University of Technology

Computer Science Department.

الجامعـــــة التكنولوجيـــــة

قســم علوم الحاسوب

ARTIFICIAL INTELLIGENCE

3nd Class

Mustafa Jasim Hadi

2017-2018

Introduction to Prolog Language: More examples in → Appendix-A

1.1What is Prolog?

Prolog (programming in logic) is one of the most widely used programming languages in artificial intelligence research.

Programming languages are of two kinds:

- **Procedural** (BASIC, ForTran, C++, Pascal, Java);
- **Declarative** (LISP, Prolog, ML).

In **procedural programming**, we tell the computer **how** to solve a problem, but in **declarative programming**, we tell the computer **what** problem we want solved. One makes statements about facts and relations among objects, and provides a set of rules (as predicate calculus implications) for making inference.

1.2 What is Prolog used for?

Good at

- Grammars and Language processing,
- Knowledge representation and reasoning,
- Unification,
- Pattern matching,
- Planning and Search.

By following this course, you will learn how to use Prolog as a programming language to solve practical problems in computer science and artificial intelligence. You will also learn how the Prolog interpreter actually works. The latter will include an introduction to the logical foundations of the Prolog language.

1.3 Prolog Terms

The central data structure in Prolog is that of a term. There are terms of four kinds: atoms, numbers, variables, and compound terms. Atoms and numbers are sometimes grouped together and called atomic terms.

Everything in a Prolog program both the program and the data it manipulates is build from Prolog terms.

1.3.1 Constants

Integers, real numbers, or an atom. Any name that starts with a lowercase letter (followed by zero or more additional letters, digits, or underscores) is an atom. Atoms look like variables of other languages, but are treated as constants in Prolog. Sequences of most non-alphanumeric characters (+, *, -, etc.) are also atoms.

1.3.2 Variables

A variable is any name beginning with an uppercase later or an underscore, followed by zero or more additional letters, digits, or underscores.

For example:

X, Y, Variable, _tag, X_526, and List, List24, _head, Tail, _input and Output are all Prolog variables.

1.4 Basic Elements of Prolog

There are only three basic constructs in Prolog: **Facts**, **Rules**, and **Queries**. A collection of facts and rules is called a knowledge base (or a database) and Prolog programming is all about writing knowledge bases. That is, Prolog programs simply are knowledge bases, collections of facts and rules which describe some collection of relationships that we find interesting.

- *Some are always true (facts):*

  father(john, jim).
- *Some are dependent on others being true (rules):*

  parent(Person1, Person2) :- father(Person1, Person2).
- To run a program, we ask questions about the database.

1.4.1 Example of Facts Database:

John is the father of Jim.	father(john, jim).
Jane is the mother of Jim.	mother(jane, jim).
Jack is the father of John.	father(jack, john).

1.4.2 Example of Rules Database:

- Person 1 is a parent of Person 2 **if** Person 1 is the father of Person 2 **or** Person 1 is the mother of Person 2.

parent(Person1, Person2):- father(Person1, Person2).

parent(Person1, Person2):- mother(Person1, Person2).

- Person 1 is a grandparent of Person 2 **if** some Person 3 is a parent of Person 2 **and** Person 1 is a parent of Person 3.

  grandparent(Person1, Person2):- parent(Person3, Person2), parent(Person1, Person3).

1.4.3 Example questions:

- Who is Jim's father? ?- father(Who, jim).
- Is Jane the mother of Fred? ?- mother(jane, fred).
- Is Jane the mother of Jim? ?- mother(jane, jim).
- Does Jack have a grandchild? ?- grandparent(jack, _)

1.5 Clauses

Predicate definitions consist of *clauses*. It is an individual definition (whether it be a fact or rule).

mother(jane,alan). = Fact

parent(P1,P2):- mother(P1,P2). = Rule

A clause consists of a *head* and sometimes a *body*. Facts don't have a body because they are always true.

A predicate head consists of a *predicate name* and sometimes some *arguments* contained within brackets and separated by commas.

mother(jane, alan).

1.6 Arithmetic Operators

Operators for arithmetic and value comparisons are built-in to Prolog.

- *Comparisons:* <, >, <=, >=, = (equals),< > (not equals)
- *Mathematical precedence is preserved:* /, *, +, - Can make compound sums using round brackets for example X = (5+4)*2 then X=18.

1.7 Tests within clauses

These operators can be used within the body of a clause to manipulate values:

sum(X,Y,Sum):- Sum = X+Y.

Goal: sum(3,5,S)

Output: S=8

sum(X,Y):-Sum=X+Y, write(Sum).

Goal: sum(3,5)

Output: 8

We can write the rule without arguments for example:

Sum:-readint(X),readint(Y),Sum=X+Y, write(Sum).

Goal: sum

Output: 8

Also, these operators can be used to distinguish between clauses of a predicate definition:

bigger(N,M):- N < M, write(‘The bigger number is ‘), write(M).

bigger(N,M):- N > M, write(‘The bigger number is ‘), write(N).

bigger(N,M):- N = M, write(‘Numbers are the same‘).

Goal: bigger(6,7)

Output: The bigger number is 7

2.1 Recursion

The recursion in any language is a function that can call itself until the goal has been succeed. In Prolog, recursion appears when a predicate contain a goal that refers to itself. There are two types of recursion.

1. **Tail Recursion**: is recursion in which the recursive call is the last subgoal of the last clause. That is, in tail recursion the recursive call is always made just before the procedure exits: the last step. For example:

   tail_recursion(X):- b, c, tail_recursion(Y).

2. **Non Tail Recursion:** when a recursive call is not last we don't have tail recursion. If you think about it, tail recursive calls can be implemented very efficiently because they can reuse the current stack frame or activation record rather than pushing a new one, filling it with data, doing some computation, and popping it. So tail recursion is often a fast option. For example:

   nontail_recursion(X):- b, nontail_recursion(X), c.

2.2 Pattern Matching

Prolog uses *unification* to match variables to values. An expression that contains variables like *X+Y*Z* describes a pattern where there are three blank spaces to fill in named X, Y, and Z. The expression *1+2*3* has the same structure (pattern) but no variables. If we input this query

X+Y*Z=1+2*3.

then Prolog will respond that X=1, Y=2, and Z=3! The pattern matching is very powerful because you can match variables to expressions like this

X+Y=1+2*3.

and get X=1 and Y=2*3! You can also match variable to variables:

X+1+Y=Y+Z+2.

This sets X=Y=2 and Z=1!

Example 1:

a(1).

a(2).

a(4).

b(2).

b(3).

d(X,Y):-a(X),b(Y), X=4.

Goal: d(X,Y)

X=1, Y=2, 1=4 Fail

X=1, Y=3, 1=4 Fail

X=2, Y=2, 2=4 Fail

X=2, Y=3, 2=4 Fail

X=4, Y=2, 4=4 True

X=4, Y=3, 4=4 True

Output: 2 solutions X=4, Y=2 and X=4 Y=3.

Goal: d(A,2)

X=1, Y=2, 1=4 Fail

X=2, Y=2, 2=4 Fail

X=4, Y=2, 4=4 True

Output: 1solution A=4.

Goal: d(2,2)

Output: no solution.

2.3 Backtracking

2.3.1 Cut

Up to this point, the Prolog *cut* adds an 'else' to Prolog, we have worked with Prolog's backtracking. Sometimes it is desirable to selectively turn off backtracking. Prolog provides a predicate that performs this function. It is called the **cut**, represented by an exclamation point (!). The **cut** effectively tells Prolog to freeze all the decisions made so far in this predicate. That is, if required to backtrack, it will automatically fail without trying other alternatives.

Example 1:

a(1).

a(2).

a(4).

b(2).

b(3).

d(X,Y):-a(X),b(Y), X=4,!.

Goal: d(X,Y)

X=1, Y=2, 1=4 Fail

X=1, Y=3, 1=4 Fail

X=2, Y=2, 2=4 Fail

X=2, Y=3, 2=4 Fail

X=4, Y=2, 4=4 True

Output: 1 solutions X=4, Y=2.

Example 2:

a(1).

a(2).

a(4).

b(2).

b(3).

d(X,Y):-a(X),**!**,b(Y), X=4.

Goal: d(X,Y)

X=1, Y=2, 1=4 Fail

X=1, Y=3, 1=4 Fail

Output: no solutions found.

2.3.2 Fail

The **fail** predicate is provided by Prolog. When it is called, it causes the failure of the rule. And this will be forever; nothing can change the statement of this predicate.

Example of using fail in the program.

clauses

a(1).

a(2).

a(3).

a(4).

Begin:-a(X),write(X),fail.

Goal: Begin

Output: 1234 NO

2.3.3 Passing Back Answers

To report back answers we need to put an un instantiated variable in the query, and then, instantiate the answer to that variable when the query succeeds, Finally, pass the variable all the way back to the query.

bigger(X,Y,Answer):- X>Y, Answer = X.

bigger(X,Y,Answer):- X=<Y, Answer = Y.

Goal: bigger(6,4,Answer),bigger(Answer,5,New_answer).

Output: New_answer=6

Built-in mathematical functions

Prolog has a full range of built-in mathematical functions and predicates that operate on integral and real values.

X mod Y Returns the remainder (modulos) of X divided by Y.
X div Y Returns the quotient of X divided by Y.
abs(X) If X is bound to a positive value val, abs(X) returns that value; otherwise, it returns -1 * val.
cos(X) The trigonometric functions require that X be bound to
sin(X) a value representing an angle in radians.
tan(X) Returns the tangent of its argument.
arctan(X) Returns the arc tangent of the real value to which X is bound.
exp(X) e raised to the value to which X is bound.
ln(X) Logarithm of X, base e.
log(X) Logarithm of X, base 10.
sqrt(X) Square root of X.

random(X) Binds X to a random real; 0 <= X < 1.
random(X, Y) Binds Y to a random integer; 0 <= Y < X.
round(X) Returns the rounded value of X. The result still being a real
trunc(X) Truncates X. The result still being a real
val(domain,X) Explicit conversion between numeric domains.

3.1 Complete Prolog Programs

domains

i=integer

predicates

counter(i)

sum(i,i,i).

sum_int(i,i,i).

fact(i,i,i).

power(i,i,i,i).

clauses

/* counter from 1-10*/

counter(10):-!.

counter(X):-write(X), X1=X+1, counter(X1).

Goal: counter(1)

Output: 1 2 3 4 5 6 7 8 9

/*summation of 10 numbers*/

sum(10,S,S):-!.

```
sum(X,Y,S):-Y1=Y+X,X1=X+1,sum(X1,Y1,S),!.
```

Goal: sum(1,0,M)

Output: ?

/* summation of 10 given integer numbers*/

```
sum_int(10,S,S):-!.
sum_int(X,Y,S):-readint(Z),Z>0, X1=X+1,Y1=Y+Z,sum_int(X1,Y1,S),!.
sum_int(X,Y,S):-X1=X+1,sum_int(X1,Y,S),!.
```

Goal: sum_int(1,0,M)

Output: ?

