Artificial intelligence

What are Knowledge Representation Schemes ?

In Al, there are four basic categories of representational schemes: logical, procedural, network and structured representation schemes.

1) **Logical representation** uses expressions in formal logic to represent its knowledge base. Predicate Calculus is the most widely used representation scheme.

2) **Procedural representation** represents knowledge as a set of instructions for solving a problem. These are usually if-then rules we use in rule-based systems.

3) **Network representation** captures knowledge as a graph in which the nodes represent objects or concepts in the problem domain and the arcs -represent relations or associations between them.

4) **Structured representation** extends network representation schemes by allowing each node to have complex data structures named slots with attached values.

We will focus on logic representation schemes in this chapter.

2.1 The Propositional Calculus

2.1.1 Symbols and Sentences

The prepositional calculus and, in the next subsection, the predicate calculus are first of all languages. Using their words, phrases, and sentences, we can represent and reason about properties and relationships in the world. The first step in describing a language is to introduce the pieces that make it up: ils set of symbols.

DEFINTION

PROPOSITIONAL CALCULUS SYMBOLS

The symbols of prepositional Calculus are, the prepositional symbols:

P. Q. R,S,T....

truth symbols

true, false

and connectives:

K,V, €, <, =

Prepositional symbols denote *propositions of* statements about the world that may be either true or raise, such as "the car is red" or "water is wet." Propositions are denoted by uppercase letters near the end of the English alphabet. Sentences in the propoEitional calculus are formed from these atomic symbols according to the following rules :

DEFINITION

PROPOSIONAL CALCULUS SENTENCES

Every prepositional symbol and truth symbol is a sentence.

For example: true, P, Q, and R are sentences.

The negation of a sentence is a sentence

for example: ¬P and ¬false are sentences

The *conjunction* or and two sentences : is a sentence

For example: P K¬P is a sentence.

The *disjunction or* of two sentences is a sentence.

For example P K¬P. is a sentence,

The *implication of* one sentence for another is a sentence.

For example P < Q» is sentence.

The *equivalence of two* sentences Is a sentence.

For example: P v Q = R is a sentence.

Legal sentences are also called *well-formed formulas* 01 *WFFs.*

IN expressions of the form P K Q. P and Q are called the *conjuncts* In P V Q and Q are referred to as *disjuncts* in an implication P ‹ Q, P is the premise or antecedent and Q, the conclusion or consequent .

In propositional calculus sentences. the symbols( ) and[ ] are used to group symbols into subexpressions and so control their order of evaluation and meaning . for example (P K Q)=R is quit different from P V (Q=R), as can be demonstrated using truth tables (section 2.1.2).

An expression is a sentence, or well-formed formula, of the prepositional calculus **if** and only if it can be formed of legal symbols through some sequence of these rules . For example .

P K Q **<** R = ¬P v ¬ QvR

is a well-formed sentence in the in the propositions calculus because:

P, Q, and R are-proposition and .thus sentences.

P K Q, the conjunction of two sentences, is a sentence.

(P K Q) ‹ R, the implication of a sentence for another, is a sentence.

¬*P* and ¬Q, the negations of sentences, are sentences.

¬P v ¬Q, the disjunction of two sentences, is a sentence.

¬P v ¬ Q v R, the disjunction of two sentences, is a sentence.

((P KQ) **<** R)= ¬ P v ¬ Q v R, the equivalence of two sentences, is a sentence.

This is our original sentence, which has been Constructed through a series of applications of legal roles and is therefore well formed.

DEFINITION

PROPOSITIONAL CALCULUS SEMANTICS

An interpretation of a set of propositional is the assignment of a truth value , either T or F , to each propositional symbol .

The symbol true is always assigned T, and the symbol false is assigned F .

The interpretation or truth value for sentence is determined by:

The truth assignment of negation , ¬ P, where P is any propositional symbol , is F if the assignment to P is T , and T if the assignment to P is F

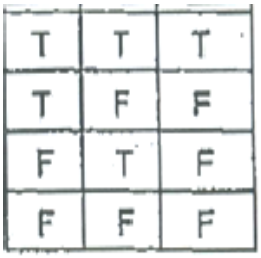
The truth assignment of conjunction K, is T only when both conjuncts have truth value T; otherwise it is F.

The truth assignment of disjunction, v is F only when both disjuncts have truth value F; otherwise it is T.

The truth assignment of implication , < is F only when the premise or symbol before the implication is T and the truth value of the consequent or symbol after the implication is F; otherwise it is T .

The truth assignment of equivalence , = is T only when both expresslone have the same assignment for all possible interpretation otherwise it is F .

P Q PKQ



T	T	T
T	F	F
F	T	F
F	F	F

Figure 2.1 truth table for the operator K

P Q ¬P ¬pvQ P<Q (¬pvQ)=(P<Q)

T.	**T**	**F**	**T**	**T**	**T**
T	**F**	**F**	**F**	**F**	**T**
F	**T**	**T**	**T**	**T**	**T**
F	**F**	**T**	**T**	**T**	**T**

Figure 2.2 Truth table demonstrating the equivalence of (P<Q) and (¬pvQ)

By demonstrating that they have identical truth tables, we can prove the following prepositional calculus equivalences. For prepositional expressions P, Q, and R;

¬ (¬P) = P

(P v Q)=(¬P < Q)

The contra positive law : (P < Q)=(¬Q < ¬P)

De morgan's law : ¬ (P v Q)=(¬P K ¬Q) and (P KQ)=(¬P v ¬Q)

The commutative laws : (P K Q)=(Q K P) and (P v Q) = (Q v P)

Associative law: ((P K Q) K R) = (P K (Q K R))

Associative law : ((P v Q) v R) = (P v (Q v R))

Distributive law : P v (Q K R) = (P v Q) K (P v R)

Distributive law : P K (Q v R) = (P K Q) v (P K R)

2.2 The Predicate Calculus

In prepositional calculus, each atomic symbol (P, Q, etc.) denotes a proposition of some complexity. There **is** no way to access the components of an individual assertion. Predicate calculus provides this ability. For example, instead **pf** letting a single prepositional symbol, P, denote ;1he entire sentence "it rained on Tuesday," we can create a predicate weather that describes a relationship between a date and the weather. weather (Tuesday, rain) through inference rules we can manipulate predicate calculus expression accessing their individual components and inferring new sentences.

Predicate calculus also allows expressions^ contain variables. Variables let us create general assertions about classes of entities. For example, we could state that for ;all values, of X, where X is a day of the week , the statement weather (X, rain) is true ; I,e., it rains it rains everyday. As with propositional calculus, we will first define the syntax of the language and then discuss its semantics.

Examples of English sentences represented in predicate calculus:

1- If it doesn't rain tomorrow, Tom will go to the mountains.

¬ weather (rain, tomorrow) ≺ go(tom, mountains).

2- Emma is a Doberman pinscher and a good dog. Good dog (emma) K isa (emma, Doberman)

3. All basketball players are tall.

6 X (basketball _ player(X) ≺ tall (X))

4- Some people like anchovies.

E X (person(X) K likes(X, anchovies)),

5- If wishes were horses, beggars would ride. equal(wishes, horses) ≺ ride(beggars).

6- Nobody likes taxes

¬ E X likes(X,taxes).

3-1 What is Resolution?

