\ t(W).$

4- Move all quantifiers to the left keeping their order.

$\forall X\ \exists Y\ \forall Z\ \forall W\ \{[\neg\, p(X) \vee \neg\, q(X)] \vee [r(X,a) \wedge (\neg\, r(Y,Z) \vee s(X,Y))]\} \vee t(W).$

5- Skolemize $\exists$ variables.

$\forall X\ \forall Z\ \forall W\ \{[\neg\, p(X) \vee \neg\, q(X)] \vee [r(X,a) \wedge (\neg\, r(f(X),Z) \vee s(X,f(X)))]\} \vee t(W).$

6- Drop the $\forall$ quantification.

$\{[\neg\, p(X) \vee \neg\, q(X)] \vee [r(X,a) \wedge (\neg\, r(f(X),Z) \vee s(X,f(X)))]\} \vee t(W).$

7- Convert into an expressions, which consists of conjunctions of disjunctions.

$\{([\neg\, p(X) \vee \neg\, q(X)] \vee r(X,a)) \wedge ([\neg\, p(X) \vee \neg\, q(X)] \vee (\neg\, r(f(X),Z) \vee s(X,f(X))))\} \vee t(W).$

$\{(\neg\, p(X) \vee \neg\, q(X) \vee r(X,a)) \wedge (\neg\, p(X) \vee \neg\, q(X) \vee \neg\, r(f(X),Z) \vee s(X,f(X)))\} \vee t(W).$

$(\neg\, p(X) \vee \neg\, q(X) \vee r(X,a) \vee t(W)) \wedge (\neg\, p(X) \vee \neg\, q(X) \vee \neg\, r(f(X),Z) \vee s(X,f(X)) \vee t(W)).$

8- Regard each disjunction as a separate clause.

ii- $\neg\, p(X) \vee \neg\, q(X) \vee r(X,a) \vee t(W).$
iii- $\neg\, p(X) \vee \neg\, q(X) \vee \neg\, r(f(X),Z) \vee s(X,f(X)) \vee t(W).$

9- Rename variables such that each clause uses a different set of variables.

i- $\neg\, p(X_1) \vee \neg\, q(X_1) \vee r(X_1,a) \vee t(W_1).$
ii- $\neg\, p(X_2) \vee \neg\, q(X_2) \vee \neg\, r(f(X_2),Z) \vee s(X_2,f(X_2)) \vee t(W_2).$

* **Reasoning With Logic**

Definition: (sound inference rule):

An inference rule is said to be sound when every logical expression $\underline{X}$ produced by the rule follows logically from the set $\underline{S}$ of clauses.

Definition: (complete):

An inference rule is complete if it is able to produce every expressions that logically follows from $\underline{S}$.

Inference Rules

1- Modus-Ponens.
2- Resolution.

Modus Ponens

If p $\longrightarrow$ q
  p true.
_____

Conclusion: q is true.

Example: " if it is raining then the ground is wet", and "it is raining"

Conclusion: "the ground is wet".

Example: " all men are mortal", and "Socrates is a man"

C1: $\forall X$ [man (X) $\longrightarrow$ mortal(X)].
C2: man ("Socrates").

Since C1 is true for all X, so it is true when X="Socrates".
man ("Socrates") $\longrightarrow$ mortal("Socrates").
man ("Socrates").
_____

Conclusion: "mortal (""Socrates").

Example:
C1: p $\longrightarrow$ q.
C2: q $\wedge$ r $\longrightarrow$ t.
C3: p.
C4: r.
_____
Is t true?

p $\longrightarrow$ q
p is true.
_____

q is true.
q $\wedge$ r $\longrightarrow$ t
q is true.
r is true.
_____

Conclusion: t is true.
* **Resolution**
    If C_1 and C_2 in normal form such that C_1 contains a literal and C_2 contains the negation of that literal. The result is a clause, which consist of the disjunction of all literals in C_1 and C_2, except the literal and its negation in the two clauses (P, $\neg$P). The resulting clause is called resolvent.

Example:

C_1: p ∨ q̸ ∨ r ∨ s
C_2: t ∨ z ∨ ¬q̸

p ∨ r ∨ s ∨ t ∨ z ————→ resolvent.

Example:

C_1: p ∨ q
C_2: ¬p
C_3: ¬q

 Empty Clause
C_4: C_1 and C_2: q
C_5: C_4 and C_3: [ ]

If the [ ] is appear in the result it mean that the contradiction in the terms (q,¬q). So to prove the goal, add the negation of the goal and then apply resolution and if [ ] appears then there is an error in the negation of the goal, which means the goal, is true and vice-versa.

* **Steps of Inference Using Resolution**
1- put the axioms in clause normal form.
2- add the negation of the goal to be proved in clause normal form to the set of axioms.
3- produce a contradiction by generating the empty clause using resolution if no contradiction can be generated, the goal cannot be proved.
4- the substitutions that are used to produce the empty caluse represent the values of the variables for which the goal is true.

Example: given the following:
1- fido is a dog.
2- all dogs are animals.
3- all animals will die.
Prove: "fido will die".

Solution:

C_1: dog (fido).
C_2: ∀X1 [dog (X1) ——→ animal (X1)].
C_3: ∀X2 [animal(X2) ——→ die (X2)].
C_1': dog (fido).
C_2': ¬ dog (X1) ∨ animal (X1).
C_3': ¬ animal (X2) ∨ die (X2).
g': ¬ die (fido).

#1

g': ¬ die (fido).

C_3': ¬ animal (X2) ∨ die (X2).   (fido,X2) → C_4: ¬ animal (fido).

#2

C_4: ¬ animal (fido).

C_2': ¬ dog (X1) ∨ animal (X1).   (fido,X1) → C_5: ¬ dog (fido).

#3

C_5: ¬ dog (fido).

C_1': dog (fido).   →   [ ] Empty clause
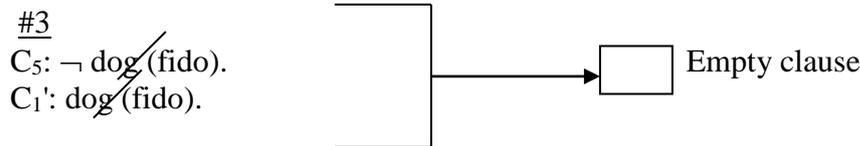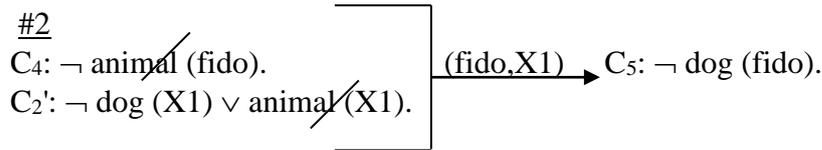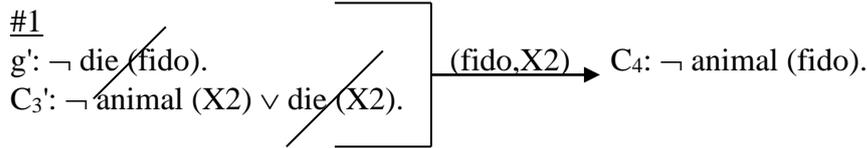
Contradiction, which means goal, is true.

Example:

" all people that are not poor and smart are happy. Those people that read are not stupid. Ali can read and is wealthy. Happy people have easy life".