/* factorial program*/

```
fact(0,_,1):-!.
fact(1,F,F):-!.
fact(X,Y,F):-Y1=X*Y,X1=X-1,fact(X1,Y1,F),!.
```

Goal: fact(3,1,F)

Output: F=6

/*power program*/

```
power(_,0,P,P):-!.
power(X,Y,Z,P):-Z1=Z*X,Y1=Y-1,power(X,Y1,Z1,P),!.
```

Goal: power(5,2,1,P)

Output: P=25

4.1 Lists in Prolog

Lists are ordered sequences of elements that can have any length. Lists can be represented as a special kind of tree. A list is either empty, or it is a structure that has two components: the head **H** and tail **T**. List notation consists of the elements of the list separated by commas, and the whole list is enclosed in square brackets.

For example:

- [a] and [a,b,c], where a, b and c are symbols type.
- [1], [2,3,4] these are a lists of integer.
- [] is the atom representing the empty list.
- Lists can contain other lists. Split a list into its head and tail using the operation [X|Y].

4.2 Examples about Lists

1. p([1,2,3]).

p([the,cat,sat,[on,the,hat]]).

Goal: p([X|Y]).

Output:

X = 1 Y = [2,3] ;

X = the Y = [cat,sat,[on,the,hat]].

---

2. p([a]).

Goal: p([H | T]).

Output:

H = a, T = [].

---

3. p([a, b, c, d]).

Goal: p([X, Y | T]).

Output:

X = a, Y = b, T = [c, d].

---

4. P([[a, b, c], d, e]).

Goal: p(H,T)

Output:

H = [a, b, c], T = [d, e].

4.3 List Membership

- Member is possibly the most used user-defined predicate (i.e. you have to define it every time you want to use it!).
- It checks to see if a term is an element of a list.
  - it returns **yes** if it is.
  - and **fails** if it isn't.

```
member(X,[X|_]).
member(X,[_|Y]) :- member(X,Y).
```

- It 1st checks if the Head of the list unifies with the first argument.

  If yes then succeed.

  If no then fail first clause.
- The 2nd clause ignores the head of the list (which we know doesn't match) and recourses on the Tail.

Goal: member(a, [b, c, a]).

Output: Yes

Goal: member(a, [c, d]).

Output: No.

4.4 Print the contents of the list.

```
print([]).
print([i(X,Y)|T]):-write(X,Y),print(T).
```

Goal: print([3,4,5])

Output: 3 4 5

4.5 Find the maximum value of the list

```
list([H],H).
list([H1,H2|T],H1):-H1>H2,list([H1|T],_).
list([_,H2|T],H2):-list([H2|T],H2).
```

Goal: list([3,9,4,5],M)

Output: M=9

4.6 Append two lists

```
app([],L,L).
app([X|L1],L2,[X|L3]) :-app(L1,L2,L3).
```

Goal: app([3,4,5],[6,7,8],L)

Output: L=[3,4,5,6,7,8]

4.7 Write the contents of list inside at the given list

```
list([[]]).
list([[H|T]]):-write(H),list([T]).
list([H|T]):-list([H]),list(T).
```

Goal: list([[3,4,5],[6,7,8]])

Output: 3 4 5 6 7 8

4.8 Reveres the contents of the given list.

```
app([],X,X).
app([H|T1],X,[H|T]):-app(T1,X,T).
rev([X],[X]).
rev([H|T],L):-rev(T,L1),app(L1,[H],L).
```

Goal: rev([a,b,c,d],R)

Output: R=[d,c,b,a]

5.1 Tail and non tail Recursive Programs.

In general, tail-recursive programs are more efficient than non-tail-recursive programs.

Non-tail Summation of 10 integer number.

```
Sum-nontail(11,0).
Sum-nontail(X,S):-X1=X+1, Sum-nontail(X1,S1),S=S1+X.
```

Trace the above program with the goal ? Sum-nontail(4,S)

```
Sum-nontail(1,S):-X1=2, Sum-nontail(2,S1),S=S1+1.
Sum-nontail(2,S):-X1=3, Sum-nontail(3,S1),S=S1+2.
Sum-nontail(3,S):-X1=4, Sum-nontail(4,S1),S=S1+3.
```

Then ? Sum-nontail(4,S)

S=6.

Non-tail Factorial program.

```
Fact(0,1).
Fact(1,1).
Fact(X,Y):-X1=X-1,fact(X1,Y1),Y=X*Y1.
```

Non-tail Power program.

```
Power(_,0,1).
Power(X,Y,Z):-Y1=Y-1,power(X,Y1,Z1),Z=Z1+X.
```

Standard String Predicates

Prolog provides several standard predicates for powerful and efficient string manipulations. In this section, we summarize the standard predicates available for string manipulating and type conversion.

1. **str_len** (String,Length) (string,integer) (i,o): Determines the length of String. Succeeds if the length could be matched with Len.

   str_len("prolog",X)
   X=6.

2. **str_char** (String,Char) (string,char) (i,o) (o,i): Converts a string into a character or vice versa. If string is bound, it must have length 1 for the predicate to succeed.

   str_char("A",X)
   X='A'.

3. **str_int** (String,Int) (string,integer) (i,o) (o,i): Converts a string of one character to ASCII code or vice versa. If string is bound, it must contain a single number for the predicate to succeed.

   str_int("A",X)
   X=65 .

4. **char_int** (Char,Int) (char,integer) (i,o) (o,i): Converts a character to ASCII code or vice versa.

   char_int('A',X)
   X=65.

5. **isname** (String) (string) (i): Tests whether a string would match a prolog symbol.

   isname("s2") return YES.
   isname("4r") return NO.

6. **frontchar** (String,FrontChar,RestString) (string,char,string) (i,x,x) (x,i,i): Extracts the first character from a string, the remainder is matched with RestString.

   frontchar ("prolog",C,R)
   C='p', R="rolog".

7. **<u>fronttoken</u>** (String,Token,Rest) (string,string,string) (i,o,o) (i,i,o) (o,i,i): Skips all white space characters (blanks,tabs) and separates from the resulting string the first valid token. The remainder is matched with RestString. A valid token is either a variable or name 'A'...'Z','a'...'z','','0'...'9', a number '0'...'9' or a single character. It fails if String was empty or contained only whitespace.

   <u>fronttoken</u> (“complete prolog program”,T,R)
   T=”complete”, R=”prolog program”.

8. **<u>frontstr</u>** (StrLen,String,FrontStr,RestStr) (i,i,o,o): Extracts the first n characters from a string. This establishes a relation between String, Count, FronStr, and RestString, thus that String = FrontStr+RestString and str_len(FronStr,Count) is true. The String and Count arguments must be initalized.

   frontstr(3,”cdab 2000”,T,R)
   T=”cda”, R=”b 2000”.

9. **<u>concat</u>** (Str1,Str2,ResStr) (string,string,string) (i,i,o): Merges to strings to one by appending the second argument to the first.

   <u>concat</u> (“prolog”,”2011”,R)
   R=”prolog2011”.

Examples: Write prolog program to do the following:

1. Convert the string of words into a list of words:

```
split("",[]).
split(S,[H|T]):-fronttoken(S,H,R),split(R,T).
```

2. Count the number of words in the given string.

```
counts("",0).
counts(S,C):-fronttoken(S,_,R), counts(R,C1),C=C1+1.
```

3.Count the number of characters in the given string.

```
counts("",0).
counts(S,C):-frontchar(S,_,R), counts(R,C1),C=C1+1.
```

4. Reverse the given string.

```
rev("","").
 rev(Str,R_Str):-frontstr(1,Str,C,RemStr),rev(RemStr,L1),concat(L1,C,R_Str).
con([],"").
```

```
con([H|T],Str):-con(T,Str1),concat(H,Str1,Str).
```

5. Count the number of words that contain "tion" in the given list.

```
count(S):-frontstr(4,S,X,_),X="tion".
count(S):-frontchar(S,_,R),count(R).
set([],0).
set([H|T],L):-count(H),set(T,L1),L=L1+1,!.
set([_|T],L):-set(T,L).
```

Prolog Database

It uses to retrieve information from the database. The database predicate retrieves information from the database by matching its argument with any term stored in the database. Until the DATABASE keyword is implemented, we have to use this or clause to make a non-destructive database query. The database contains related predicates such as: **asserta**, **assertz**, **retract**, **retractall,** these predicates are standard prolog built-in as shown below:

1. **asserta()** adds **Term** to the beginning of the database. Term must should bound to an atom or a struct, but may contain free variables. The database entry is not removed upon backtracking. Database entries can only be removed by the use of the.
2. **assertz()** adds **Term** to the end of the database. Term must should bound to an atom or a struct, but may contain free variables. The database entry is not removed upon backtracking. Database entries can only be removed by the use of the **retract()** predicate.
3. **retract():** it retrieves information from the database by matching its argument with any term stored in the database. If it matches, the database entry is removed. Upon backtracking multiple solutions may occur. You may invoke indefinite loops by using retract(X),...,assertz(X),fail.construct.

   Different from standard prolog we may use this with an uninstantiated variable to remove all entries.

4. **retractall():** It succeeds once and remove all database entries that match Term on the way. Different from standard prolog we may use this with an uninstantiated variable to remove all entries.

To load and save dos file, the predicates **consult()** and **save()** are used as shown below:

1- consult (FileName): is used to read and parse the file given by **FileName**. The file may contain clauses and facts, separated by the dots. The current operator declarations are used. For example **consult("data.dat")**, where “data.dat” is dos file name.

2- save(FileName): is used to save information the file given by **FileName**. For example **save("data.dat"),** where “data.dat” is dos file name.

Examples: Write prolog program to perform the following:

1- Read person’s name S1, gender S2 and age I then save them into file “p.dat”.

Solution:

```
repeat.
repeat:-repeat.
a:-repeat,consult(“a1.dat”),readln(S1),readln(S2),readint(I),assertz(Per(S1,I,S2)),
I<=18,save(“p.dat”).
```

2-Read car’s name S, owner N and car’s model M then save them into file “c.dat”.

```
repeat.
repeat:-repeat.
b:-repeat,consult(“a2.dat”),readln(S),readln(N),readint(M),assertz(car(S,N,M)),
M<=1980,save(“c.dat”).
```

3- Count how many cars with “VOLVO” name?

```
c:-consult("c.dat"),findall(X,car(X,_,_),L),count(L,C),write(C).
count([],0).
count([H|T],X):-H="volvo",count(T,X1),X=X1+1.
count([_|T],X):-count(T,X).
```

4- Count how many cars with car name “TOYOTA” and owner name “ALI”?

```
d:-consult("c.dat"),findall(X,car("toyota",X,_),L),count(L,C),write(C).
count([],0).
count([H|T],X):-H="ali",count(T,X1),X=X1+1.
count([_|T],X):-count(T,X).
```

Artificial Intelligence: AI Programs in ➡ Appendix-B

What is Artificial Intelligence (AI)?