Resolution is a technique for theorem. proving in propositional and predicate calculus which attempts to show that the negation of the statement produces a contradiction with the known statements.
In the following expression, uppercase letters indicate variables (W,X,Y,andZ); lowercase letters in the middle of the alphabet indicate Constants or bound variables (I, m, and n); and early alphabetic lowercase letters indicate the predicate names (a, b, c, d, and e). To improve readability of the expressions, we use two types of brackets: ( ) and [ ]. Where possible in the derivation, we remove redundant brackets: The expression we will reduce to clause form is:

(i)(6X)([a(X) K b(X)] < [c(X, I) K (EY)(EZ)[c(Y, Z)] < d(X, Y))]) V (6X)(e(X))

1. First we eliminate the by using: $a \rightarrow b \equiv \neg a \vee b$. This transformation reduces the expression in (i) above:

(ii)(6X)(¬ [a(X) K b(X)]$[c(X, I) K (EY)(¬(EZ)[c(Y, Z)$d(X, Y))]$ < d(X, Y))]) V (6X)(e(X))

2. Next we reduce the scope of negation."This may be accomplished using a number of the transformations that includes:

¬(¬a) ÷ a

¬(EX)a(X) ÷ (6X)¬a(X)

¬(6X)b(X) ÷ (EX)¬b(X)

¬(a$b) ÷¬a$¬b

¬(a$b) ÷¬a$¬b

Using the second and fourth equivalences (ii) becomes:

(iii)(6X) (¬ [a(X)$b(X)]$ [c(X, I) K (EY)((6Z)[¬c(Y, Z)$d(X, Y])] $(6X)(e(X))

3. Next we standardize by renaming all variables so that variables bound by different quantifiers have unique names. Because variable names are "dummies" or "place holders," the particular name chosen for a variable does not affect either the truth value or the generality of the clause. Transformations used at this step are of the form:

((6X)a(X) v (6X)b(X≡(6X)a(X) v (y)b(y)

Because {iii)bas two instances of the variable X, we rename:

(ir)(6X)(¬ a(X)$¬b(X)]$[c(X, I) K (EY)((6Z)[¬c(Y, Z)$d(X, Y))]$d(X, Y))]) V (6W)(e(W))

4. Move all quantifiers to the left without changing their order. This is possible. because step 3 has removed the possibility of any conflict between variable names. (iv) now becomes:

(r)(6X)(EY)(6Z)(6W)(¬ a(X)$¬b(X)]$[c(X, I) K ([¬c(Y, Z)$d(X, Y))]$ e(W))]

After step 4 the clause is said to be in prenex normal form, because all the quantifiers are in front as a prefix and the expression or matrix follows after.

5. At this point all existential quantifiers are eliminated by a process called skolemization. Expression (v) has an existential quantifier for Y.

When an expression contains an existentially quantified variable, for example, (E Z)(foo(...Z,...)), it may be concluded that there is an assignment to Z under which foo is true. Skolemization identifies such a value. Skolemization does not necessarily show how to produce such a value; It is only a method for giving a name to an assignment that must exist. If k represents that assignment, then we have foo(….K….). Thus:

(6X)(dog(X)may be replaced by dog (fido)

where the name fido is picked from the domain of definition of X to

represent that individual X. fido is called a skolem constant. If the prediCate has more than one argument and the existentially quantified variable is within the scope of universally quantified variables, the existential variable must be a function of those other variables. This is represented in the skolemizatlon process:

$$(6X)(EY)(\text{mother}(X, Y))$$

This expression indicates that every person has' a mother. Every person is an X and the existing mother will be a function of the particular person X that is picked. Thus skolemization gives:

$$(6X)\text{mother}(X, m(X))$$

which indicates that each X has a mother (the m of that X). In another example:

$$(6X)(6Y)(EZ)(6W)(\text{foo}(X, Y, Z, W))$$

Is skolemized to:

$$(6X)(6Y)(6W)(\text{foo}(X, Y), W))$$

We note that the existentially quantified *Z* was. Within the scope (to the right of) universally quantified X and Y Thus the skolemassignment is a function of X and Y but not *of* W. With skolemization (v) becomes:

(ri)(6X)(6Z)(6W)([¬a(X)$¬b(X)]$[c(X,I)$([¬c(f(X),Z)]$d(X,f(X)))])$e(W))

where f is the skolem function of X that replaces the existential Y. Once the skolemization has occurred, step 6 can take place, which simply drops the prefix.

6. Drop all universal quantification. By this point only universally quantified variables exist (step 5) with no variable conflicts (step 3). Thus all<quantifiers can be dropped, and any proof procedure employed assumes all variables are universally quantified.

   Formula (vi) now becomes:

(rii)[¬a(X)$¬b(X)]$[c(X, I)$(¬c(f(X), Z)$d(X,f(X)))]$e(W)

7. Next we convert the expression to the conjunct of disjuncts form. This requires using the associative and distributive properties of A and v. Recall that

a$(b$c) = (a$b)$c
a$(b$c) = (a$b)$c

which indicates that $ or $ may be grouped in any desired fashion. The distributive property is also used, when necessary. Because

a$(b$c)

is already in clause form, A is not distributed However, $ must be distributed across $ using:

a$(b$c) = (a$b)$(a$c)

The final form of (vii) is:
(riii)[¬a(X)$¬b(X)$c(X, I)$e(W)$[¬a(X)$¬b(X)$¬c(f(X), Z)$d(X, f(X))$e(W)]

8. Now call each conjunct a separate clause. In the example (viii) above there are two clauses:

(ixa)[¬a(X) V c(X, I) V e(W)] K
(ixb)[¬a(X) V ¬b(X) V c(f(X), Z) V d(X, f(X)) V e(W)]

9. The final step is to standardize the variables apart again. This requires giving. The variable in each clause generated by step 8 different names. This procedure arises from the following equivalence:

(6X)(a(X) K b(X)) ÷ (6X)a(X) K (6Y)b(Y)

Which follows from the nature of variable names as place holders. (ixa) and (ixb) now become, using new-variable names U and V:
(xa)[¬a(X) V ¬b(X) V c(X, I) V e(W)] K
(xa)[¬a(U) V ¬b(U) V ¬c(f(U), Z) V d(U, f(U))V e(V)]

Ex (3): As a final example, suppose:

"All people who are not poor and are smart are hippy. Those people who' read are not stupid. John can read are is wealthy. Happy people have exciting lives. Can anyone be found with an exciting life?"

a) First change the sentences to predicate form:

We assume 6X (smart (X) ≡ stupid (X) and 6Y (wealthy (Y) ≡ poor (Y)),

and get:

6X(¬Poor (X)K smart (X) ‹ happy(X))

6Y(read(Y) ‹ smart(Y))

read (john)K ¬ poor (john)

6Z (happy (Z) ≺ exciting (Z)

The negation of the conclusion is:

¬EW (exciting (W))

b) These predicate calculus expressions for the happy life problem are transformed into the following clauses:

poor (X) V ¬ smart (X) V happy (X)

¬ read (Y) V smart(Y)

read (john)

¬ poor (john)

¬ happy (Z) V exciting (Z)

¬ exciting (W)

The resolution refutation for this example is found in Figure (3-4).

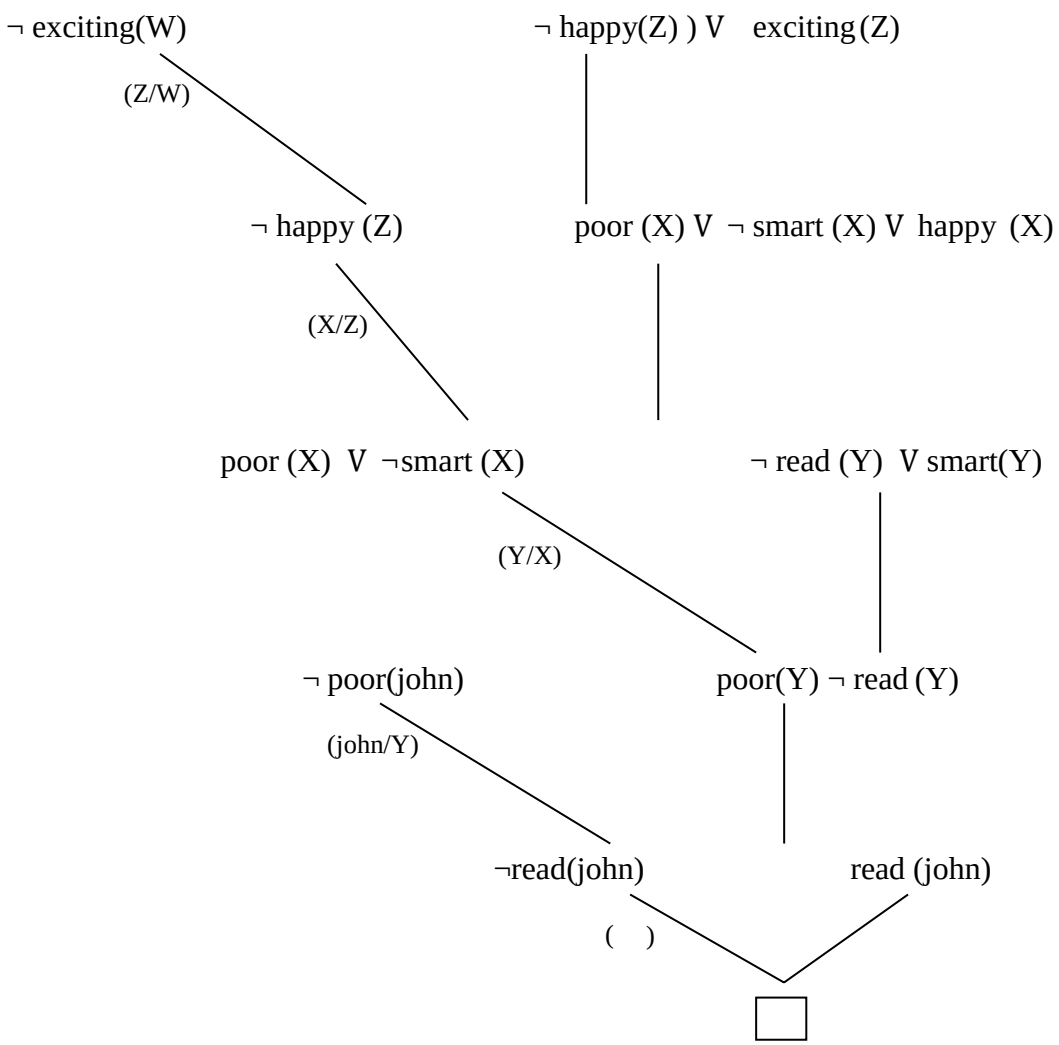


Figure (3-4):Resolution prove for the "exciting life" problem .