Prove: can any one be found with easy life.

Solution:

C_1: ∀X1 [¬ poor (X1) ∧ smart (X1) ⟶ happy(X1)].

C_2: ∀X2 [read (X2) ⟶ smart (X2)].

C_3: read(ali) ∧ ¬ poor (ali).

C_4: ∀X3 [happy (X3) ⟶ easy_life (X3)].

Goal: ∃Z easy_life(Z).

C_1': poor (X1) ∨ ¬smart (X1) ∨ happy (X1).

C_2': ¬ read (X2) ∨ smart (X2).

C_{31}': read (ali).

C_{32}': ¬ poor (ali).

C_4': ¬ happy (X3) ∨ easy_life (X3).

g': ¬ ∃Z easy_life(Z) ≡ ∀Z ¬ easy_life(Z) ≡ ¬ easy_life(Z) .

#1

g': ¬ easy_life(Z).

C_4': ¬ happy (X3) ∨ easy_life (X3).   (Z,X3) → C_5: ¬ happy (Z).

#2

C_5: ¬ happy (Z).

C_1': poor (X1) ∨ ¬smart (X1) ∨ happy (X1).   (Z,X1) → C_6: poor (Z) ∨¬smart (Z).

#3
C_6: poor (Z) ∨¬smart (Z).
C_{32}': ¬ poor (ali).

(ali,Z) → C_7: ¬smart (ali).

#4
C_7: ¬smart (ali).
C_2': ¬ read (X2) ∨ smart (X2).

(ali,X2) → C_8: ¬read (ali).

#5
C_8: ¬read (ali).
C_{31}': read (ali).

→ [ ] Empty clause
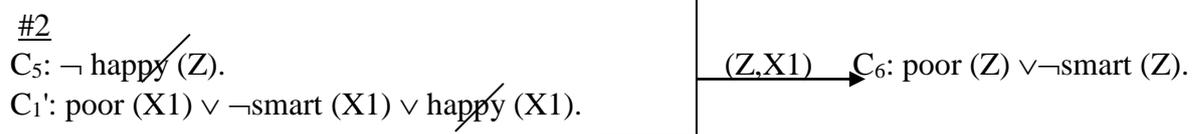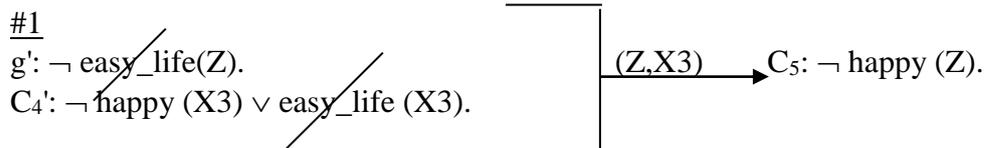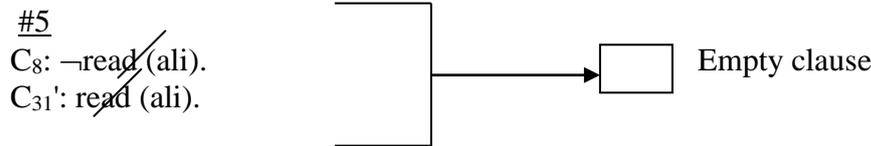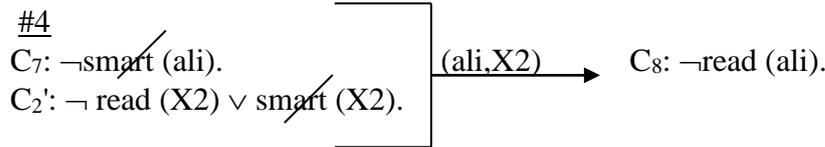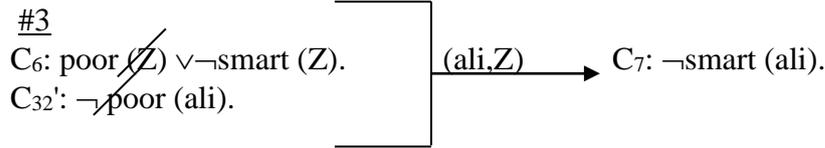
Contradiction, which means goal, is true.

* **Resolution Control Strategies**

1- **Breadth First Strategy**
In this strategy, each clause in the base set (starting set of clauses) is compared for resolution with every other clause on the first round. On the second round, the new clauses produced on the first round plus all the clauses of the base set are compared for resolution. For the nth round all previously generated clauses are added to the base set and all clauses are compared for resolution.

In this strategy, the number of clauses to be compared can become extremely large, since all early rounds are considered makes this approach inefficient for large problems.

Example: consider the following clauses:

C_1: ∀X1 [r (X1) ⟶ t(X1)].
C_2: ∀X2 [d (X2) ⟶ ¬t(X2)].
C_3: ∀X3 [d (X3) ∧ h (X3)].

Prove/ disprove the goal ∃Z [h (Z) ∧ ¬ r (Z)].

Solution:

Convert to normal form:

C_1': ¬ r (X1) ∨ t (X1).
C_2': ¬ d (X2) ∨ ¬ t (X1).
C_{31}': d (a).
C_{32}': h (a).

Goal: ¬ h (Z) ∨ r (Z).

Round 1

1- resolves C_1' and C_2'
   $C_4: \neg\, r\,(X1) \vee \neg\, d\,(X1).$ $\{(X1,X2)\}$
2- resolves C_1' and goal
   $C_5: t\,(X1) \vee \neg\, h\,(X1).$ $\{(X1,Z)\}$
3- resolves C_2' and C_{31}'
   $C_6: \neg\, t\,(a)$ $\{(a,X2)\}$
4- resolves C_{32}' and goal
   $C_7: r\,(a).$ $\{(a,Z)\}$

Round 2

5- resolves C_1' and C_6
   $C_8: \neg\, r\,(a).$ $\{(a,X1)\}$
6- resolves C_1' and C_7
   $C_9: t\,(a).$ $\{(a,X1)\}$
7- resolves C_2' and C_5
   $C_{10}: \neg\, d\,(X1) \vee \neg\, h\,(X1).$ $\{(X1,X2)\}$
8- resolves C_{31}' and C_4
   $C_{11}: \neg\, r\,(a).$ $\{(a,X1)\}$
9- resolves C_{32}' and C_5
   $C_{12}: t\,(a).$ $\{(a,X1)\}$
10- resolves goal and C_4
   $C_{13}: \neg\, d\,(X1) \vee \neg\, h\,(X1).$ $\{(X1,Z)\}$
11- resolves C_4 and C_7
   $C_{14}: \neg\, d\,(a).$ $\{(a,X1)\}$
12- resolves C_5 and C_6
   $C_{15}: \neg\, h\,(a).$ $\{(a,X1)\}$

Round 3

13- resolves C_2' and C_9
   $C_{16}: \neg\, d\,(a).$ $\{(a,X2)\}$
14- resolves C_2' and C_{12}
   $C_{17}: \neg\, d\,(a).$ $\{(a,X2)\}$