Artificial Intelligence (AI) is usually defined as the science of making computers do things that require intelligence when done by humans.

Some common terms in the searching issues

State:

State is a representation that an agent can find itself in.

State Space:

A state space is a graph whose nodes are the set of all states, and whose links are actions that take the agent from one state into another.

Search Tree:

A search tree is a tree in which the root node is the start state and has a reachable set of children.

Search Node:

A search node is a node in the search tree.

Goal:

A goal is a state that the agent is trying to reach.

Action:

An action is something that the agent can choose to do.

Branching Factor:

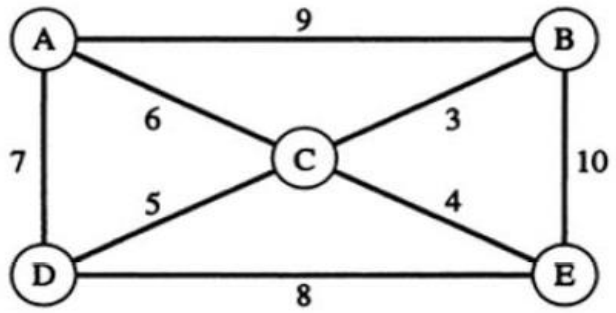
The branching factor in a search tree is the number of actions available to the agent.

Example: **Describe and give an example for the Travelling Salesman Problem (TSP) as a state space?**

Solution:

Given an undirected weighted graph, we should find a shortest tour (a shortest path in which every node (city) is visited exactly once, except that the initial and terminal nodes are the same).

Figure below shows an example of such a graph and its optimal solution. A, B, etc., are cities and the numbers associated with the links are the distances between the cities.



Optimal solution:

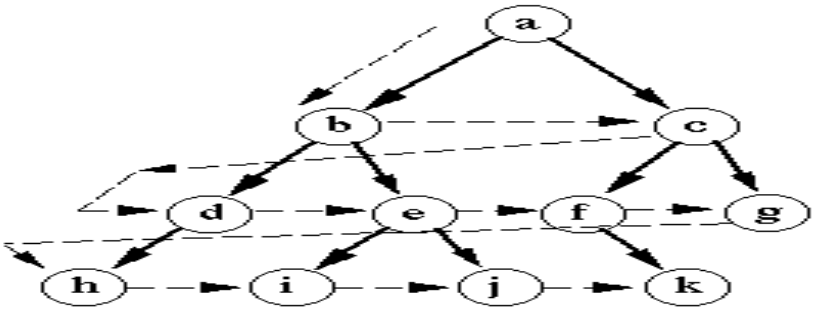
A —9— B —3— C —4— E —8— D —7— (A)

The total distance is 31 and represents an optimal solution.

Uninformed Search (Blind Search)

1-Breadth – First – Search

In breadth –first search, when a state is examined, all of its siblings are examined before any of its children. The space is searched level-by-level, proceeding all the way across one level before doing down to the next level.

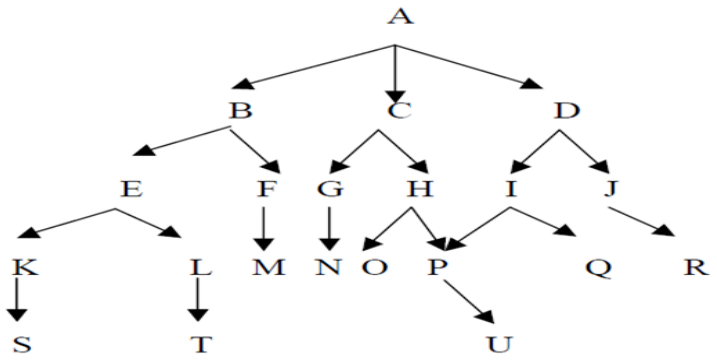


Breadth-first search

Breadth – first – search Algorithm

Begin
Open: = [start];
Closed: = [ ];
While open ≠ [ ] do
Begin
Remove left most state from open, call it x;
If x is a goal the return (success)
Else
Begin
Generate children of x;
Put x on closed;
Eliminate children of x on open or closed;
Put remaining children on right end of open
End
End
Return (failure)
End.

Example:



Breadth – first – search

1 – Open = [A]; closed = [ ].

2 – Open = [B, C, D]; closed = [A].

3 – Open = [C, D, E, F]; closed = [B, A].

4 – Open = [D, E, F, G, H]; closed = [C, B, A].

5 – Open = [E, F, G, H, I, J]; closed = [D, C, B, A].

6 – Open = [F, G, H, I, J, K, L]; closed = [E, D, C, B, A].

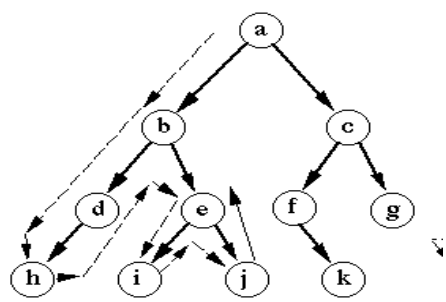
7 – Open = [G, H, I, J, K, L, M]; closed = [F, E, D, C, B, A].

8 – Open = [H, I, J, K, L, M, N,]; closed = [G, F, E, D, C, B, A].

9 – and so on until either U is found or open = [ ].

2-Depth – first – search

In depth – first – search, when a state is examined, all of its children and their descendants are examined before any of its siblings. Depth – first search goes deeper in to the search space whenever this is possible.



Depth-first search

<u>Depth – first – search Algorithm</u>

Begin
Open: = [start];
Closed: = [ ];
While open ≠ [ ] do
Remove leftmost state from open, call it x;
If x is a goal then return (success)
Else begin
Generate children of x;

Put x on closed;
Eliminate children of x on open or closed; put remaining children on left end of open end
End;
Return (failure)
End.

Informed Search (Heuristic Search)

A heuristic is a method that might not always find the best solution but is guaranteed to find a good solution in reasonable time. By sacrificing completeness it increases efficiency. Heuristic search is useful in solving problems which:-

• Could not be solved any other way.

• Solution takes an infinite time or very long time to compute.

• Heuristic search methods generate and test algorithms, from these methods are:-

1- Hill Climbing.

2- Best-First Search.

3- A and A* algorithm.

1- Hill Climbing

The idea here is that, you don't keep the big list of states around you just keep track of the one state you are considering, and the path that got you there from the initial state. At every state you choose the state leads you closer to the goal (according to the heuristic estimate), and continue from there.

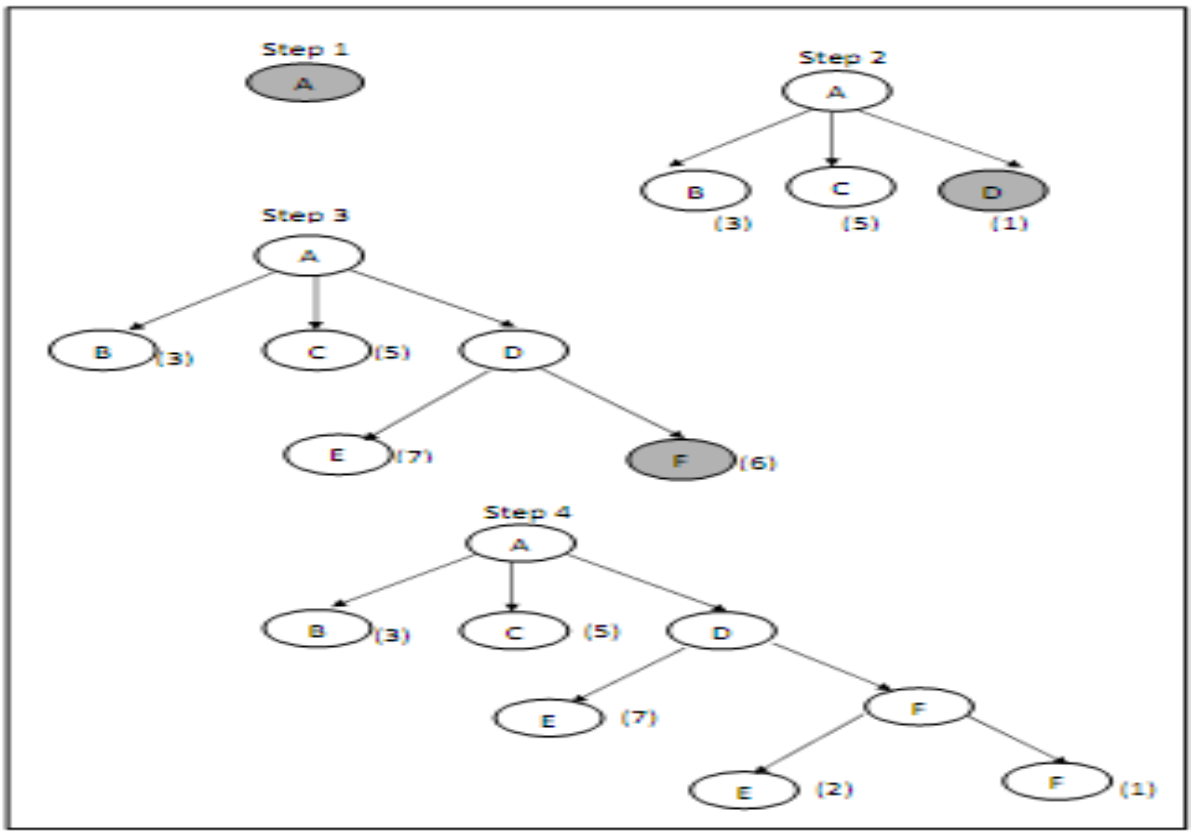
The name "Hill Climbing" comes from the idea that you are trying to find the top of a hill, and you go in the direction that is up from wherever

you are. This technique often works, but since it only uses local information.

Hill Climbing Algorithm

```
Begin
Cs=start state;
Open=[start];
Stop=false;
Path=[start];
While (not stop) do
{
if (cs=goal) then
return (path);
generate all children of cs and put it into open
if (open=[]) then
stop=true
else
{
x:= cs;
for each state in open do
{
compute the heuristic value of y (h(y));
if y is better than x then
x=y
}
if x is better than cs then
cs=x
else
stop =true;
}
}
return failure;
}
```

The figure bellow illustrates the hill climbing steps algorithm as it described in tree data structure. (Note: we assume that the best value is the smallest, in other problems we may assume the largest value is the best)



<u>Hill climbing Problems:-</u>

Hill climbing may fail due to one or more of the following reasons:-

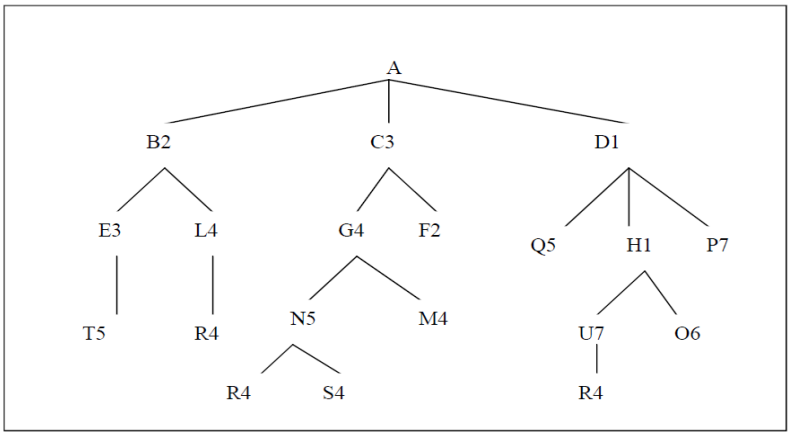
1- **A local maxima**: Is a state that is better than all of its neighbors but is not better than some other states.