1 -Intelligent Search Methods and Strategies

Search is inherent to the problem and methods of artificial intelligence (AI). This is because AI problems are intrinsically complex. Efforts to solve problems with computers which human can routinely innate cognitive abilities, pattern recognition, perception and experience, invariably must turn to considerations of search. All search methods essentially fall into one of two categories, exhaustive (blind) methods and heuristic or informed methods.

2 -State Space Search

The state space search is a collection of several states with appropriate connections (links) between them. Any problem can be represented as such space search to be solved by applying some rules with technical strategy according to suitable intelligent search algorithm.

What we have just said, in order to provide a formal description of a problem, we must do the following:

1- Define a state space that contains all the possible configurations of the relevant objects (and perhaps some impossible ones). It is, of course, possible to define this space without explicitly enumerating all of the states it contains.

2- Specify one or more states within that space that describe possible situations from which the problem-solving process may start. These states are called the initial states.

3- Specify one or more states that would be acceptable as solutions to the problem. These states are called goal states.

4- Specify a set of rules that describe the actions (operators) available. Doing this will require giving thought to the following issues:

- What unstated assumptions are present in the informal problem description?
- How general should the rules be?
- How much of the work required to solve the problem should be precomputed and represented in the rules?

The problem can then be solved by using rules, in combination with an appropriate control strategy, to move through the problem space until a path from an initial state to a goal state is found. Thus the process of search is fundamental to the problem-solving process. The fact that search provides the basis for the process of problem solving does not, however, mean that other, more direct approaches cannot also be exploited. Whenever possible, they can be included as steps in the search by encoding them rules. Of course, for complex problems, more sophisticated computations will be needed. Search is a general mechanism that can be used when no more direct methods is known. At the same time, it provide the framework into which more direct methods for solving subparts of a problem can be embedded.

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, is it guaranteed to be optimal?
- What is the complexity of the search process in terms of time usage? Memory usage?
- How can the interpreter most effectively reduce search complexity?
- How can an interpreter be designed to most effectively utilize a representation language?

To get a suitable answer for these questions search can be structured into three parts. A first part presents a set of definitions and concepts that lay the foundations for the search procedure into which induction is mapped. The second part presents an alternative approaches that have been taken to induction as a search procedure and finally the third part present the version space as a general methodology to implement induction as a search procedure. If the search procedure contains the principles of the above three requirement parts, then the search algorithm can give a guarantee to get an optimal solution for the current problem.

3 -General Problem Solving Approaches

There exist quite a large number of problem solving techniques in AI that rely on search. The simplest among them is the generate-and-test method. The algorithm for the generate-and-test method can be fom1ally stated in the figure (1) follow.

It is clear from the above algorithm that the algorithm continues the possibility of exploring a new state in each iteration of the repeat-until loop and exits only when the current state is equal to the goal. Most

important part in the algorithm is to generate a new state. This is not an easy task.

```
Procedure Generate & Test
Begin
    Repeat
      Generate a new state and call it current-state;
    Until current-state = Goal;
End.
```

Figure (1) Generate and Test Algorithm

If generation of new states is not feasible, the algorithm should be terminated. In our simple algorithm, we, however, did not include this intentionally to keep it simplified. But how does one generate the states of a problem? To formalize this, we define a four tuple, called state space, denoted by

{nodes, arc, goal, current },

where

nodes represent the set of existing states in the search space;

an arc denotes an operator applied to an existing state to cause transition to another state; goal denotes the desired state to be identified in the nodes; and current represents the state, now generated for matching with the goal. The state space for most of the search problems takes the form of a tree or a graph. Graph may contain more

than one path between two distinct nodes, while for a tree it has maximum value of one.

To build a system to solve a particular problem, we need to do four things:

1. Define the problem precisely. This definition must include precise specifications of what the initial situation(s) will be as well as what final situations constitute acceptable solutions to the problem.
2. Analyze the problem. A few very important features can have an immense impact on the appropriateness of various possible techniques for solving the problem.
3. Isolate and represent the task knowledge that is necessary to solve the problem.
4. Choose the best problem-solving technique(s) and apply it (them) to the particular problem.

Measuring problem-solving performance is an essential matter in term of any problem solving approach. The output of a problem-solving algorithm is either failure or a solution. (Some algorithm might get stuck in an infinite loop and never return an output.) We will evaluate an algorithm's performance in four ways:

- Completeness: Is the algorithm guaranteed to find a solution when there is one?
- Optimality: Does the strategy find the optimal solution?
- Time complexity: How long does it take to find a solution?1
- Space complexity: How much memory is needed to perform the search?

4 - Search Technique

Having formulated some problems, we now need to solve them. This is done by a search through the state space. The root of the search tree is a search node corresponding to the initial state. The first step is to test whether this is a goal state. Because this is not a goal state, we need to consider some other states. This is done by expanding the current state; that is, applying the successor function to the current state, thereby generating a new set of states. Now we must choose which of these possibilities to consider further. We continue choosing, testing and expanding either a solution is found or there are no more states to be expanded. The choice of which state to expand is determined by the search strategy. It is important to distinguish between the state space and the search tree. For the route finding problem, there are only N states in the state space, one for each city. But there are an infinite number of nodes.

There are many ways to represent nodes, but we will assume that a node is a data structure with five components:

- STATE: the state in the state space to which the node corresponds;
- PARENT-NODE: the node in the search tree that generated this node;
- ACTION: the action that was applied to the parent to generate the node;
- PATH-COST: the cost, traditionally denoted by g(n), of the path from the initial state to the node, as indicated by the parent pointers; and
- DEPTH: the number of steps along the path from the initial state.

As usual, we differentiate between two main families of search strategies: systematic search and local search. Systematic search visits each state that could be a solution, or skips only states that are shown to be dominated by others, so it is always able to find an optimal solution. Local search does not guarantee this behavior. When it terminates, after having exhausted resources (such as time available or a limit number of iterations), it reports the best solution found so far, but there is no guarantee that it is an optimal solution. To prove optimality, systematic algorithms are required, at the extra cost of longer running times with respect to local search. Systematic search algorithms often scale worse with problem size than local search algorithms.

5 - Search Technique Types

Usually types of intelligent search are classified into three classes; blind, heuristic and random search. Blind search is a technique to find the goal without any additional information that help to infer the goal, with this type there is no any consideration with process time or memory capacity. In the other side the heuristic search always has an evaluating function called heuristic function which guides and controls the behavior of the search algorithm to reach the goal with minimum cost, time and memory space. While random search is a special type of search in which it begins with the initial population that is generated randomly and the search algorithm will be the responsible for generating the new population bases on some operations according to a special type function called fitness function. The following sections have details of each type of search with simple examples.

5.1 Blind Search

There many search strategies that come under the heading of blind search (also called uniformed search). The term means that they have no additional information about states beyond that provided in the problem definition. All they can do is generate successors and distinguish a goal state from a nongoal state.

Thus blind search strategies have not any previous information about the goal nor the simple paths lead to it. However blind search is not bad, since more problems or applications need it to be solved; in other words there are some problems give good solutions if they are solved by using depth or breadth first search.