15- resolves C_{31}' and C_{10}
   $C_{18}: \neg\, h\,(a).$ $\{(a,X1)\}$
16- resolves C_{31}' and C_{13}
   $C_{19}: \neg\, h\,(a).$ $\{(a,X1)\}$
17- resolves C_{31}' and C_{14}
   $C_{20}:$ ☐   empty.
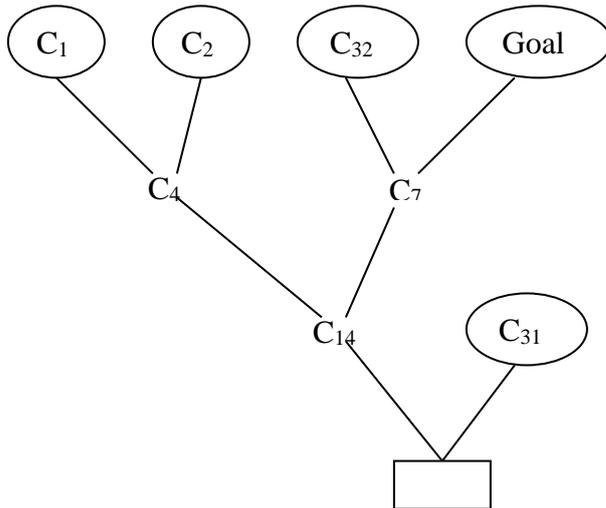
2- **Set of Support Strategy**

For a set of input clauses S, we can specify a subset T of S called the set of support. The strategy requires that at least one of the resolvents in each resolution operation be in the set of support. This strategy is based on the principle that the negation of the goal is going to be responsible for generating empty clause.

The set of support consists initially of the negation of the goal consequently any resolvent whose parent in the set of support become member of the set of support. This strategy is good for dealing with large number of clauses.

Example: consider the following clauses:

C_1: $\forall X1$ [r (X1) $\longrightarrow$ t(X1)].
C_2: $\forall X2$ [d (X2) $\longrightarrow$ $\neg t$(X2)].
C_3: $\forall X3$ [d (X3) $\wedge$ h (X3)].

Prove/ disprove the goal $\exists Z$ [h (Z) $\wedge \neg$ r (Z)].

Solution: Convert to normal form:

C_1': $\neg$ r (X1) $\vee$ t (X1).
C_2': $\neg$ d (X2) $\vee \neg$ t (X1).
C_{31}': d (a).
C_{32}': h (a).
Goal: $\neg$ h (Z) $\vee$ r (Z).
S = {$\neg$ h (Z) $\vee$ r (Z)}.

1- resolves goal and C_1'
   C_4: $\neg$ h (Z) $\vee$ t (Z1).   {(Z,X1)}
   S = {$\neg$ h (Z) $\vee$ r (Z), C_4}.
2- resolves C_4 and C_2'
   C_5: $\neg$ h (Z) $\vee \neg$ d (Z).   {(Z,X2)}
   S = {$\neg$ h (Z) $\vee$ r (Z), C_4, C_5}.

3- resolves C_5 and C_{31}'

   $C_6: \neg h (a)$ $\{(a,Z)\}$

   $S = \{\neg h (Z) \vee r (Z), C_4, C_5, C_6\}$.

4- resolves C_6 and C_{32}'

   $C_7:$ [ ] empty.
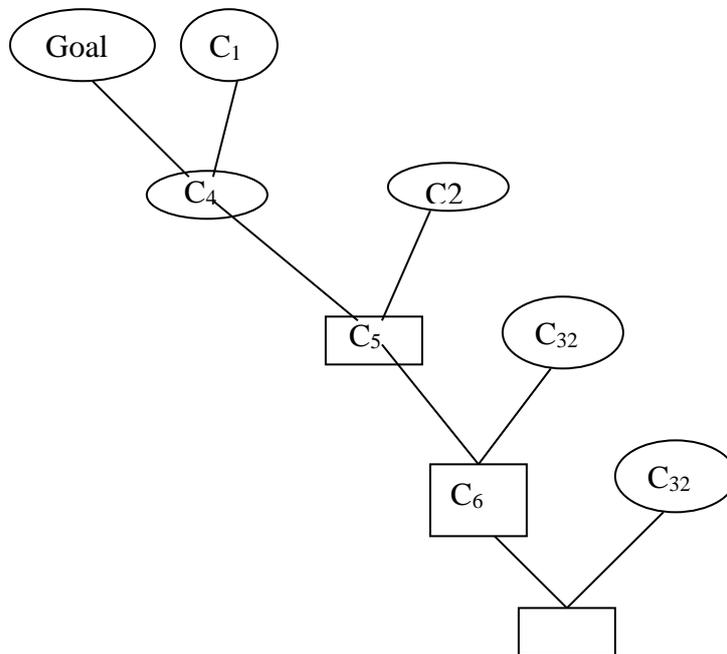
3- **Unit Preference Strategy**

    In this strategy, clauses are chosen for resolution, with as fewer literals as possible so that the empty clause can be produced with fewer number of resolutions.

Example: consider the following clauses:

$C_1: \forall X1 [r (X1) \longrightarrow t(X1)]$.

$C_2: \forall X2 [d (X2) \longrightarrow \neg t(X2)]$.

$C_3: \forall X3 [d (X3) \wedge h (X3)]$.

Prove/ disprove the goal $\exists Z [h (Z) \wedge \neg r (Z)]$.

Solution: Convert to normal form:

$C_1': \neg r (X1) \vee t (X1)$.

$C_2': \neg d (X2) \vee \neg t (X1)$.

$C_{31}': d (a)$.

$C_{32}': h (a)$.

Goal: $\neg h (Z) \vee r (Z)$.

1- resolves goal and C_{32}'
   C_4: r (a). $\{(a,Z)\}$
2- resolves C_4 and C_1'
   C_5: t (a). $\{(a,X1)\}$
3- resolves C_5 and C_2'
   C_6: $\neg$ d (a) $\{(a,X2)\}$
4- resolves C_6 and C_{31}'
   C_7: [   ] empty.

4- **Linear Input Format Strategy**

     In this strategy, the negated goal is resolved with one of the original input clauses. The resulting clause is resolved with one of the original clauses and so on until the empty clause is generated or there are no other resolutions. This strategy is not complete.

Example: consider the following clauses:

C_1: $\forall X1$ [r (X1) $\longrightarrow$ t(X1)].
C_2: $\forall X2$ [d (X2) $\longrightarrow$ $\neg$t(X2)].
C_3: $\forall X3$ [d (X3) $\wedge$ h (X3)].
Prove/ disprove the goal $\exists Z$ [h (Z) $\wedge$ $\neg$ r (Z)].
Solution: Convert to normal form:

C_1': $\neg$ r (X1) $\vee$ t (X1).
C_2': $\neg$ d (X2) $\vee$ $\neg$ t (X1).
C_{31}': d (a).
C_{32}': h (a).

Goal: $\neg$ h (Z) $\vee$ r (Z).

1- resolves goal and C_{32}'
   C_4: r (a).   {(a,Z)}
2- resolves C_4 and C_1'
   C_5: t (a).   {(a,X1)}
3- resolves C_5 and C_2'
   C_6: ¬ d (a)   {(a,X2)}
4- resolves C_6 and C_{31}'
   C_7: [   ] empty.

5-the ancestry filtered form strategy
6-combinations of strategy

Expert Systems

Expert system breakthrough began in the late 1970's on the bases of the success of a few programs that performed specific tasks almost as well as human expert.

An expert system refers to a computer system, which exhibits the human expert's intelligence, handles real world problems requiring the expert's involvement, uses a computer model of expert knowledge and expert reasoning and is compatible with or even superior to a human expert in performance.