2- **A Plateau**: Is a flat area of the search space in which a number of states have the same best value, on plateau it's not possible to determine the best direction in which to move.

3- **A ridge**: Is an area of the search space that is higher than surrounding areas, but that cannot be traversed by a single move in any one direction.

Example:

Searches for R4



Open	Close	X
A	_	A
C3 B2 D1	A	C3
G4 F2	A C3	G4
N5 M4	A C3	N5
R4 S4	A C3 G4 N5	R4

The solution path is : A-C3-G4-N5

2- Best-First-Search

Best-First-search 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 st eps, 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 thischeck, we can gua

rantee that each node only appears once in the graph, although many node s may point to it as a successors. Then the next step begins.

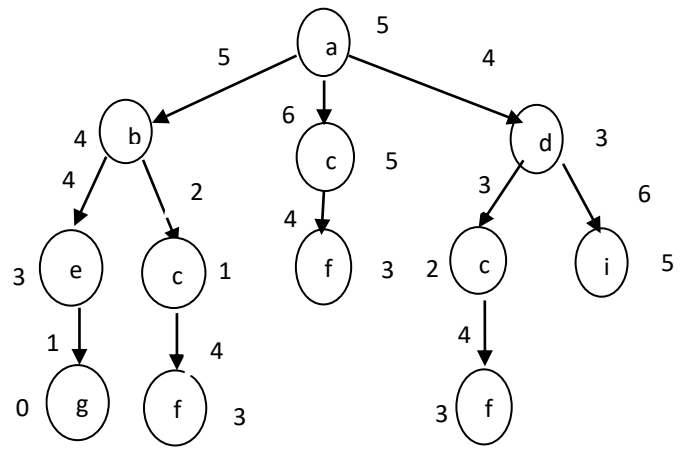
In Best-First search, the search space is evaluated according to a heuristic function. Nodes yet to be evaluated are kept on an OPEN list and those that have already been evaluated are stored on a CLOSED list. The OPEN list is represented as a priority queue, such that unvisited nodes can be queued in order of their evaluation function. The evaluation function f(n) is made from only the heuristic function (h(n)) as: f (n) = h(n) .

<u>Best-First-Search Algorithm</u>

```
{
Open:=[start];
Closed:=[];
While open ≠ [] do
{
Remove the leftmost from open, call it x;
If x= goal then
Return the path from start to x
Else
{
Generate children of x;
For each child of x do
Do case
The child is not already on open or closed;
{ assign a heuristic value to the child state ;
Add the child state to open;
}
The child is already on open:
If the child was reached along a shorter path than the state currently
on open then give the state on open this shorter path value.
The child is already on closed:
If the child was reached along a shorter path than the state currently
on open then
 {
Give the state on closed this shorter path value
Move this state from closed to open
 }
```

}
Put x on closed;
Re-order state on open according to heuristic (best value first)
}
Return (failure);
}

Example:

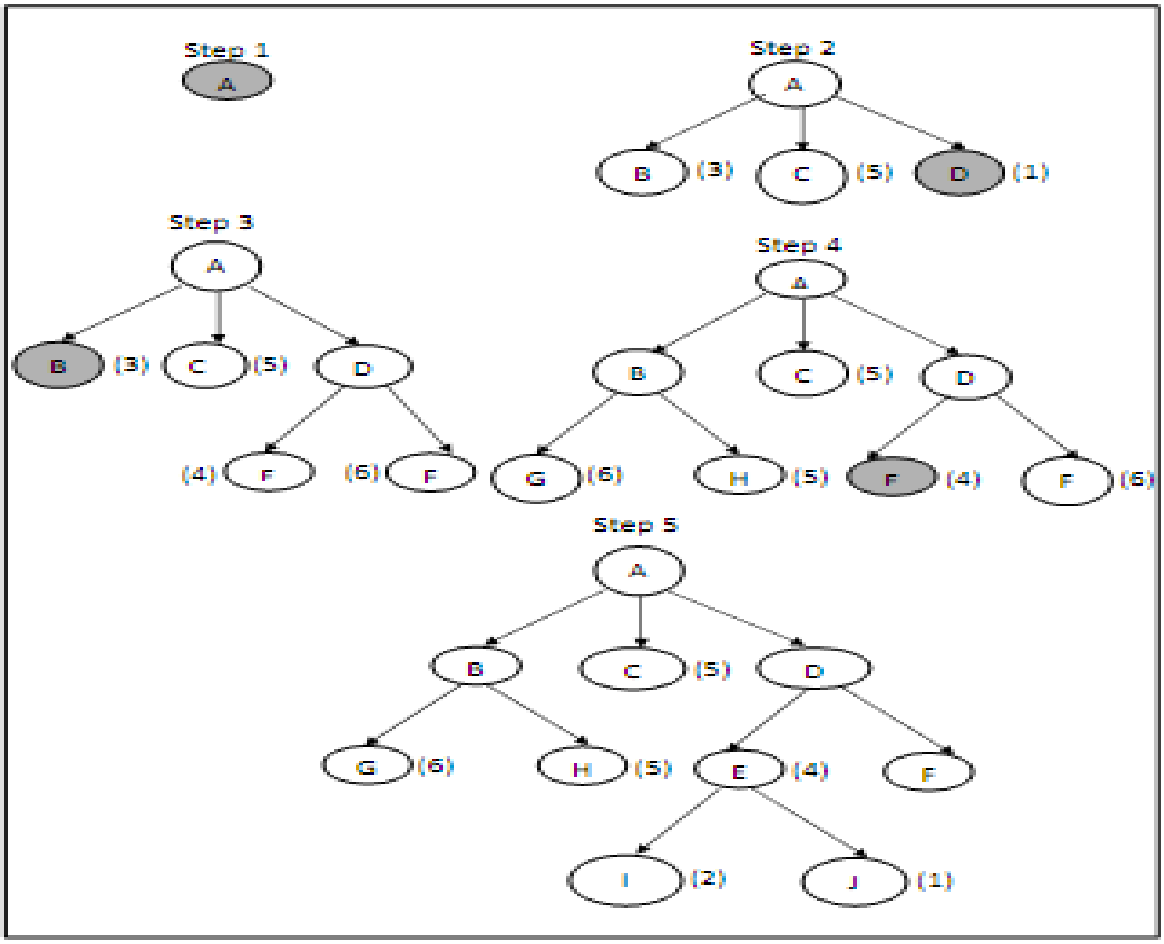


Open	Closed
[A5]	[]
[D3,B4,C5]	[A5]
[C2,B4,I5]	[A5,D3]
[F3,B4,I5]	[A5,D3,C2]
[B4,I5]	[A5,D3,C2,F3]
[C1,E3,I5]	[A5,D3,C2,F3,B4]
[E3,I5]	[A5,D3,F3,B4,C1]
[G0,I5]	[A5,D3,F3,B4,C1,E3]
	[A5,D3,F3,B4,C1,E3,G0]

The goal is found &the resulted path is:

A0 → D4 → F7 → B 16 → C 2 → E 6 → G1 =36

The figure bellow shows the steps of the best first search algorithm on a given tree as an assumption search space.



3- A - Search algorithm

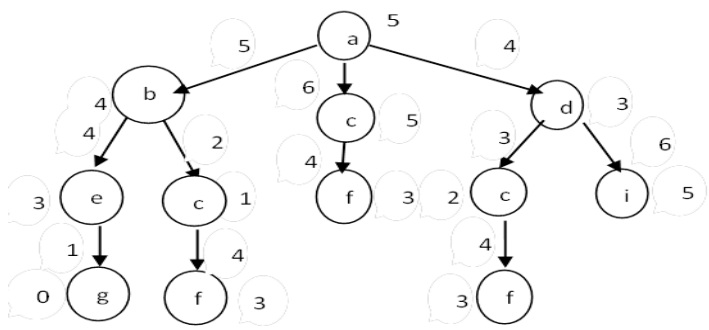
A-algorithm is simply define as a best first search plus specific function. This specific function represent the actual distance (levels) between the initial state and the current state and is denoted by g(n). A notice will be mentioned here that the same steps that are used in the best first search 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 estimate of the distance from state n to the goal.

g(n):- Measures the actual length of path from any state (n) to the start state.

Example:



Open	Closed
[A5]	[]
[D4,B5,C6]	[A5]
[C4,B5,I7]	[A5,D4]
[B5,F6,I7]	[A5,D4,C4]
[C3,E5,F6,I7]	[A5,D4,B5]
[E5,F6,I7]	[A5,D4,B5,C3]
[G3,F6,I7]	[A5,D4,B5,C3,E5]
	[A5,D4,B5,C3,E5,G3]

The goal is found &the resulted path is:

A0 → D4 →B9→C2→ E6→G1 =22

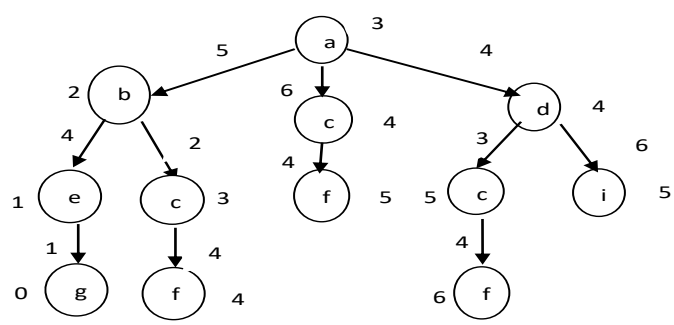
4- A-Star search algorithm

A* algorithm is simply define as a best first search plus specific function. This specific function represents the actual distance (levels) between the current state and the goal state and is denoted by h(n). It evaluates nodes by combining g(n), the cost to reach the node, an d 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. Example:



Open	Closed
[A3]	[]
[B7,D8,C10]	[A3]
[D8,E10,C10]	[A3,B7]
[E10,C10,I15]	[A3,B7,D8]
[G10,C10,I15]	[A3,B7,D8,E10]
	[A3,B7,D8,E10,G10]

The goal is found &the resulted path is:

A0 → B5 → D9 → E13 → G1 =28

A* Algorithm Properties:-

1) Admissibility

Admissibility means that h(n) is less than or equal to the cost of the minimal path from n to the goal