Breadth first search is a simple strategy in which the root node is expanded first, then all the successors of the root node are expanded next, then their successors, and so on. In general all the nodes are expanded at a given depth in the search tree before any nodes at the next level are expanded. Breadth first search can be implemented by calling TREE-SEARCH with any empty fringe that is a first-in-first-out (FIFO) queue, assuring that the nodes that are visited first will be expanded first. In other words, calling TREE-SEARCH (problem, FIFO-QUEUE) result in a breadth first search. The FIFO queue puts all newly generated successors at the end of the queue, which means that shallow nodes are expanded before deeper nodes.

Depth first search always expands the deepest node in the current fringe of the search tree. The search proceeds immediately to the deepest level of the search tree, where the nodes have no successors. As

those nodes are expanded, they are dropped from the fringe, so then the search “backs up” to the next shallowest node that still has unexplored successors. This strategy can be implemented by TREE-SEARCH with a last-in first-out (LIFO) queue, also known as a stack. As an alternative to the TREE-SEARCH implementation, it is common to implement depth-first search with a recursive function that calls itself on each of its children in turn.

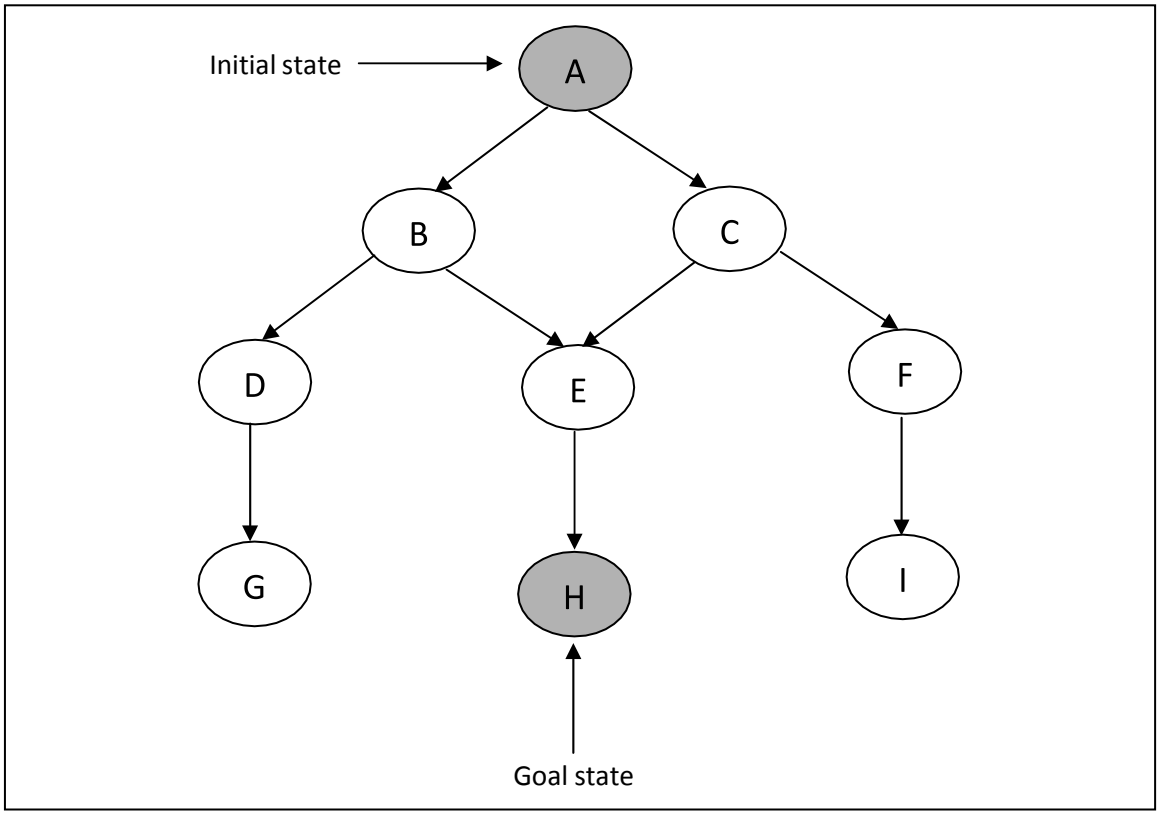


Figure (2) Blind Search Tree

The paths from the initial state (A) to the goal state (H) are:

Using Breadth first search **A€ B€ C€ D€ E€ F € G € H**

Using Depth first search **A€ B€ D€ G € E€H**

5.2 Heuristic Search

Classically heuristics means rule of thumb. In heuristic search, we generally use one or more heuristic functions to determine the better candidate states among a set of legal states that could be generated from a known state. The heuristic function, in other words, measures the fitness of the candidate states. The better the selection of the states, the fewer will be the number of intermediate states for reaching the goal. However; the most difficult task in heuristic search problems is the selection of the heuristic functions. One has to select them intuitively, so that in most cases hopefully it would be able to prune the search space correctly.

The search processes that will be described in this chapter build on the fact that the search does not proceed uniformly outward from the start node: instead, it proceeds preferentially through nodes that heuristic, problem-specific information indicates might be on the best path to a goal. We call such processes best-first or heuristic search. Here is the basic idea.

1. We assume that we have a heuristic (evaluation) function, *f*. to help decide which node is the best one to expand next. This function is based on information specific to the problem domain. It is real-valued function of state descriptions.
2. Expand next that node, n, having the smallest value of *f(n)*. Resolve ties arbitrarily.
3. Terminate when the node to be expanded next is a goal node.

A function which applies such an algorithm to nodes and assigns a value to them accordingly is a heuristic function. Determining good

heuristics is very often the most difficult element involved in solving complex problems in the symbolic artificial intelligence tradition.

Therefore, heuristic can be defined as “the study of the methods and rules of discovery and invention”.

AI problems solvers employ heuristics in two basic situations:

1. A problem may not have an exact solution because of inherent ambiguities in the problem statement or available data. A given set of symptoms may have several possible causes.

2. A problem may have an exact solution, but the computational cost of finding it may be prohibitive. In many problems state space growth is combinatorially explosive, with the number of possible states increasing exponentially or factorially with the depth of the search.

In order to solve many hard problems efficiently, it is often necessary to compromise the requirements of mobility and systematicity and to construct a control structure that is no longer guaranteed to find the best answer but that will almost always find a very answer. Thus we introduce the idea of a heuristic. A heuristic is a technique that improves the efficiency of a search process, possibly by sacrificing claims of completeness. Heuristics are like tour guides. They are good to the extent that they point in generally interesting directions; they are bad to the extent that they may miss points of interest to particular individuals. Some heuristics help to guide a search process without sacrificing any claims to completeness that the process might previously have had. Others (in fact, many of the best ones) may occasionally cause an

excellent path to be overlooked. But, on the average, they improve the quality of the paths that are explored. Using good heuristics, we can hope to get good (though possibly nonoptimal) solutions to hard problems, such as the traveling salesman, in less than exponential time. There are some good general-purpose heuristics that are useful in a wide variety of problem domains. In addition, it is possible to construct special-purpose heuristics that exploit domain-specific knowledge to solve particular problems.

Performance of the search can be drastically improved by using specific knowledge about a search problem to guide the decisions made where to search. This knowledge can be presented in the form of heuristics – rules that guide the decision-making during the search. This gives us another class of heuristic search algorithms. It is quite obvious that heuristics will depend very much on the class of solvable problems. The heuristic rules class is considered one of the more important and complex way to guide the search procedure in which to reach the solution (goal state) such class usually used in embedded system so to form what today known as searching with heuristic embedded in rules.

The search for plans is guided by heuristics that provide an estimate of the cost (heuristic value) that are extracted automatically from the problem encoding *P*. In order to simplify the definition of some of the heuristics, we introduce in some cases a new dummy End action with zero cost, whose preconditions *G*1, . . . , *Gn* are the goals of the problem, and whose effect is a dummy atom *G*. In such cases, we will obtain the heuristic estimate *h*(*s*) of the cost from state *s* to the goal, from the estimate *h*(*G*; *s*) of achieving the 'dummy' atom *G* from *s*. In the section

3.7 a detail description about the heuristic search algorithms with simple examples.

5.3 Random Search

Random Search is a method of searching with no planned structure or memory. This method has the same intent of the typical intuitive search, but the opposite strategy to achieve it. A random technique may be preferred if it takes more time to devise a better technique, or the additional work involved is negligible.