The expert system relies on a body of knowledge to perform somewhat difficult task usually performed by a human expert.

An expert system deals successfully with problems for which clear algorithmic solutions do not exist.

Expert systems are practical programs that use heuristic strategies to solve specific classes of problems.

Many expert systems serve the role of an expert advisor to help experts solve problems by providing them with useful advice.
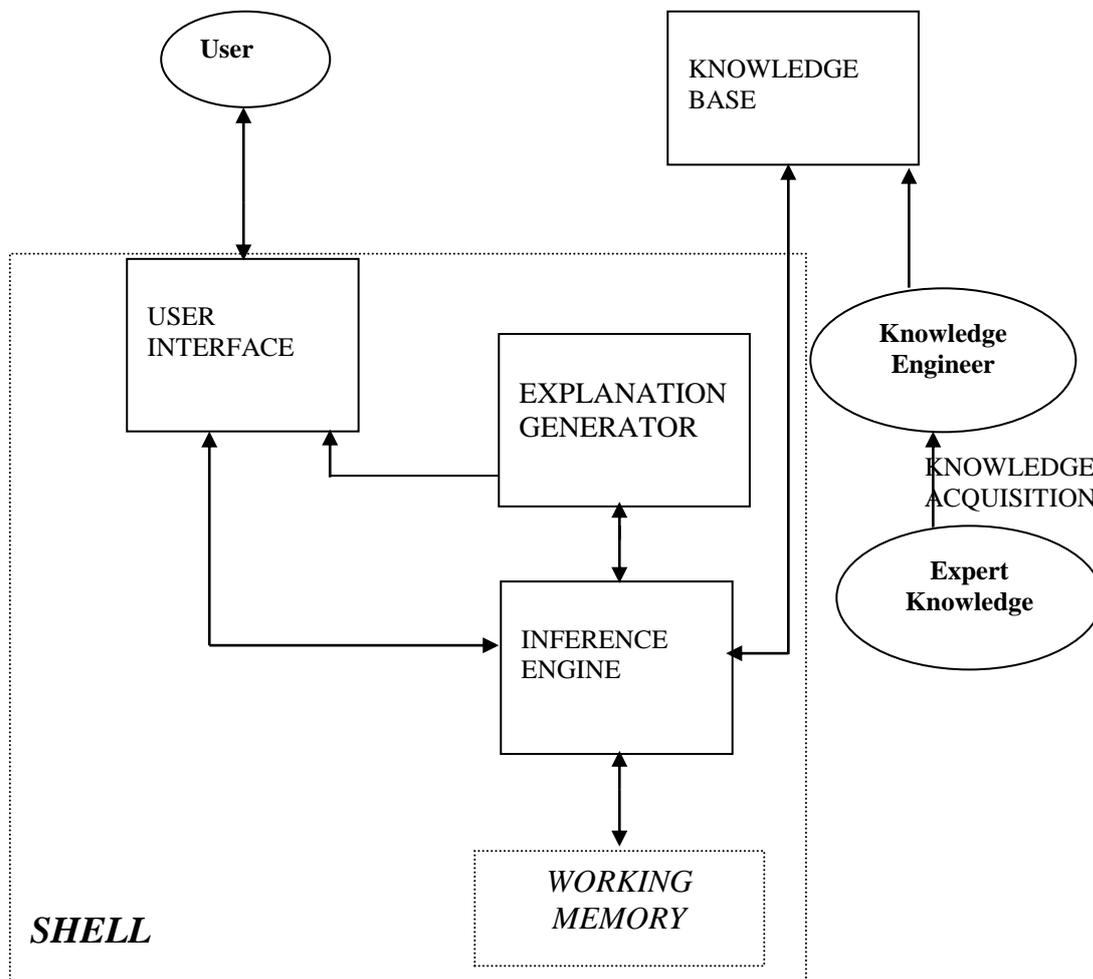
Main components of any expert system.

44

Most expert systems are composed of four parts:

1- The knowledge base which contains the necessary long term memory of facts, structures and rules that represent the knowledge about the domain of the expertise.
2- The inference engine which controls the reasoning process of the system, it uses the expert knowledge to solve the problem.
3- A user interface in the form of a communication facility that allows users to query the system, supply information and receive advice.
4- The explanation facility, which provides explanation to justify the expert systems recommendations or decisions. Here it helps the end user to feel more assured about the actions and enables the developer to follow through the operation of the system.

General characteristics of expert systems:

Expert systems generally have the following characteristics:-

1- They allow easy modification of the knowledge base (both in adding and deleting skills or facts).
2- Reason heuristically using often-imperfect knowledge to obtain useful problem solutions.
3- Support inspection of their reasoning processes both in presenting intermediate steps and in answering questions about the solutions.
4- Conduct a dialogue with human specialists in a language that is natural to them (i.e. expert systems are frequently interactive and use human-oriented dialogue). The appearance of the intelligence is enhanced to the extent that the program can accept free form input in simple sentences and can state its conclusions in the same way. Such a system is said to have a natural language interface.
5- Capability of dealing with uncertainty rather than only clear facts and information.
6- Knowledge base and inference engine constitute separate parts of the system.
7- Most expert systems have been written for relatively specialized expert level domains.
8- Generally expert systems use rule-based architecture for automated reasoning. The mechanisms used by most expert systems are built on rule-based architecture. This is because rules are easy to understand and constitute a basic construct in logic programming.

One of the design principles of rule-based expert systems is that the knowledge base is kept separate from the inference engine. This is because:

1- The separation of the knowledge base and the inference engine makes it possible to represent knowledge in a more natural fashion (if …. Then….)

than a program, which embeds this knowledge in a lower level computer code.

2- The separation of the knowledge and control a long with the modularity provided by the rules and other structures, allows changes to be made in the knowledge base without affecting other parts of the program.

3- It allows the same control and interface software to be used in a variety of systems (the expert system SHELL). The expert system shell has all components of the expert system except that knowledge base and the case-specific data contain no information. Programmers can use empty shell and create a new knowledge base appropriate to another application.

4- This modularity allows the experimentation with alternative control regimes for the same rule-base.

Categories of Expert System

Expert systems have been used successfully in diverse problem domains and may be classified according to type of tasks for which they have been constructed. The basic activities of the expert system can be grouped into the categories shown in the following table.

Category	Problem Addressed
Interpretation	Inferring situation descriptions from sensor data
Prediction	Inferring likely consequences of given situations
Diagnosis	Inferring system malfunctions from observables
Design	Configuring objects under constraints
Planning	Designing actions
Monitoring	Comparing observations to expected outcomes
Debugging	Prescribing remedies for malfunctions
Repair	Executing plans to administer prescribed remedies
Instruction	Diagnosing, debugging, and repairing student behavior
Control	Governing overall system behavior

The inference process

To be useful, inference must be controlled. Inference begins with a well-defined goal, thus reasoning must be goal-directed in some way. Inference may be viewed in terms of a tree of possibilities, which provides a diagrammatic way of representing the structure of knowledge.

With rule-based systems, each rule consists of a premise and a conclusion. Hence, can construct an AND/OR tree whose nodes are the clauses used in the rule and whose branches are arrows connecting the clauses, where :-

- Clauses joined by an AND connective form an AND node.

- Clauses joined by an Or connective form an OR node.

The root of the tree is top-level goal to be proved. To prove the goal, we have to traverse part of the tree, as is the case. The parts of the tree that we traverse to prove the goal is called the **PROOF path.** The proof path it self is a sub tree, which is called the INFERENCE TREE.