2) Consistency

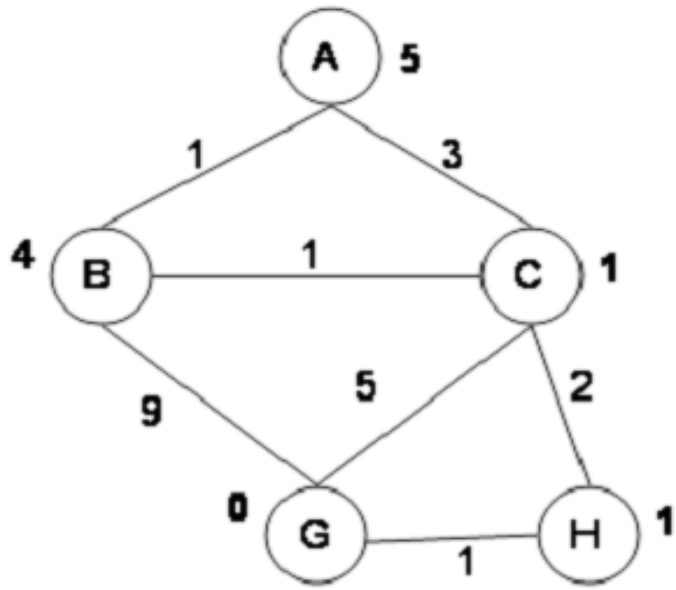
Consistency means that the difference between the heuristic of a state and the heuristic of its descendent is less than or equal the cost between them, and the heuristic of the goal equal zero. In other words,

1) **h(ni)-h(nj) ≤ cost(ni,nj).**
2) **h(goal)=0.**

3) Informedness

For two A* heuristics h1 and h2 , if h1(n) ≤ h2(n), for all states n in the search space , heuristics h2 is said to be more informed than h1.

Example: Consider the graph shown below where the numbers on the links are link costs and the numbers next to the states are heuristic estimates. Note that the arcs are undirected. Let **A** be the start state and **G** be the goal state.



1. Is the heuristic given in Problem above admissible? Explain.

Yes. The heuristic is admissible because it is less than or equal to the actual shortest distance to the goal.

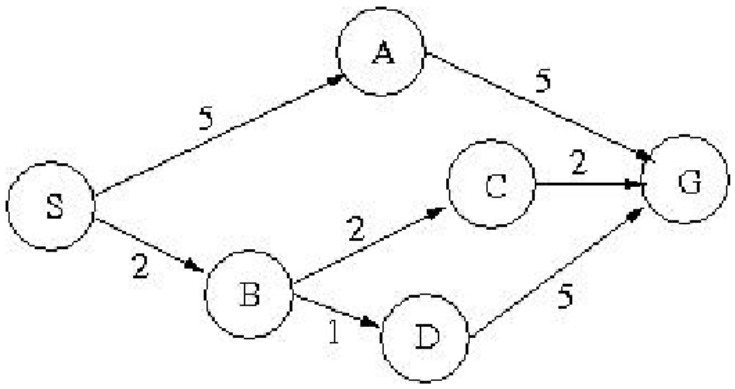
2. Is the heuristic given in Problem above consistent? Explain.

No, the heuristic is not consistent. There are two places in the graph where consistency fails. One is between A and C where the drop in heuristic is 4, but the path length is only 3. The other is between B and C where the drop in heuristic is 3 but the path length is only 1.

3. Did the A* algorithm find the optimal path in the previous example? If it did find the optimal path, explain why you would expect that. If it didn't find the optimal path, explain why you would expect that and give a simple (specific) change of state values of the heuristic that would be sufficient to get the correct behavior.

A* will not find the shortest path (which is ABCHG with cost 5). This is because the heuristic is not consistent. We can make the heuristic consistent by changing its value at C to be 3. There are other valid ways to make the graph consistent (change h(B) to 2 and h(A) to 3, for example) and those were right as well.

Example: Consider the following search space where we want to find a path from the start state S to the goal state G. The table shows three different heuristic functions h1, h2, and h3.



Node	*h1*	*h2*	*h3*
S	0	5	6
A	0	3	5
B	0	4	2
C	0	2	5
D	0	5	3
G	0	0	0

What solution path is found by Best-first search using h2, and h3?

Answer: With h2 the path is S,A,G and with h3 the path is…..(HW1).

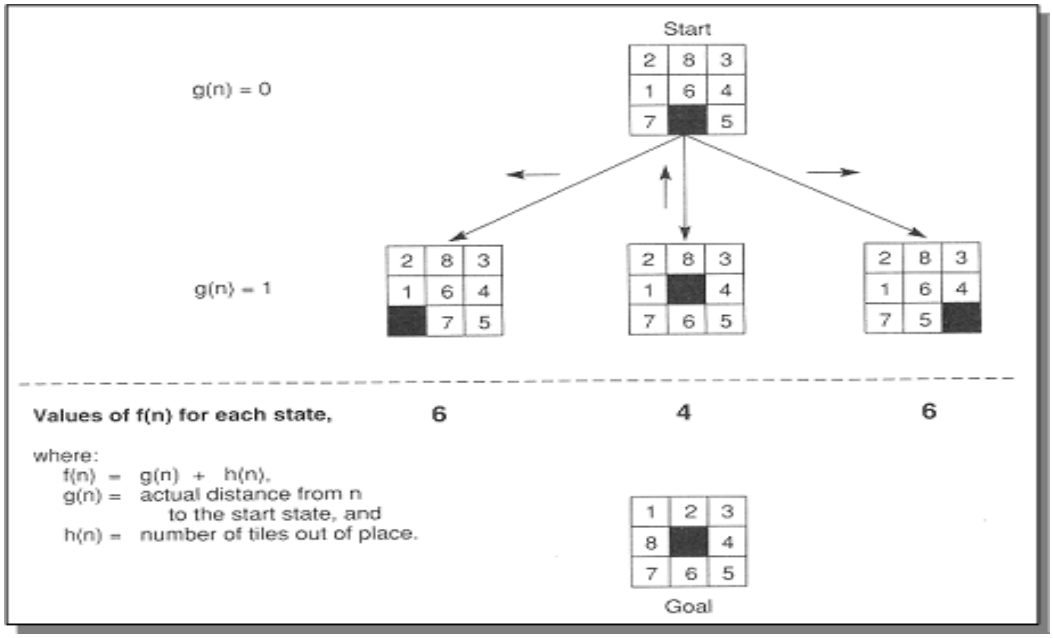
HW2: Use A Search with heuristic h1 to get the goal from S to G. Explain the Open and Closed lists and write the final path.

HW3: Use A* Search with heuristic h2 to get the goal from S to G. Explain the Open and Closed lists and write the final path.

<u>HW4:</u>Using Breadth-first search to get the goal from S to G. Explain the Open and Closed lists and write the final path. (Note: The children nodes are explored in alphabetical order whenever possible).

Complex Search Space and problem solving Approach

1- 8-puzzle problem:



To summarize:

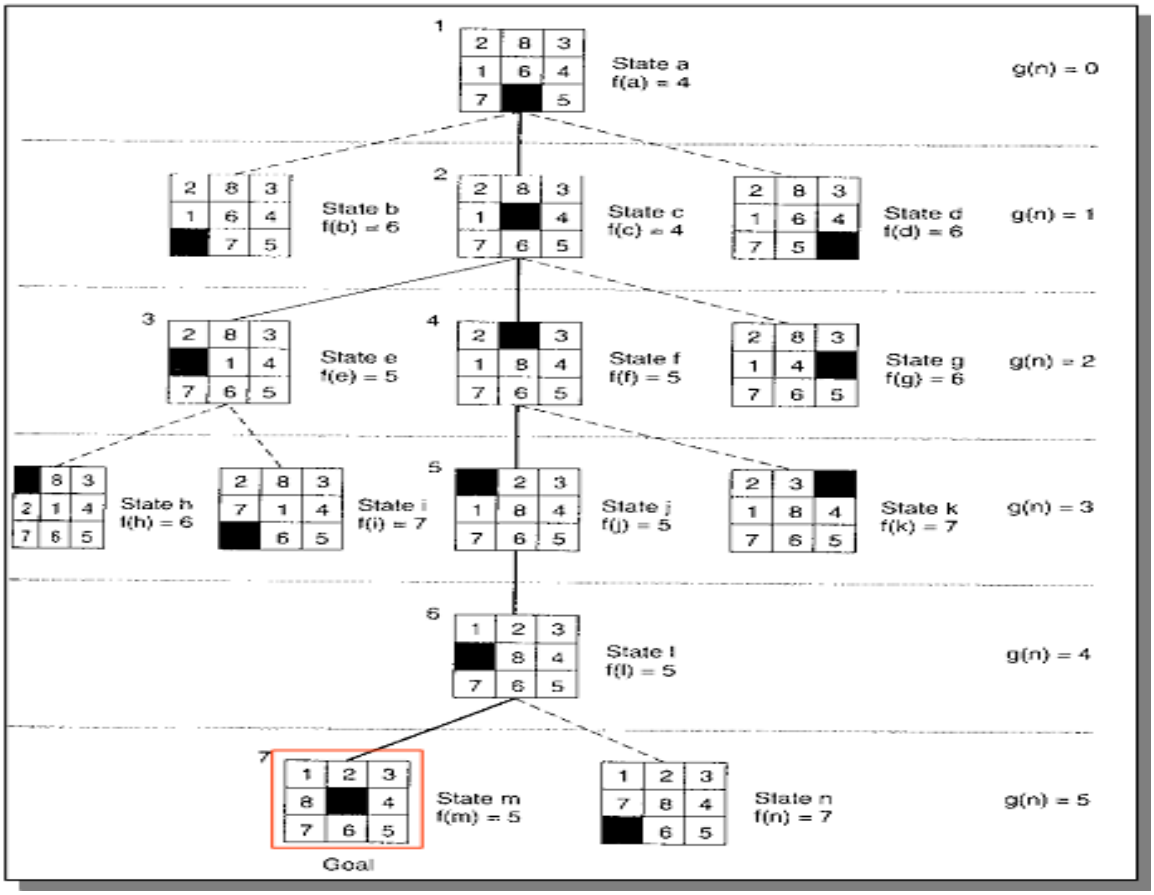
1. Operations on states generate children of the state currently under examination.

2. Each new state is checked to see whether it has occurred before (is on either open or closed), thereby preventing loops.

3. Each state n is given an I 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). The h value guides search toward heuristically promising states while the **g** value prevents search from persisting indefinitely on a fruitless path.

4. States on open are sorted by their f values. By keeping all states on open until they are examined or a goal is found, the algorithm recovers from dead ends.

5. As an implementation point, the algorithm's efficiency can be improved through maintenance of the open and closed lists, perhaps as heaps or leftist trees.