Random Search (RS) is a population-based search algorithm. It is first proposed by Price and also called Price's algorithm. RS first randomly generates a population of samples from the parameter space, and then uses downhill simplex method to move the population towards the global optima. Suppose a *n*-dimensional objective function is to be optimized, the basic RS algorithm can be described as follows:

1. Randomly generate a population of *m* points from the parameter space.
2. Randomly select *n*+1 points from the population and make a downhill simplex move.
3. If the new sample is better than the worst member in the population, then the worst member is replaced with this new sample.
4. Repeat the above process until a certain stopping criterion is satisfied.

In RS, random sampling is used for exploration and downhill simplex for exploitation. Its balance strategy first executes exploration and then

switches completely to exploitation. Since exploration is only performed in the beginning of the search, the convergence to the global optima is not guaranteed. The problem of getting trapped in a local extreme can be alleviated by using a large population or introducing new members into the population with random sampling during the search. Despite of this disadvantage, RS has proved to be very effective in practice and is widely used. In addition, since there is no local search method involved, RS is more robust to noise in the objective function. Besides it's failing to provide the convergence guarantee, another disadvantage is its low efficiency. Especially in the beginning, the population is composed of random samples

and RS essentially performs like a pure random sampling. Therefore, for many situations in practice where it is desired to obtain a good solution quickly, RS may not be able to fulfill the objective.

To make sure that RS being with suitable solutions the algorithm must be evolved to decide the following features:

High efficiency is required for the desired search algorithm. More specifically, the emphasis of the search algorithm should be on finding a better operating point within the limited time frame instead of seeking the strictly global optimum.

High dimension is another feature in solve problems. High-dimensional optimization problems are usually much more difficult to solve than low-dimensional problems because of "curse of dimensionality".

6 -Heuristic Search Algorithms

In this section, we can saw that many of the problems that fall within the purview of artificial intelligence are too complex to be solved by direct techniques; rather they must be attacked by appropriate search methods armed with whatever direct techniques are available to guide the search. These methods are all varieties of heuristic search. They can be described independently any particular task or problem domain. But when applied to Particular problems, their efficacy is highly dependent on the way they exploit domain-specific knowledge since in and of themselves they are unable to overcome the combinatorial explosion to which search processes are so vulnerable. For this reason, these techniques are often called weak methods. Although a realization of the limited effectiveness of these weak methods to solve hard problems by themselves has been an important result that emerged from the last decades of AI research, these techniques continue to provide the framework into which domain-specific knowledge can be placed, either by hand or as a result of automatic learning.

Hill climbing is a variant of generate-and-test in which feedback from the test procedure is used to help the generator decide which direction to move in the search space. In a pure generate-and-test procedure, the test function responds with only a yes or no. but if the test function is augmented with a heuristic function that provide an estimate of how close a given is to a goal state. This is particularly nice because often the computation of the heuristic function can be done at almost no cost at the same time that the test for a solution is being performed. Hill climbing is often used when a good heuristic function is available for

evaluating states but when no other useful knowledge is available. For example, suppose you are in an unfamiliar city without a map and you want to get downtown. You simply aim for the tall buildings. The heuristic function is just distance between the current location and the location of the tall buildings and the desirable states are those in which this distance is minimized.

Now let us discuss a new heuristic method called **best first search**, which is a way of combining the advantages of both depth-first and breadth-first search into a single method.

The actual operation of the algorithm is very simple. It proceeds in steps, expanding one node at each step, until it generates a node that corresponds to a goal state. At each step, it picks the most promising of the nodes that have so far been generated but not expanded. It generates the successors of the chosen node, applies the heuristic function to them, and adds them to the list of open nodes, after checking to see if any of them have been generated before. By doing this check, we can guarantee that each node only appears once in the graph, although many nodes may point to it as a successors. Then the next step begins.

The figure (3) bellow illustrates the hill climbing steps algorithm as it described in tree data structure.

Also the figure (4) bellow shows the steps of the best first search algorithm on a given tree as an assumption search space.