Different methods of inference traverse the tree in different order :-

- In backward chaining, we start at the root of the tree and follow the branches, toward the leaves until we find facts in the fact base.

- In forward chaining, we start from the leaves and work our way toward the root until we find a chain of branches that leads to the goal.

Example

1- if A and B then C

2- if C or D then E.

The inference tree is:-
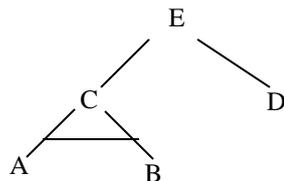
The inference tree consists of each rule represented as a conclusion with the relevant premises nested beneath the conclusion. Node C is a conclusion in the first rule and a premise in the second rule.

Rule-Based Expert System

Control Structures

Define the way and order in which facts, rules and parts of rules are used. The choice of a control structure can enormously affect the success and efficiency of a rule-based expert system. Rule based expert systems typically need large number of rules about a problem domain in order to approach skilled human performance. Hence the choice of control structure is important in such system.

Control structures may be divided into the following types:-

1- Forward chaining.

2- Backward chaining.

3- Hybrid control structure.

Backward Chaining

In a backward chaining the system is provided with a specific goal to prove. To prove a goal, backward chaining begins with and focuses on the conclusions of rules. The prolog interpreter follows the backward chaining mechanism or **goal-directed reasoning**.

It gives the goal order top priority and then treats rules and facts order of equal importance after that if alternative conclusions are possible, backward chaining can try to prove the first, then try to prove the second, if the first fails and so on.

Backward chaining is a good control structure when there are many more facts than final conclusions or goals.

Example:
Consider the following rules:
R1: goal1:-fact1.
R2: goal1:-a,b.
R3: goal2(X):-c(X).
R4: a:-not (d).
R5: b:-d.
R6: b:-e.
R7: c(2):-not (e).
R8: d:-fact2,fact3.
R9: e:-fact2,fact4.

Suppose the goals are goal1 and goal2(X) and the facts are fact2 and fact3.

Solution
The rules will be tried in the following order:-

1- R1: fails because fact1 is not true.

2- R2: invokes R4.

3- R4: invokes R8.

4- R8: succeeds, hence R4 fails.

5- R3: invokes R7.

6- R7: invokes R9.

7- R9: fails. Hence R3 succeeds with X=2, i.e. goal2(2) succeeds.

There are some enhancement of backward chaining that can often improve efficiency, these include:-

1- Cache or enter as facts some or all of the conclusions reached. For instance, once we prove conclusion (b) with the preceding rules we could add b as a fact to the database, it should be put in front of rules that can prove it, so a subsequent query will find it before the rules.

2- Ask for facts as needed (virtual facts).

Implementation of Backward Chaining

The goal of most expert systems is to reach a diagnosis, assume that this is obtained in prolog style by typing the query diagnosis(X), where X describes a diagnosis.

Example: diagnosis("fuse blown"):-nosie("pop").

One problem is that rules require advance entry of facts (often many facts). Hence can use method of facts demanded when needed. Define a predicate (ask) as a way to get facts. This predicate has two arguments. The first is a question text to be typed out on the screen and the second argument can be a variable to be bound to the answer.

Ask(Q,A):-write(Q), write("?"), readln(A), assert(asked(Q,A)).

In large rule based expert systems, two important coding techniques are used:-

1- Coding of input (answers).

2- Coding of output (questions and diagnosis).

Input coding groups user answers into categories. An important case is questions with "yes" or "no" answers, can define two new predicates: affirmative and negative for +ve or −ve answers respectively.

affirmative("yes").   negative("no")

affirmative("y").   negative("n")

affirmative("right").   negative("not")

affirmative("ok").   negative("impossible")

Can also define a predicate:

ask_if(Q):- ask(Q,A), positive_answer(Q,A).

positive_answer(Q,A):- affirmative(A).

positive_answer(Qcode,A):- not (negative(A)), not (affirmative(A)),

write("please   answer   yes   or   no"),   readln(A2),
retract(asked(Qcode,A)),
assert(asked(Qcode,A2)),affirmative(A2).

Ask_if_not(Q):-not(ask_if(Q)).

The predicate (ask_if) will succeed if the question is answered affirmatively.

Users may not always understand a question, can let them type (?) and give them some explanatory text and then give them another chance to answer:-

ask(Q,A):- asked(Q,A).
ask(Q,A):- not (asked(Q,A)), write(Q), write("?"), readln(A2), ask2(Q,A2,A).

ask2(Q,"?",A):- explain(Q), ask(Q,A).

ask2(Q,A,A):- not(A="?"), assert(asked(Q,A)).

Where explain is some explanatory text.

Output coding

1- Can code questions so we need not repeat their text at each mention in the rules. This makes rules easier to read and help prevent mistakes. Hence may use a predicate "question_code" of two arguments (a code word and a text string for the question).

Example :

diagnosis ("fuse blown"):- ask_if("device dead"), ask_if ("lights out").

question_code("device dead", "does the devise refuse to do anything").

question_code("lights out", "do all lights in the house seem to be off").

To handle this, must redefine ask:

ask(Qcode,A):- asked(Qcode,A).

ask(Qcode,A):- not(asked(Qcode,A)), question_code(Qcode,Q), write(Q), write("?"),
        readln(A2), ask2(Q,Qcode,A2,A).

ask2(Q,Qcode,"?",A):- explain(Qcode), ask(Qcode,A).

ask2(Q,Qcode,A,A):- not(A="?"), assert(asked(Qcode,A)).

2- Can handle a class of related questions together by giving arguments to output codes as in hear(X) to represents a question about hearing a sound X and writing a question code rule instead of a fact.

   Example: question_code(hear(X),X):- write("did you hear a sound like a"),
                                            write(X).

3- Can also code diagnosis which helps when diagnose are provable in many different ways. Diagnosis coding request a new top-level predicate that users may query instead of diagnosis.

   coded_diagnosis(D):- diagnosis(X), diagnosis_code(X,D).

Forward Chaining

Often rule-based systems work from just a few facts, but are capable of reaching many possible conclusions. Examples are those expert systems that identify an object from a description of what you see at a distance or diagnosis systems that tell you what to do when the car breaks down from a description of what isn't working.

Forward Chaining Algorithm

1- Mark all facts as unused and get a fresh copy of rules.
2- Until no more unused facts remain, pick the first listed one call it F, pursue it:-
   a- For each rule R (in order) that can match F with a predicate expression on its right side, ignoring appearances of F in NOTs:-
      i- Create a new rule just like R except with the expression matching F removed, if variables had to be bound to make the match, substitute these bindings for variables in the rules.
      ii- If you have now removed all of the right side of rule R, you have proved a fact, the current left side. Enter that left side into the list of facts and mark it "unused". The focus of attention here puts the new facts in front of other unused facts. Eliminate from further consideration, all rules whose left sides are equivalent to the fact just proved.

iii-   Otherwise, if there is still some right side remaining, put the new simplified rule in front of the old rule, with one exception: if the fact can match other expressions in the same rule, put the new rule after the old rule. Cross out the old rule, if it is now redundant. It is redundant if the old rule always succeeds whenever the new rule succeeds, which is true in the case where no variables were bound to make the match.

    b-   Mark F as used.