After implementation of **A** algorithm, the Open and Closed is shown as follows:

1. Open=[a4], Closed=[]

2. Open=[c4,b6,d6],Closed=[a4]

3. Open=[e5,f5,b6,d6,g6],Closed=[a4,c4]

4. Open=[f5,b6,d6,g6,h6,i7],Closed=[a4,c4,e5]

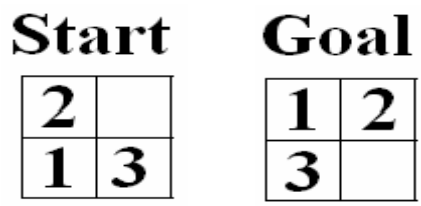
5. Open=[j5,b6,d6,g6,h6,j7,k7], Closed=[a4,c4,e5,f5]

6. Open=[l5, b6,d6,g6,h6,j7,k7],Closed=[a4,c4,e5,f5,j5]

7. Open=[m5, b6,d6,g6,h6,j7,k7,n7],Closed=[a4,c4,e5,f5,j5,l5]

8. Success, m=goal!!

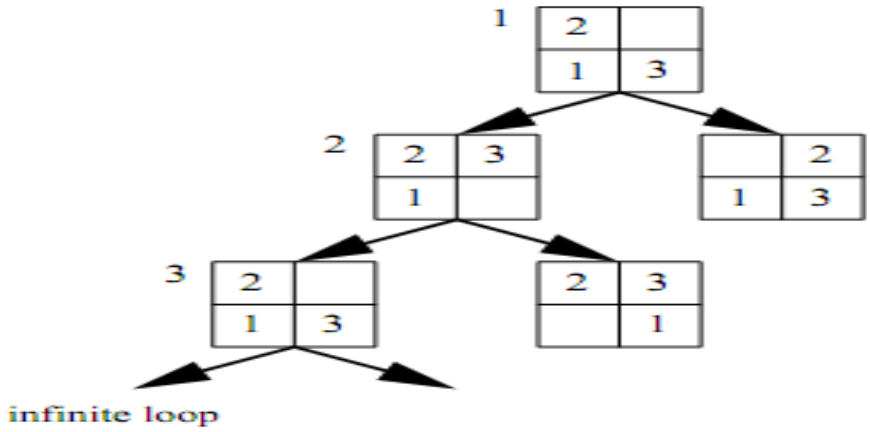
Example: Consider the 3-puzzle problem, which is a simpler version of the 8-puzzle where the board is 2 × 2 and there are three tiles, numbered 1, 2, and 3. There are four moves: move the blank up, right, down, and left. The cost of each move is 1. Consider this start and goal states:



Are the goal is found using Depth First Search algorithm <u>(If duplicate state is allow)</u>? If not explain why?

Answer:

The goal <u>cannot be found</u> using Depth First Search algorithm because there is an infinite loop will occur before we get the goal as shown below:



<u>HW:</u> How the solution will be when we delete duplicate state (duplicate state is not allow)?

HW: Draw the complete path using the A_algorithm Search to find the above goal form the above start (Note: the heuristic h(n) equal to the number of tiles out of place).Explain the value of each node.

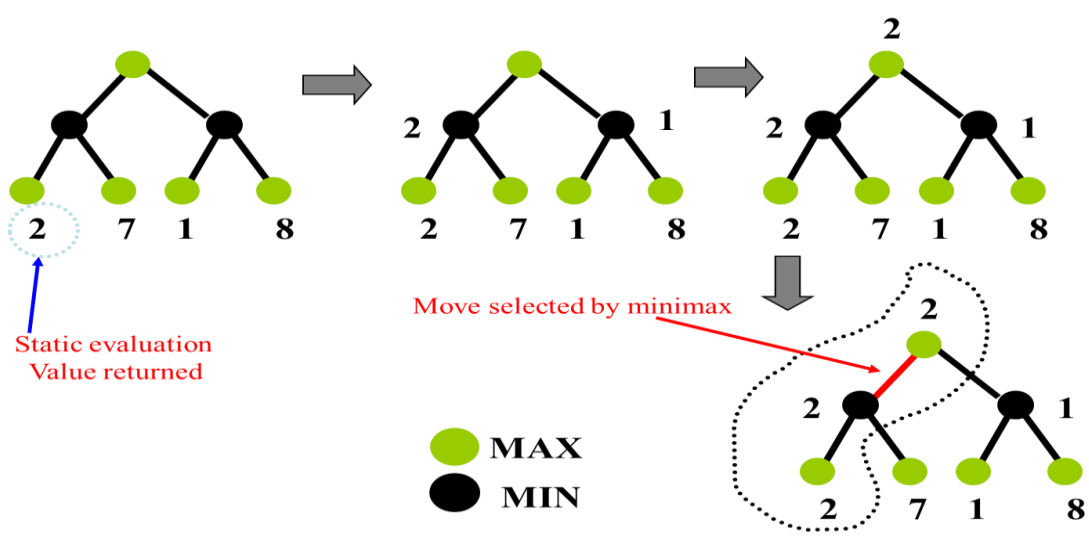
Adversarial Search in Game playing

In simple games, algorithms exist that can search the game trees to determine the best move to make from the current state. The most well-known is called the Minimax algorithm. The minimax algorithm is a useful method for simple two-player games. It is a method for selecting the best move given an alternating game where each player opposes the other working toward a mutually exclusive goal. Each player knows the moves that are possible given a current game state, so for each move, all subsequent moves can be discovered.

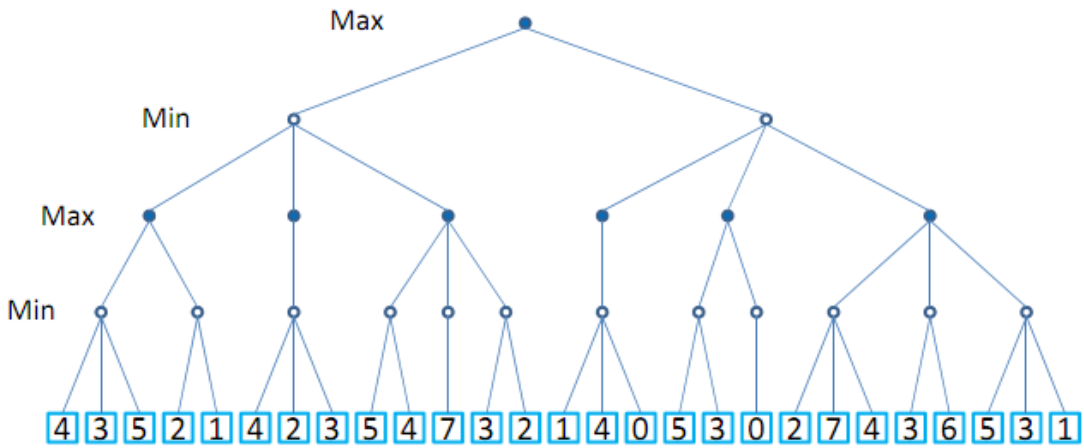
At each node in the tree (possible move) a value defining the goodness of the move toward the player winning the game can be provided. So at a given node, the child nodes (possible moves from this state in the game) each have an attribute defining the relative goodness of the move. It's an easy task then to choose the best move given the current state. But given the alternating nature of two-player games, the next player makes a move that benefits them. Minimax can use one of two basic strategies. In the first, the entire game tree is searched to the leaf nodes (end-games), and in the second, the tree is searched only to a predefined depth and then evaluated. The minimax algorithm assumes two players are represented as max and min (such as Max = Computer & Min = Opponent) and perform three tasks:

1- Construct tree (depth-bound)
2- Compute evaluation leaves
3- Propagate upwards (min/max)

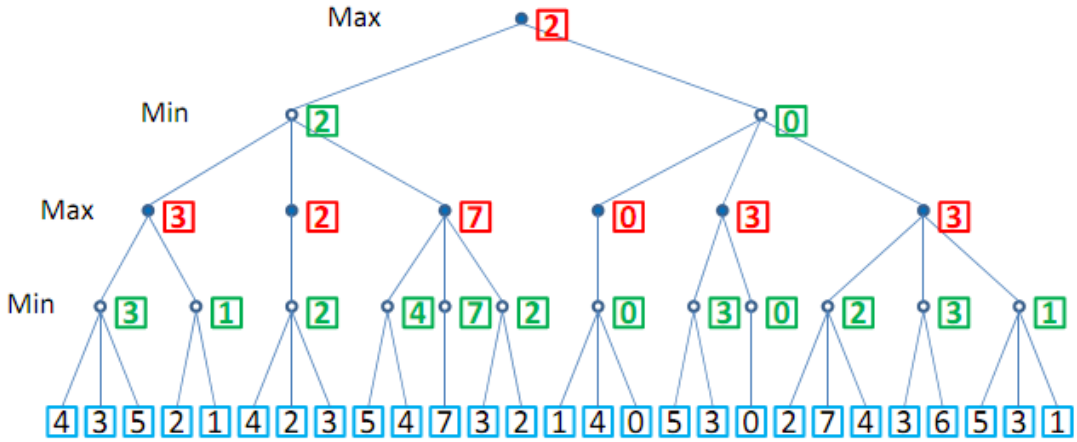
This is explained as follows :



Example: Perform the MiniMax algorithm on the figure below.



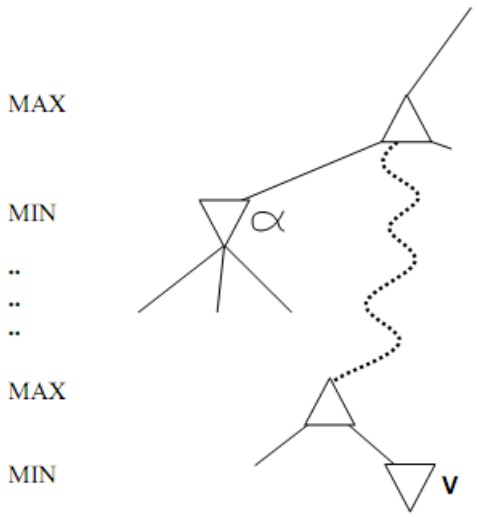
Solution:



Minimax with Alpha-Beta Pruning

Alpha-beta pruning is a simple algorithm that minimizes the game-tree search for moves that are obviously bad. The basic idea of alpha-beta pruning is the Identifying moves that are not beneficial, and remove them from the game tree. The higher in the game tree that branches are pruned the greater effect in minimizing the search space of the tree. Let's now explore the algorithm behind alpha-beta pruning. During the depth-first search of the game tree, we calculate and maintain two variables called alpha and beta. The alpha variable defines the best move that can be made to maximize (our best move) and the beta variable defines the best move that can be made to minimize (the opposing best move). While we traverse the game tree, if alpha is ever greater than or

equal to beta, then the opponent's move forces us into a worse position (than our current best move). In this case, we avoid evaluating this branch any further. The Idea of α-β then is summarized as follows:



α is the best value (to MAX) found so far off the current path

If value x of some node below V is known to be less than α,

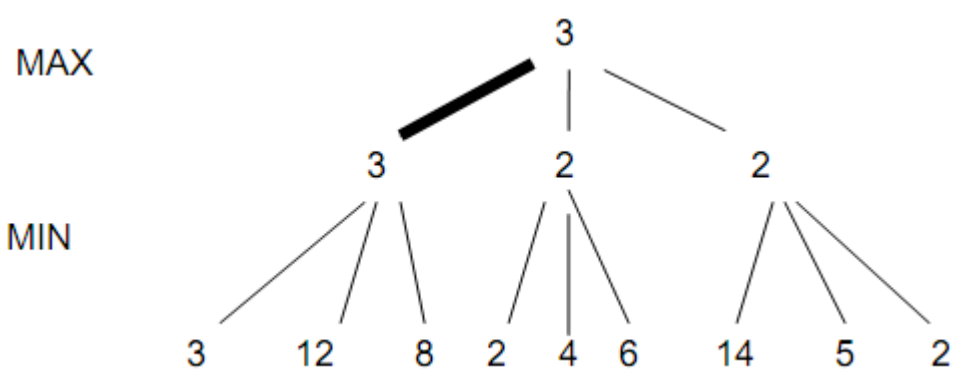
then value of V is known to be at most x, i.e., less than α,

therefore MAX will avoid node V

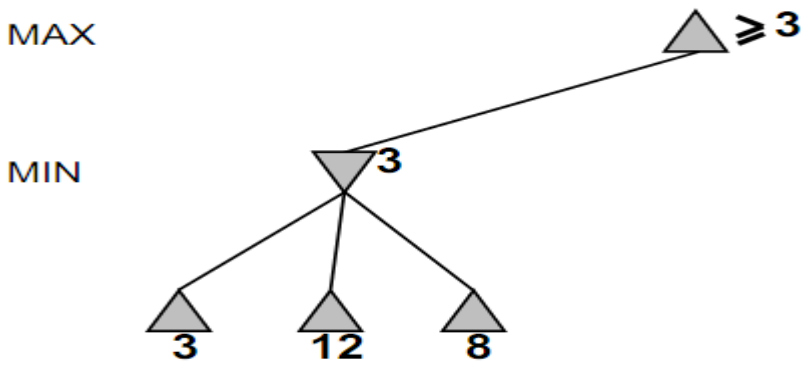
Consequence

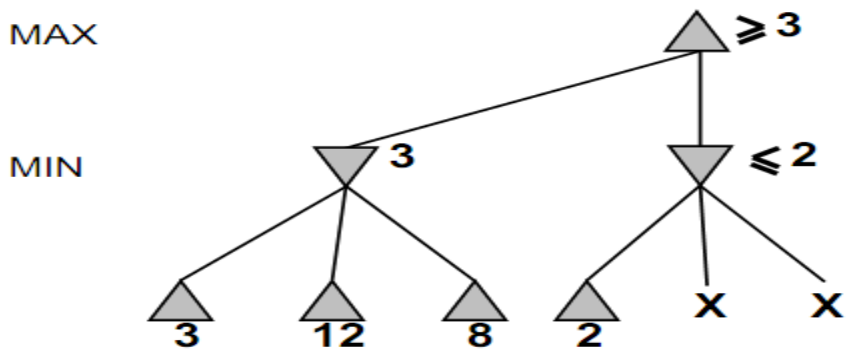
No need to expand further nodes below V

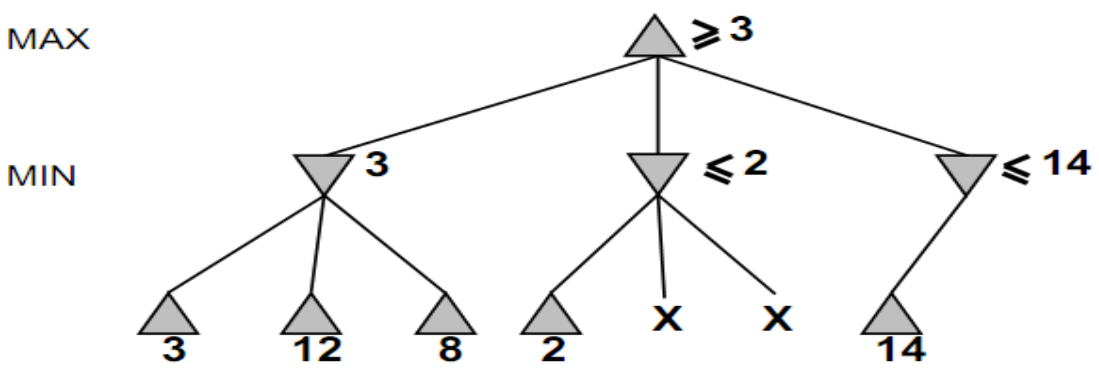
For example: Suppose a minmax problem is shown as in figure below:

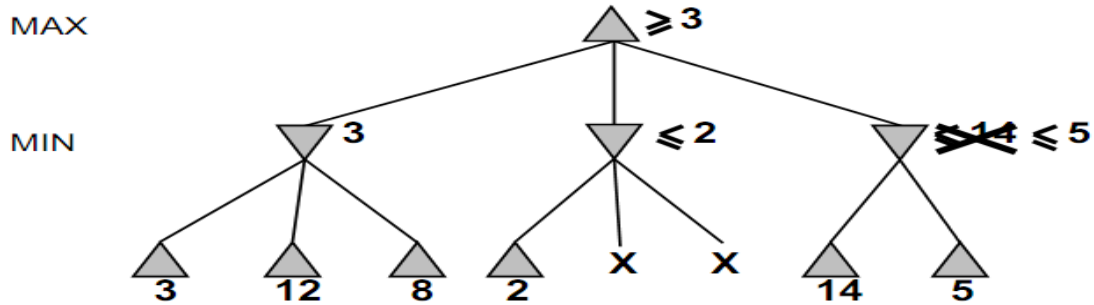


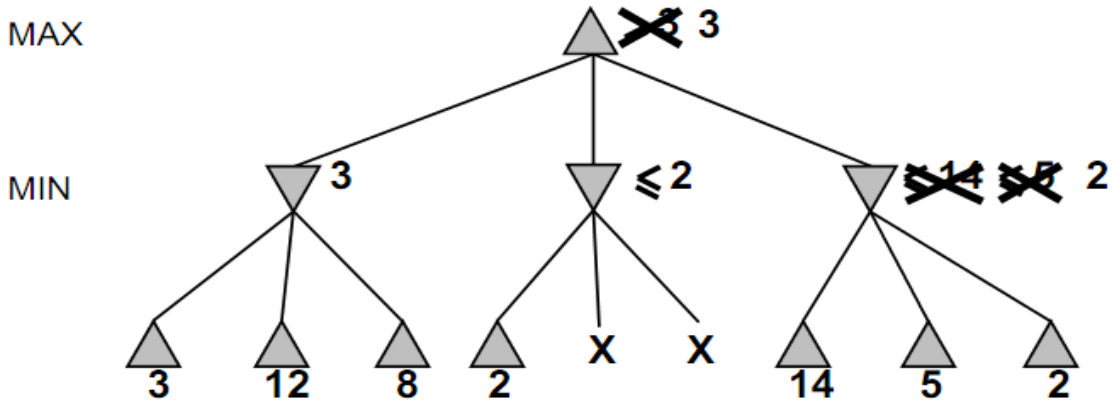
We can perform the MiniMax algorithm with $\boldsymbol{\alpha\beta}$**-pruning** on the figure as follows:


MAX
MIN
≥ 3
3
3
12
8

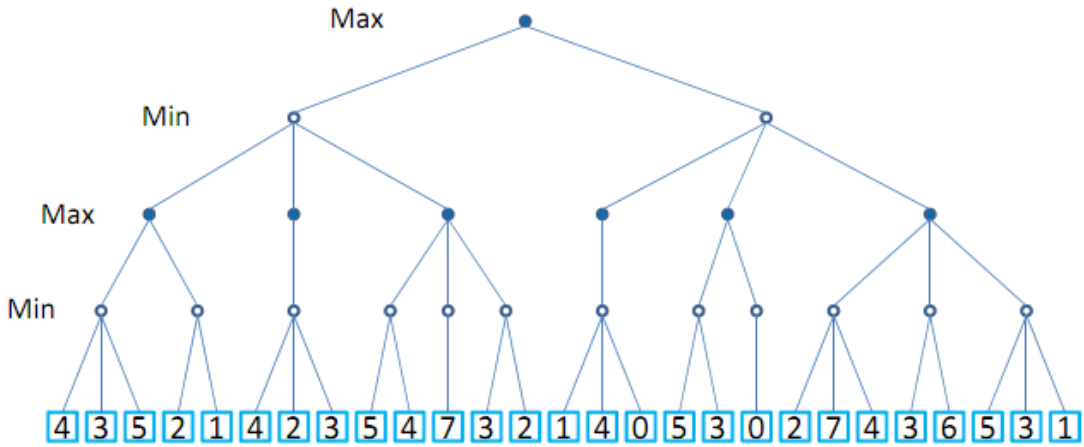

MAX
MIN
≥ 3
3
≤ 2
3
12
8
2
X
X


MAX
MIN
≥ 3
3
≤ 2
≤ 14
3
12
8
2
X
X
14



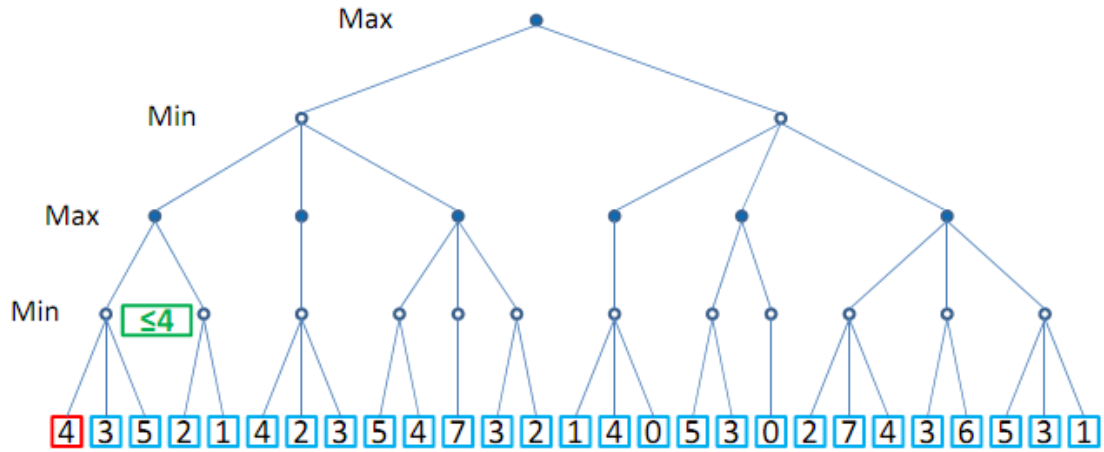


Example: Perform the MiniMax algorithm with $\alpha\beta$**-pruning** to the figure below and then enumerate and list (from left to right) the evaluation leaves that should be avoided.