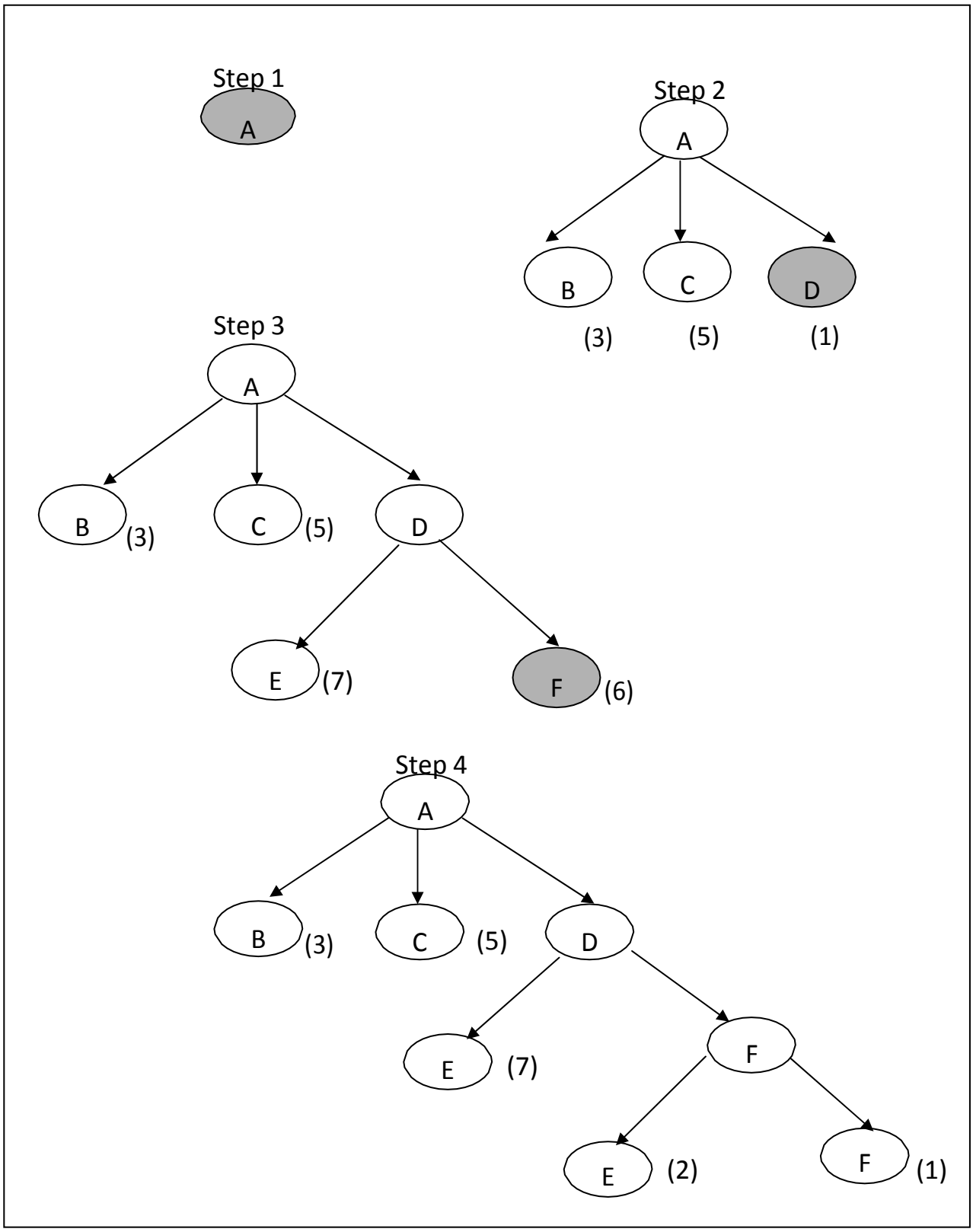


Figure (3) Hill Climbing Search Tree

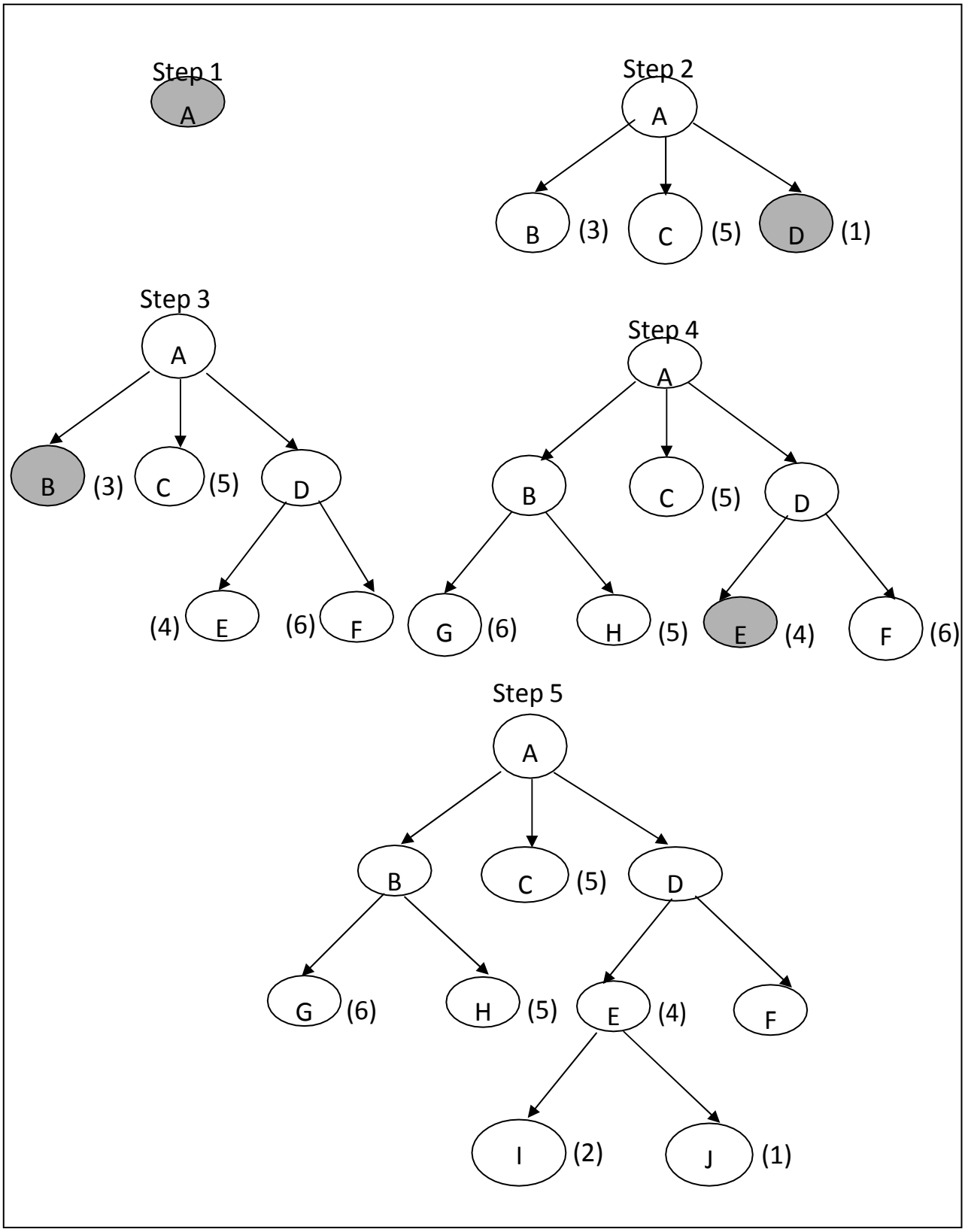


Figure (4) Best First Search Tree

The first advance approach to the best first search is known as **A-search algorithm**. A algorithm is simply define as a best first search plus specific function. This specific function represent the actual distance (levels) between the initial state and the current state and is denoted by g(n). A notice will be mentioned here that the same steps that are used in the best first search are used in an A algorithm but in addition to the g(n) as follow;

F(n) h(n) + g(n) where h(n) is a heuristic function. The second advance approach to the best first search is known as **A*-search algorithm**. A* algorithm is simply define as a best first search plus specific function. This specific function represent the actual distance (levels) between the current state and the goal state and is denoted by h(n). This algorithm will be described in detail in the next section of this chapter.

But just as it was necessary to modify our search procedure slightly to handle both maximizing and minimizing players, it is also necessary to modify the branch-and-bound strategy to include two bounds, one for each of players. This modified strategy is called alpha-beta pruning. It requires the maintenance of two threshold, one representing a lower bound on the value that a maximizing node may ultimately be assigned (we call this alpha) and another representing an upper bound on the value that minimizing node may be assigned (this we call beta).

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. But, of course, if the best path were

known in advance so that it could be guaranteed to be examined first, we would not need to bother with the search process. If, however, we knew how effective the pruning technique is in the perfect case, we would have an upper bound on its performance in other situations. It is possible to prove that if the nodes are perfectly ordered, then the number of terminal nodes considered by a search to depth *d* using alpha-beta pruning is approximately equal to twice the number of terminal nodes generated by a search to depth *d*/2 without alpha-beta. A doubling of the depth to which the search can be pursued is a significant gain. Even though all of this improvement cannot typically be realized, the alpha-beta technique is a significant improvement to the minimax search procedure.

7- A* Search: minimizing the total estimated solution cost

The most widely-known form of best-first search is called A* search (pronounced "A-star search"). It evaluates nodes by combining g(n), the cost to reach the node, and h(n), the cost to get from the node to the goal:

f(n) = g(n) + h(n).

Since g(n) gives the path cost from the start node to node n, and h(n) is the estimated cost of the cheapest path from n to the goal, we have

f (n) = estimated cost of the cheapest solution through n.

Thus, if we are trying to find the cheapest solution, a reasonable thing to try first is the node with the lowest value of g(n) + h(n). It turns out that this strategy is more than just reasonable: provided that the heuristic function h(n) satisfies certain conditions, A* search is both complete

and optimal. The optimality of A* is straightforward to analyze if it is used with TREE-SEARCH. In this case, A* is optimal if h(n) is an admissible heuristic-that is, provided that h(n) never overestimates the cost to reach the goal. Admissible heuristics are by nature optimistic, because they think the cost of solving the problem is less than it actually is. Since g(n) is the exact cost to reach n, we has immediate consequence that f(n) never overestimates the true cost of a solution through n.

7.1 A*- Search Algorithm

As we mentioned before, the A* algorithm to be discussed shortly is a complete realization of the best first algorithm that takes into account these issues in detail. The following definitions, however, are required for representing the A* algorithm. These are in order.

Definition 3.1: A node is called open if the node has been generated and the h'(x) has been applied over it but it has not been expanded yet.

Definition 3.2: A node is called closed if it has been expanded for generating offsprings.

In order to measure the goodness of a node in A* algorithm, we require two cost functions:

Heuristic cost.

Generation cost.

The heuristic cost measures the distance of the current node x with respect to the goal and is denoted by h(x). The cost of generating a node x, denoted by g(x), on the other hand measures the distance of node x with respect to the starting node in the graph. The total cost

function at node x, denoted by f(x), is the sum of g(x) plus h(x). The generation cost g(x) can be measured easily as we generate node x through a few slate transitions. For instance, if node x was generated from the starting node through m state transitions, the cost g(x) will be proportional to m (or simply m). But how does one evaluate the h(x)? It may be recollected that h(x) is the cost yet to be spent to reach the goal from the current node x. Obviously, any cost we assign as h(x) is through prediction. The predicted cost for h(x) is generally denoted by h'(x). Consequently, the predicted total cost is denoted by f(x), where:

f '(x) = g(x) + h'(x).

7.2 A* Procedure

Here are the basic steps that are considered to implement the A* procedure to solve problems in an intelligent manner:

1. Operations on states generate children of the state currently under examination.
2. Each new state is checked to see whether it has occurred before thereby preventing loops.
3. Each state n is given an f value equal to the sum of its depth in the search space g(n) and a heuristic estimate of its distance to a goal h(n).
4. States on open are sorted by their f examined or a goal.
5. As an implementation point, the algorithm's can be improved through maintenance of perhaps as heaps or leftist trees.

8- The Alpha-Beta Search Algorithm

The idea for alpha-beta search is simple: rather than searching the entire space to the ply depth, alpha-beta search proceeds in a depth-first fashion. Two values, called alpha and beta, are created during the search. The alpha value associated with MAX nodes, can never decrease, and the beta value associated with MIN nodes, can never increase. Two rules for terminating search, based on alpha and beta values, are:

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 ancestors.
2. Search can be stopped below any MAX node having an alpha value greater than or equal to the beta value of any of its MIN node ancestors.

Alpha-beta pruning thus expresses a relation between nodes at ply n and nodes at ply n + 2 under which entire subtrees rooted at level n + 1 can be eliminated from consideration. Note that the resulting backed-up value is identical to the minimax result and the search saving over minimax is considerable. With fortuitous ordering states in the search space, alpha-beta can effectively double the depth of the search considered with a fixed space/time computer commitment. If there is a particular unfortunate ordering, alpha-beta searches no more of the space than normal minimax; however, the search is done in only one pass.