3- For each NOT expression in rules whose argument does not match any used fact, add it to the fact list, mark it as unused and redo step 2. Consider the expressions in rule order.

Example

R1: goal1:- fact1.
R2: goal1:- a,b.
R3: goal2(X):- c(X).
R4: a:- not(d).
R5: b:- d.
R6: b:- e.
R7: c(2):- not(e).
R8: d:- fact2,fact3.
R9: e:- fact2,fact4.

Given facts: fact2 and fact3

Solution

1- Start with fact2, matching expressions in R8 and R9. This gives new rules (R10 placed before R8 and R11 placed before R9):-

    The set of the new rules becomes:
        R1: goal1:- fact1.
        R2: goal1:- a,b.
        R3: goal2(X):- c(X).
        R4: a:- not(d).
        R5: b:- d.
        R6: b:- e.
        R7: c(2):- not(e).
        R10: d:- fact3.
        R11: e:- fact4.

2- Takes fact3, R10 succeeds and a new fact d is proved. The set of new rules becomes:
        R1: goal1:- fact1.
        R2: goal1:- a,b.
        R3: goal2(X):- c(X).
        R4: a:- not(d).
        R5: b:- d.
        R6: b:- e.

R7: c(2):- not(e).
R11: e:- fact4.

3- Fact d is used next, R5 now succeeds and give a new fact b, R5 and R6 are removed. The set of new rules becomes:
      R1: goal1:- fact1.
      R2: goal1:- a,b.
      R3: goal2(X):- c(X).
      R4: a:- not(d).
      R7: c(2):- not(e).
      R11: e:- fact4.
4- Fact b matches something in R2 giving:-
    R12: goal1:-a. , rule R2 can be eliminated. The current set of rules:-
      R1: goal1:- fact1.
      R12: goal1:- a.
      R3: goal2(X):- c(X).
      R4: a:- not(d).
      R7: c(2):- not(e).
      R11: e:- fact4.
5- Hence no more facts to pursue, takes the rules with NOTs.
6- Fact d is true, so R4 can never succeed. But fact e has not been proved. Hence add not(e) to the list of facts.
7- This matches the right side of R7, hence c(2) is a fact too. Eliminate R7, The current set of rules:-
      R1: goal1:- fact1.
      R12: goal1:- a.
      R3: goal2(X):- c(X).
      R4: a:- not(d).
      R11: e:- fact4.
8- 8- This matches the only expression on the right side of R3 when X=2 and hence goal2(2) is a fact, we can't eliminate R3 now because a variable had to be bound to make the match. The current set of rules:-
      R1: goal1:- fact1.
      R12: goal1:- a.
      R3: goal2(X):- c(X).
      R4: a:- not(d).
      R11: e:- fact4.

Example
R1: top(X,Y,Z):- side(Z,W,Y,X).
R2: side(A,B,7,D):- data(A,0,B),data(A,D,1).

Facts: data(3,0,1),data(3,2,1).

Solution

Cycle1: use fact data(3,0,1)
  R1: top(X,Y,Z):- side(Z,W,Y,X).
  R2: side(A,B,7,D):- data(A,0,B),data(A,D,1).
  R3: side(3,1,7,D):- data(3,D,1).

Obtain fact side(3,1,7,0).

Cycle2: use fact side(3,1,7,0)
   R1: top(X,Y,Z):- side(Z,W,Y,X).
   R2: side(A,B,7,D):- data(A,0,B),data(A,D,1).
   R3: side(3,1,7,D):- data(3,D,1).

   Obtain fact top(0,7,3).

Cycle3: use fact top(0,7,3)
     No change

Cycle4: use fact data(3,2,1)
   R1: top(X,Y,Z):- side(Z,W,Y,X).
   R2: side(A,B,7,D):- data(A,0,B),data(A,D,1).
   R4: side(3,B,7,2):- data(3,0,B).
   R3: side(3,1,7,D):- data(3,D,1).

   Obtain fact side(3,1,7,2).

Cycle5: use fact side(3,1,7,2)
   R1: top(X,Y,Z):- side(Z,W,Y,X).
   R2: side(A,B,7,D):- data(A,0,B),data(A,D,1).
   R4: side(3,B,7,2):- data(3,0,B).
   R3: side(3,1,7,D):- data(3,D,1).

   Obtain fact top(2,7,3).

Cycle6: use fact top(2,7,3)
     No change, and stop.

Example: consider the following rules:-
   R1: a:- v,t.
   R2: a:- b,u,not(t).
   R3: m(X):- n(X),b.
   R4: b:-c.
   R5:- t:- r,s.
   R6: u:- v,r.
   Facts: r,v,c,n(12).

Solution

Cycle1: use fact r
   R1: a:- v,t.
   R2: a:- b,u,not(t).
   R3: m(X):- n(X),b.
   R4: b:-c.
   R7:- t:- s.
   R8: u:- v.

Remove R5 and R6.
Remaining facts: v,c,n(12).

Cycle2: use fact v
 R9: a:- t.
 R2: a:- b,u,not(t).
 R3: m(X):- n(X),b.
 R4: b:-c.
 R7:- t:- s.

 Remove R1 and R8 and obtain fact u
 Remaining facts: u,c,n(12).

Cycle3: use fact u
 R9: a:- t.
 R10: a:- b,not(t).
 R3: m(X):- n(X),b.
 R4: b:-c.
 R7:- t:- s.

 Remove R2
 Remaining facts: c,n(12).

Cycle4: use fact c
 R9: a:- t.
 R10: a:- b,not(t).
 R3: m(X):- n(X),b.
 R7:- t:- s.

 Remove R4 and obtain b as a fact
 Remaining facts: b,n(12).

Cycle5: use fact b
 R9: a:- t.
 R11: a:- not(t).
 R12: m(X):- n(X).
 R7:- t:- s.

 Remove R10 and R3.
 Remaining facts: n(12).

Cycle6: use fact c(12)
 R9: a:- t.
 R11: a:- not(t).
 R12: m(X):- n(X).
 R7:- t:- s.

 Obtain fact m(12)
 Remaining facts: m(12).

Cycle7: use fact m(12)
   No change.

Now consider not
Since t is not a fact then obtain a new fact not(t).

Cycle8: use fact not(t)
   R12: m(X):- n(X).
   R7:- t:- s.

   Remove R9 and R11 and obtain a as a new fact
   Remaining facts: a.

Cycle9: use fact a
   No change, and stop.

Implementing Forward Chaining

1- Since pure forward chaining repeatedly finds and crosses out expressions on the right sides of rules, it would help to express the rule right side as a list and hence make use of "member" and "delete" list processing predicates to rewrite rules. Can do this by making rules a kind of a fact and defining a predicate say "rule" with two arguments:-
   - The first argument is the left side (conclusion) of the original rule.
   - The second argument is a list of predicate expressions on the right side of the rule.

   Example:
   a:- b.   becomes rule(a,[b]).
   c:- d,e,f.   becomes rule(c,[d,e,f]).
   g(X):- h(X,Y),not(f(Y))   becomes rule(g(X),[h(X,Y),not(f(Y))]).

2- Forward chaining requires that facts are identified and be distinguished from rules. This may be realized by making each fact an argument to a predicate "fact" (with a single argument).
   To implement focus of attention forward chaining and prevent fact reuse, can delete facts once considered by the system and copy them into a predicate "usedfact" with one argument.

3- To make every fact F and find the rules whose right sides can match it so as to derive new rules and possibly facts, the goal of the programming may be written as:-

        Forward:- done.
        Forward:-fact(F),not(pursuit(F)),assertz(usedfact(F)),
                  retract(fact(F)),forward.

   To make sure that all possible conclusions are reached such that forward chaining continues until there are no more facts:-
   done:- not(fact(F)).

The pursuit predicate can cycle through the rules:-

pursuit(F):- rule(L,R),rule_pursuit(F,L,R),fail.

Hence the revised code:-

```
Forward:- done.
Forward:- fact(F),not (pursuit(F)),assertz(usedfact(F)),
          retract(fact(F)),forward.
pursuit(F):- rule(L,R),rule_pursuit(F,L,R),fail.
rule_pursuit(F,L,R):- member(F,R),delete(F,R,Rnew),retract(rule(L,R)),
                     new_rule(L,Rnew).
new_rule(L,[]):- not(fact(L)),asserta(fact(L)).
new-rule(L,R):- not(R=[]),asserta(rule(L,R)).
```

Note: The above solution assumes that rules do not contain NOTS.

Hybrid Control Structure

These are hybrids of forward and backward chaining. The most common is the rule cycle hybrid.

Rule Cycle Hybrid

1- Cycle through the rules repeatedly until no new facts are found, on a cycle, ignoring rules with NOTS:-

> For each cycle, consider the rules in order:-

> > For each rule R, treat its right side as a query about the facts (without using any other rules via backward chaining), if R succeeds, add its left side (with substitution bindings made) as a fact at the front of the list of facts. And then eliminate from further consideration all rules whose left sides are equivalent to this new fact, if the rule left side has variables, do this for every possible way of binding those variables.

2- Repeat the previous step with all the original rules, taking also the NOT whose arguments are not facts.

Example

R1: goal1:- fact1.
R2: goal1:- a,b.
R3: goal2(X):- c(X).
R4: a:- not(d).

R5: b:- d.
R6: b:- e.
R7: c(2):- not(e).
R8: d:- fact2,fact3.
R9: e:- fact2,fact4.

Given facts: fact2 and fact3.

Solution

Cycle1: Rules R1,R2,R3,R5,and R6 are tried none succeed. R4 and R7 have NOTs. R8 is executed, fact b is obtained as a fact and hence asserted, R8 is removed, R9 fails.

Cycle2: R5 is executed, fact b is asserted. Both R5 and R6 are eliminated. Now the new rules are:

> R1: goal1:- fact1.
> R2: goal1:- a,b.
> R3: goal2(X):- c(X).
> R4: a:- not(d).
> R7: c(2):- not(e).
> R9: e:- fact2,fact4.

Cycle3: No rule is executed.

Next consider all rules including NOTs.

Cycle4: R7 is executed (e is not a fact), R4 fails because d is a fact. Now as a result of R7, c(2) is a fact, eliminate R7.

Cycle5: R3 is executed with X=2 and goal2(2) must be a fact, R3 not eliminated because goal2(2) is more specific than goal2(X).

Can stop now if we want to reach a goal.

Note: This algorithm will only work with restriction that no not(P) occurs in the right side of a rule before a rule having P as its left side.

Partitioned Control Structure

In large expert systems (where there are thousand or more rules), rules can interrelate in many ways, so a good practice is to divide rules into groups, modules, partitions for which members of each group have minimal interactions with members of other groups.

Each group may be considered as a separate rule-based expert system that may call another. This idea is called partitioning or context limiting control structure. To implement such control structure we normally use one other partition of rules, a "startup" partition to look at key evidence and decide on the partition most relevant to the problem. Also partitions can choose to transfer control to other partitions, if say none of their rules succeed.

A good example is the diagnosis of malfunctioning car: electrical problems, transmission, fuel system, car body,….

Meta Rules

Specification of control may be itself implemented by a rule-based system. Meta rules are just rules whose domain of knowledge is the operation of another rule-based system. They are a kind of heuristics. An example is rules to load partitions, also rules for rule ordering. Meta rules are in fact rules that treat other rules as data usually by choosing the one to use next.

Example: prefer(L1,R1,L2,R2):-length(R1,Len1),length(R2,Len2),Len1<Len2.

This rule says that a rule with shorter right side is preferred.

Meta rules permit flexible control adjustable to the processing control and hence enhance general-purpose control structures. This means that Meta rule implementation is different for backward, hybrid and forward chaining.

The big advantage of Meta rules is their flexibility and modifiability, which allows precise control of a rule-based system.

Decision Lattices

Choosing a good sequence for rules can be important and hard. However computers use storage structures besides sequences. Rules can be organized in a hierarchy that is called decision lattice. Therefore decision lattices do a restricted but very efficient kind of reasoning (a kind of classification). Decision lattices are useful for simple expert systems. Consider the expert system to diagnose the malfunction of an appliance.
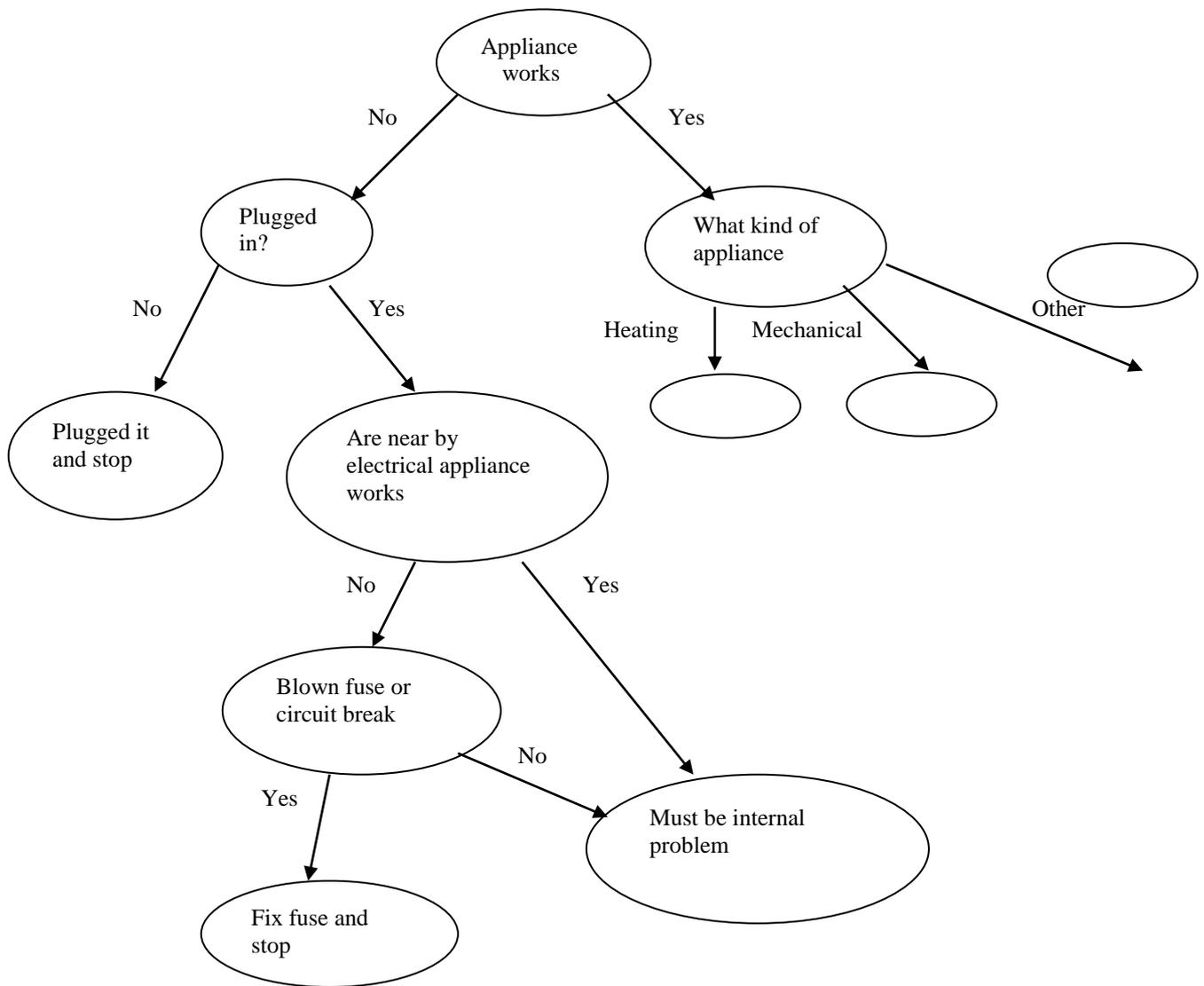
Decision lattices have several advantages:

1- Implementation is easy, can use pointers to indicate where to go next.

2- They can easily support portioning of expert systems.

3- They need not explicitly pose questions but can examine buffer contains.

However decision lattices have disadvantages:-

1- They can't reason efficiently because they don't permit backtracking or use of variables.

2- They are difficult to modify and debug since later questions assume certain results to earlier questions.

3- They may be hard to build because at each point, you must try to determine the best question to ask

Steps to Build Decision Lattices

1- For every top-level rule, repeatedly substitute in the definitions for all intermediate predicates on its right side until no more remains, if there is more than one rule proving an intermediate predicate, make multiple versions of the rule, one for each possibility.

2- Examine the right sides of the new rules pick a predicate expression that appears unnegated in some rules and negated in approximately equal number (the more rules it appears in the better and the more even. The spilt the better). Call this the partitioning predicate expression and have the first question to the user to ask about it. Create branches from the starting node to new nodes; one corresponding to each possible answer. Partition the rules into groups corresponding to the answers and associate each group with one node (copies of rules not mentioning the predicate expression should be put into every group. Remove all occurrences of the expression and its negation from rules. Within each rule group, apply this step recursively choosing a predicate that partitions the remaining rules in the group best. Repeat this until no more partitioning is possible.

Example

r:- a,d,not(e).

s:- not(a),not(c),q.

t:- not(a),p.

u:- a,d,e.

u:- a,q.

v:- not(a),not(b),c.

p:- b,c.

p:- not(c),d.

q:- not(d).

<u>Solution</u>

<u>After step1:</u>

r:- a,d,not(e).

s:- not(a),not(c),not(d).

t:- not(a),b,c.

t:- not(a),not(c),d.

u:- a,d,e.

u:- a,not(d).

v:- not(a),not(b),c.

<u>After step2:</u>

r:- a,d,not(e).	s:- not(a),not(c),not(d).
u:- a,d,e.	t:- not(a),b,c.
u:- a,not(d).	t:- not(a),not(c),d.
	v:- not(a),not(b),c.

The first set will be used whenever the fact 'a' is true and the second will be used whenever the fact 'a' is false. In the first group 'd' appears in all rules, so it can be used as the partitioning expression, similarly 'c' can partition the second group. This gives four rule groups or sub databases.

r:- not(e).   a,d rule database

u:- e.

u. } a,not(d) rule database.

t:- b.

v:- not(b).   not(a),c rule database.

s:- not(d).

t:-d.   not(a),not(c) rule database.

Three groups have two rules and hence can be portioned further to give a unique answer.

Implementation

To implement a decision lattice:

1- Give code names to every node in the decision lattice, including the starting nodes.

2- Define a successor predicate of two arguments that gives conditions for one node to be followed by another node.

Example

  successor(N1,N2):- ask_if("a").

  successor(N1,N3):- ask_if_not("a").

3- Define a diagnosis predicate as follows:-

 diagnosis(N,N):- not(successor(N,_)).

 diagnosis(D,Start):- successor(Start,X),diagnosis(D,X).

Uncertainty

Probability is used to model degrees of uncertainty in the world. A probability is the fraction of the time we expect something will be true. Rules in rule bases systems can be:-

1- Absolute, when things are absolutely true on the right side of a rule then the conclusion on the left side is absolutely true.

2- Can be uncertain or probabilistic: inference and facts can be some degree uncertain, this is particularly true when facts represent evidence and rules represent hypotheses explaining the evidence. Examples are the diagnosis rules for a car repair system.

Usually probability is added as a last argument in a prolog fact. An example:

  battery("dead",0.03).

Which implies that a battery in a randomly picked car is dead 3% of the time.

Rules may be modified in a similar way:

battery("dead",0.3):-ignition("won't start",1.0).

Which says that 20% of the time when the car won't start it is true the battery is dead.

Probability Issues

1- OR Combination Issue

Different rules of inference may be written of the same fact from different sources of evidence, each with its own probability. So if 50% of the time when the radio won't start, the battery is dead: battery("dead",0.5):- radio("won't play",1.0).

Hence the reason about whether the battery is dead, we should gather all the relevant rules and facts. Then somehow, must combine the probabilities from facts and successful rules to get accumulative property that the battery is dead. This is called the OR combination.

2- Rule Fact Combination Issue

Rules can be uncertain for a different reason than facts, in the preceding example, for the likely hood that the battery is dead when ignition won't start (which would be true if the engine is flooded). Then the probability that the battery is dead is less than the rule probability 0.2, because of the probability of the rule right side (i.e. the fact or evidence). Hence the probability of the rule as a whole must be combined with the probabilities of the facts on the right side. This is called rule probability with evidence probability combination issue.

3- AND Combination Issue

Rules can have several predicate expressions on their right sides and if each has a probability, we must somehow, combine these probabilities to get a total probability for the right side.

Combine Evidence

Assuming statistical independence:

This is assuming that different probabilities are probabilistically independent, i.e. occurrence of one kind of event does not make another kind any more or less likely. For example:

64

1-   If a quarterly report on economic indicators says that interest rates will go up this year, then the stock market index will go down tomorrow with probability 0.7.

2-   If the stock market index has gone up for three straight day, it will go down tomorrow with probability 0.4.

AND Combination

In general, if events A,B,C,…. are independent, then:-

P[A and B and C,and …]=P(A)*P(B)*P(C)*…… where P(A)= probability of A.

Example

Suppose we have the following rules, facts:-

F=a,b,c.

Given that P(a)=0.7, P(b)=1, P(c)=0.95 and rule probability=0.8. then the total probability of F=0.7*1*0.95*0.8=0.532.

OR Combination

P[A or B or C,or …]=1-[(1-P(A))*(1-P(B))*(1-P(C))*……

Example

Suppose we have the following rules, facts:-

F=a or b or c.

Given that P(a)=0.7, P(b)=1, P(c)=0.95 and rule probability=0.8. then the total probability of F=1-[(1-0.7)*(1-1)*(1-0.95)*(1-0.8)

         =1-[0.3*0*0.05*0.2]

         =1-0=1