8.1 The Alpha-Beta Cutoff Procedure (Tree Pruning)

$$c(n) = M(n) – O(n)$$

Where M(n) = number of my possible winning lines.

Now, we will discuss a new type of algorithm, which does not require expansion of the entire space exhaustively. This algorithm is referred to as alpha-beta cutoff algorithm. In this algorithm, two extra ply of movements are considered to select the current move from alternatives. Alpha and beta denote two cutoff levels associated with MAX and MIN nodes. As it is mentioned before the alpha value of MAX node cannot decrease, whereas the beta value of the MIN nodes cannot increase. But how can we compute the alpha and beta values? They are the backed up values up to the root like MINIMAX. There are a few interesting points that may be explored at this stage. Prior to the process of computing MAX / MIN of the backed up values of the children, the alpha-beta cutoff algorithm estimates e(n) at' all fringe nodes n. Now, the values are estimated following the MINIMAX algorithm. Now, to prune the unnecessary paths below a node, check whether:

- The beta value of any MIN node below a MAX node is less than or equal to its alpha value. If yes. prune that path below the MIN node.
- The alpha value of any MAX node below a MIN node exceeds the beta value of the MIN node. if yes prune the nodes below the MAX node.

Based on the above discussion, we now present the main steps in the α-β search algorithm.

1. Create a new node, if it is the beginning move, c1seexpand the existing tree by depth first manner. To make a decision about the selection of a move at depth d, the tree should be expanded at least up to a depth (d + 2).
2. Compute e(n) for all leave (fringe) nodes n in the tree.
3. Computer α_{min} (for max nodes) and β_{max} values (for min nodes) at the ancestors of the fringe nodes by the following guidelines. Estimate the minimum of the values (e or α) possessed by the children of a MINIMIZER node N and assign it its β_{max} value. Similarly, estimate the maximum of the values (e or β) possessed by the children of a MAXIMIZER node N and assign it its α_{min} value.
4. If the MAXIMIZER nodes already possess α_{min} values, then their current α_{min} value = Max (α_{min} value, $\alpha_{min,}$); on tile other hand, if the MANIMIZER nodes already possess β_{max} values, then their current β_{max} value = MIN (β_{max} value, β_{max}).
5. If the estimated β_{max} value of a MINIMIZER node N is less than the α_{min} value of its parent MAXIMIZER node N' then there is no need to search below the node MINIMIZER node N. Similarly, if the α_{min} value of a MAXIMIZER node N is more than the β_{max} value of its parent node N then there is no need to search below node N.

1. Introduction to Expert Systems

Expert systems are computer programs that are constructed to do the kinds of activities that human experts can do such as design, compose, plan, diagnose, interpret, summarize, audit, give advice. The work such a system is concerned with is typically a task from the worlds of business or engineering/science or government.

Expert system programs are usually set up to operate in a manner that will be perceived as intelligent: that is, as if there were a human expert on the other side of the video terminal.

A characteristic body of programming techniques give these programs their power. Expert systems generally use automated reasoning and the so-called weak methods, such as search or heuristics, to do their work. These techniques are quite distinct from the well-articulated algorithms and crisp mathematical procedures more traditional programming.

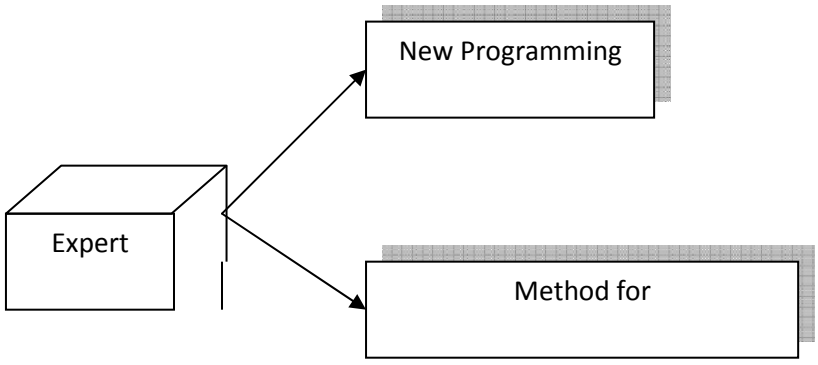


Figure (1) the vectors of expert system development

As shown in Figure(1), the development of expert systems is based on two distinct, yet complementary, vectors:

a. New programming technologies that allow us to deal with knowledge and inference with ease.

b. New design and development methodologies that allow us to effectively use these technologies to deal with complex problems.

The successful development of expert systems relies on a well-balanced approach to these two vectors.

2. Expert System Using

Here is a short nonexhaustive list of some of the things expert systems have been used for:

- To approve loan applications, evaluate insurance risks, and evaluate investment opportunities for the financial community.
- To help chemists find the proper sequence of reactions to create new molecules.
- To configure the hardware and software in a computer to match the unique arrangements specified by individual customers.
- To diagnose and locate faults in a telephone network from tests and trouble reports.
- To identify and correct malfunctions in locomotives.
- To help geologists interpret the data from instrumentation at the drill tip during oil well drilling.
- To help physicians diagnose and treat related groups of diseases, such as infections of the blood or the different kinds of cancers.

- To help navies interpret hydrophone data from arrays of microphones on the ocean floor that are used t\u the surveillance of ships in the vicinity.
- To figure out a chemical compound's molecular structure from experiments with mass spectral data and nuclear magnetic resonance.
- To examine and summarize volumes of rapidly changing data that are generated too last for human scrutiny, such as telemetry data from landsat satellites.

Most of these applications could have been done in more traditional ways as well as through an expert system, but in all these cases there were advantages to casting them in the expert system mold.

In some cases, this strategy made the program more human oriented. In others, it allowed the program to make better judgments.

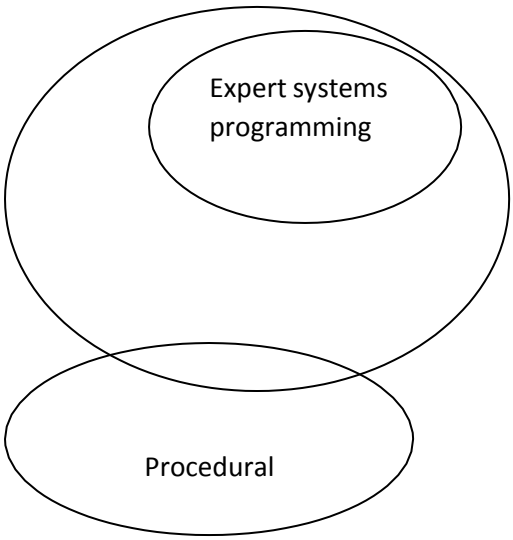
In others, using an expert system made the program easier to maintain and upgrade.

3. Expert Systems are Kind of AI ftrograms

Expert systems occupy a narrow but very important corner of the entire programming establishment. As part of saying what they are, we need to describe their place within the surrounding framework of established programming systems.

Figure(2) shows the three programming styles that will most concern us. Expert systems are part of a larger unit we might call AI (artificial intelligence) programming. Procedural programming is what everyone learns when they first begin to use BASIC or PASCAL or

FORTRAN. Procedural programming and A.I programming are quite different in what they try to do and how they try to do it.



Figure(2) three kinds of programming

In traditional programming (procedural programming), the computer has to be told in great detail exactly what to do and how to do it. This style has been very successful for problems that are well defined. They usually are found in data processing or in engineering or scientific work.

AI programming sometimes seems to have been defined by default, as anything that goes beyond what is easy to do in traditional procedural programs, but there are common elements in most AI programs. What characterizes these kinds of programs is that they deal with complex problems that are often poorly understood, for which there is no crisp algorithmic solution, and that can benefit from some sort of symbolic reasoning.

There are substantial differences in the internal mechanisms of the computer languages used for these two sorts of problems. Procedural programming focuses on the use of the assignment statement (" = " or ":-") for moving data to and from fixed, prearranged, named locations in memory. These named locations are the program variables. It also depends on a characteristic group of control constructs that tell the computer what to do. Control gets done by using

if-then-else	goto
do-while	procedure calls
repeat-until	sequential execution (as default)

AI programs are usually written in languages like Lisp and Prolog. Program variables in these languages have an ephemeral existence on the stack of the underlying computer rather than in fixed memory locations. Data manipulation is done through pattern matching and list building. The list techniques are deceptively simple, but almost any data structure can be built upon this foundation. Many examples of list building will be seen later when we begin to use Prolog. AI programs also use a different set of control constructs. They are :

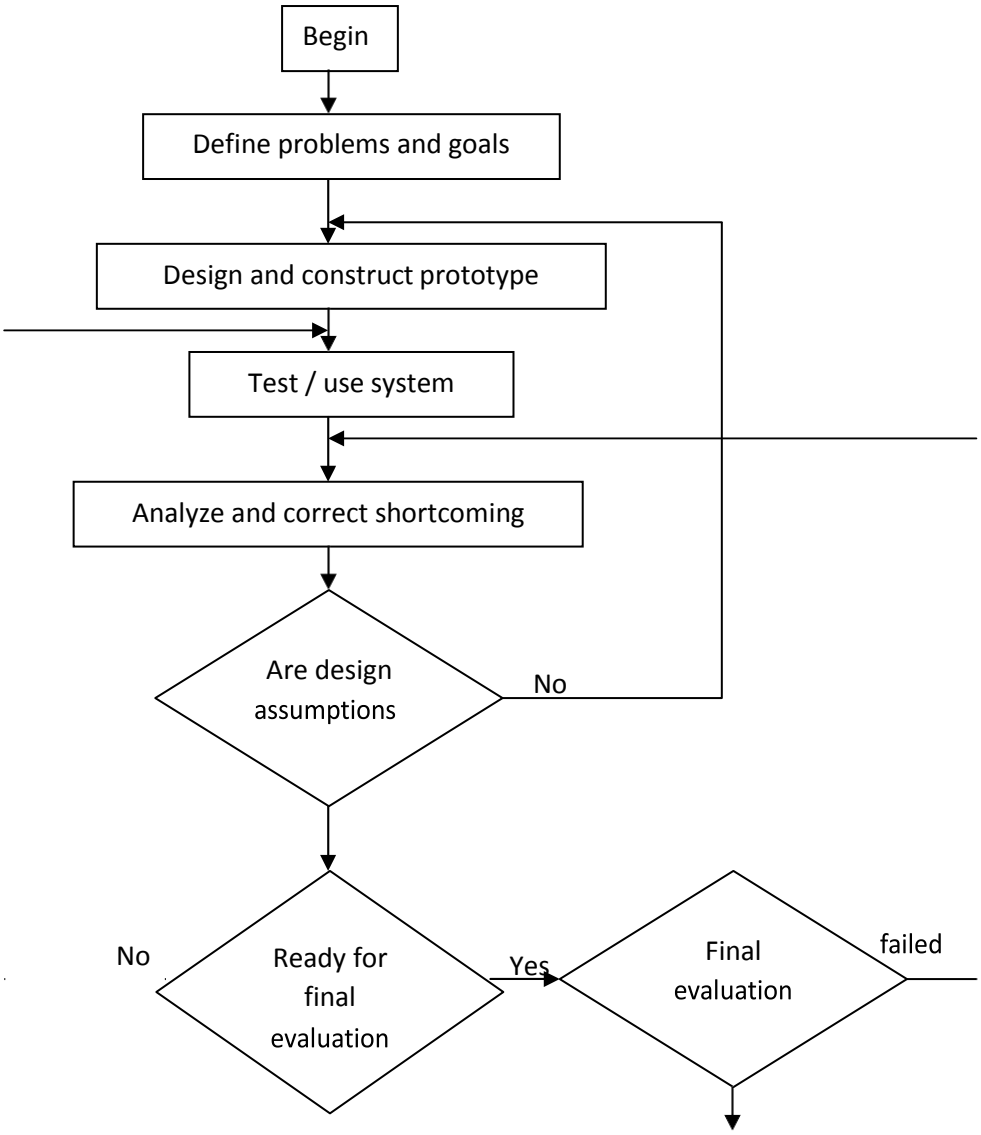
procedure calls

sequential execution

recursion

4. Expert System, Development Cycle

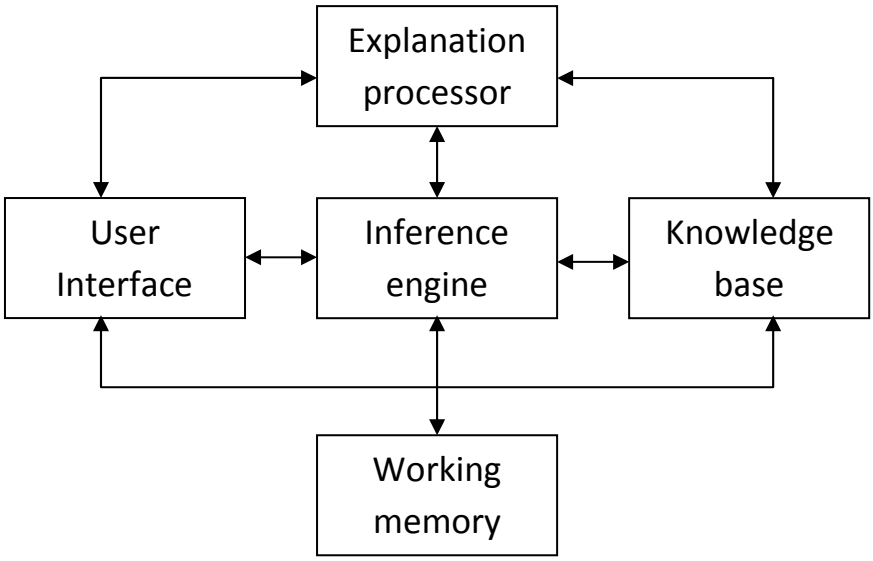
The explanation mechanism allows the program to explain its reasoning to the user, these explanations include justification for the system's conclusions, explanation of why the system needs a particular piece of data. Why questions and How questions. Figure (3) below shows the exploratory cycle for rule based expert system.



Figure(3) **The exploratory cycle for expert system**

5. Expert System Architecture and Components

The architecture of the expert system consists of several components as shown in figure (4) below:



***Figure(4)Expert system* architecture**

5.1. User Interface

The user interacts with the expert system through a user interface that make access more comfortable for the human and hides much of the system complexity. The interface styles includes questions and answers, menu-driver, natural languages, or graphics interfaces.

5.2. Explanation ftrocessor

The explanation part allows the program to explain its reasoning to the user. These explanations include justifications for the system's conclusion (HOW queries), explanation of why the system needs a particular piece of data (WHY queries).

5.3. Knowledge Base

The heart of the expert system contains the problem solving knowledge (which defined as an original collection of processed information) of the particular applications, this knowledge is represented in several ways such as if-then rules form.

5..4 Inference Engine

The inference engine applies the knowledge to the solution of actual problems. It s the interpreter for the knowledge base. The inference engine performs the recognize act control cycle.

The inference engine consists of the following components:-

1. Rule interpreter.
2. Scheduler
3. HOW process
4. WHY process
5. knowledge base interface.

5.5. Working Memory

It is a part of memory used for matching rules and calculation. When the work is finished this memory will be raised.

6. Systems that Explain their Actions

An interface system that can explain its behavior on demand will seem much more believable and intelligent to its users. In general, there are two things a user might want to know about what the system is doing. When the system asks for a piece of evidence, the user might want to ask,

"Why do you want it?"

When the system states a conclusion, the user will frequently want to ask,

"How did you arrive at that conclusion?"

This section explores simple mechanisms that accommodate both kinds of questioning. HOW and WHY questions are different in several rather obvious ways that affect how they can be handled in an automatic reasoning program. There are certain natural places where these questions are asked, and they are at opposite ends of the inference tree. It is appropriate to let the user ask a WHY question when the system is working with implications at the bottom of the tree; that is: when it will be necessary to ask the user to supply data.

The system never needs to ask for additional information when it is working in the upper parts of the tree. These nodes represent conclusions that the system has figured out. rather than asked for. so a WHY question is not pertinent.

To be able to make the conclusions at the top of the tree, however, is the purpose for which all the reasoning is being done. The system is trying to deduce information about these conclusions. It is appropriate to ask a HOW question when the system reports the results of its reasoning about such nodes.

There is also a difference in timing of the questions. WHY questions will be asked early on and then at unpredictable points all throughout the reasoning. The system asks for information when it discovers that it needs it. The. time for the HOW questions usually comes at the end when all the reasoning is complete and the system is reporting its results.

References

1-www.uotiq.org/dep-cs

2-http://www-formal.stanford.edu/jmc/whatisai/

3-https://en.wikipedia.org/wiki/Artificial_intelligence