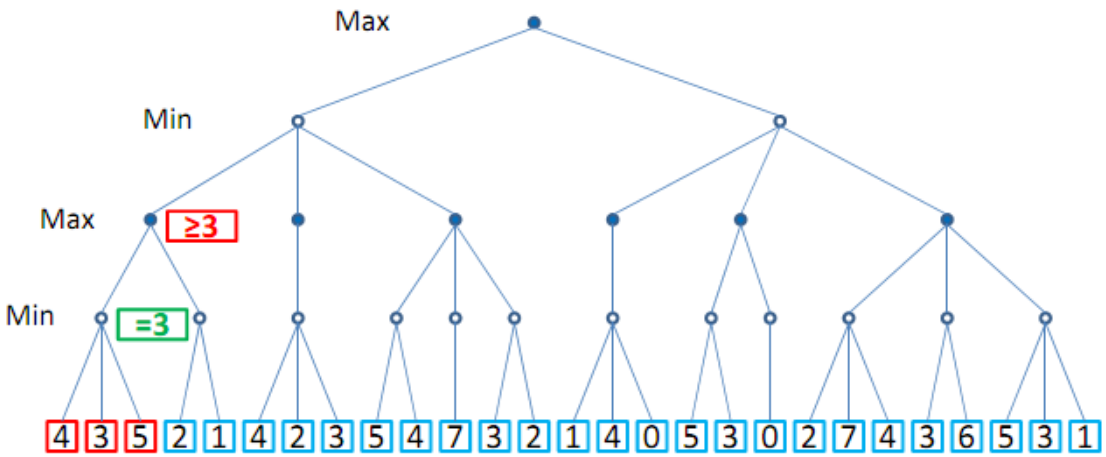


Solution:

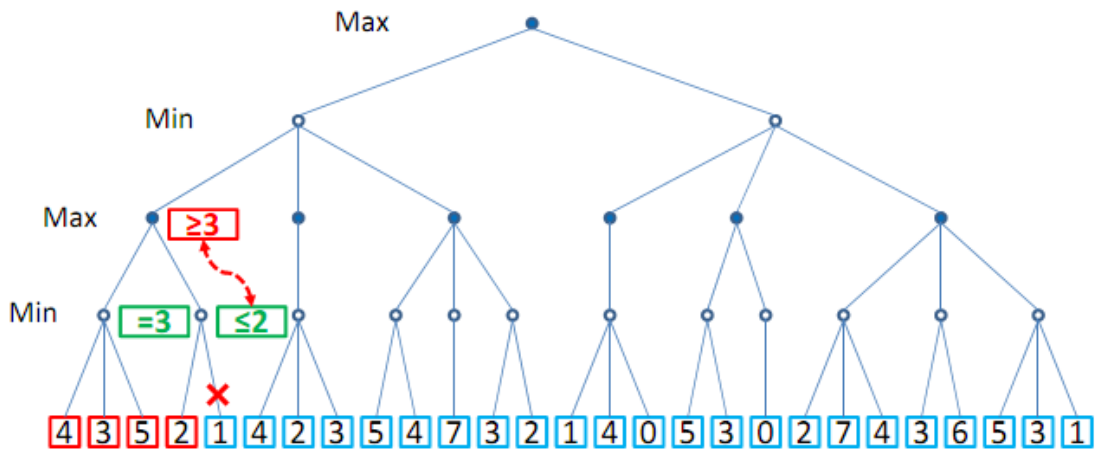
***α*-nodes**: Temporary values at MIN-nodes



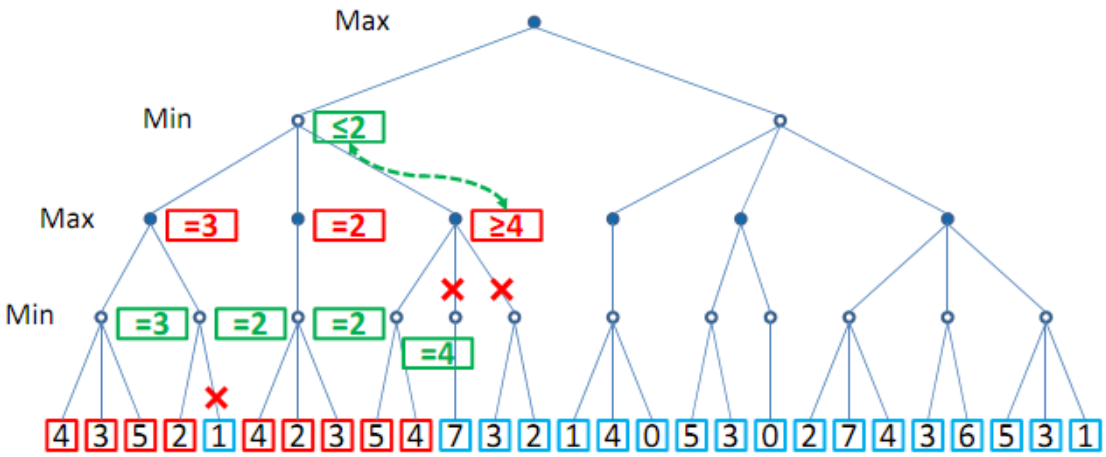
***β*-nodes**: Temporary values at MAX-nodes.



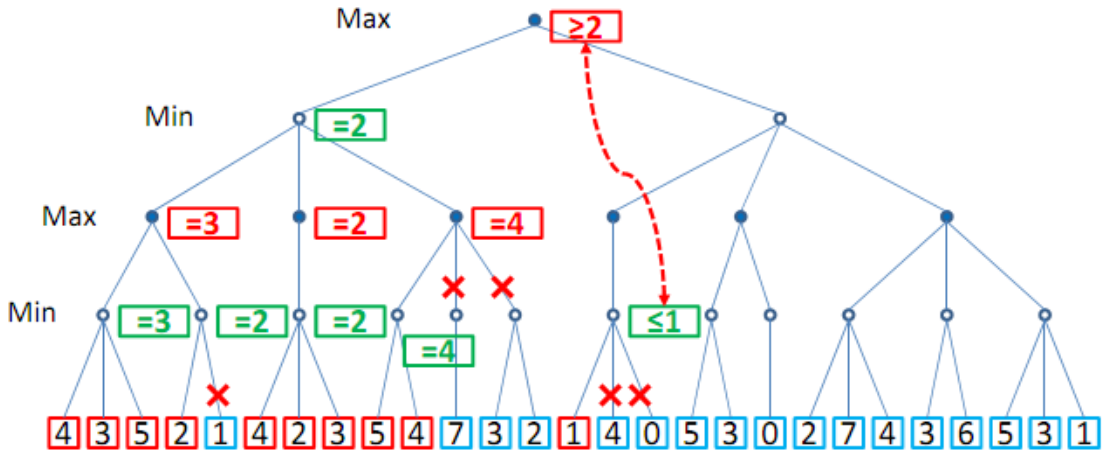
Prune: Parent ***β*-node** ≥ Child ***α*-node**



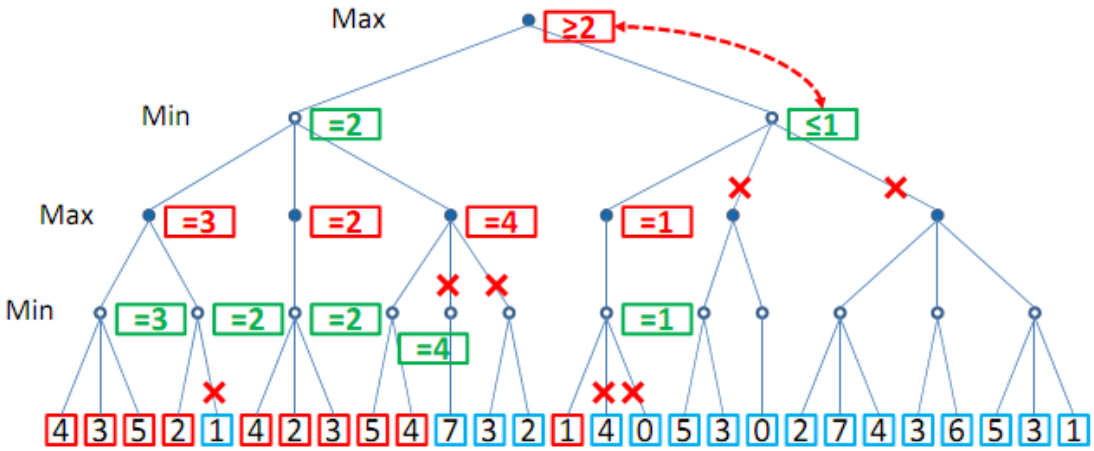
Prune: Parent α **-node** ≤ Child β**-node**



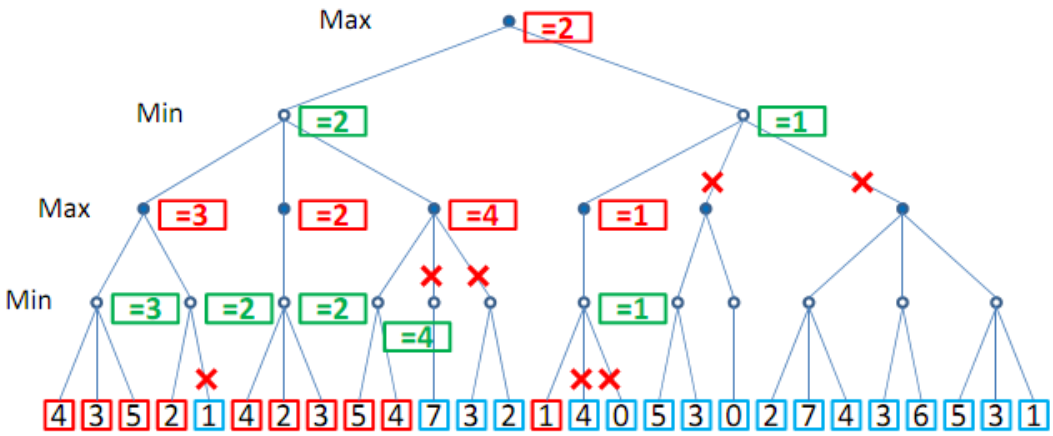
"Deep" cut-off: Ancestor β**-node** ≥ α**-node**



Prune: Parent $\boldsymbol{\beta}$**-node** ≥ Child $\boldsymbol{\alpha}$**-node**



The final evaluations are shown below:



There are **17** evaluation leaves have been avoided and they listed as **[1,7,3,2,4,0,5,3,0,2,7,4,3,6,5,3,1].**

Knowledge Representation

There are many methods can be used for knowledge representation and they can be described as follows:-

1- Semantic net.

2- Conceptual graph.

3- Frames

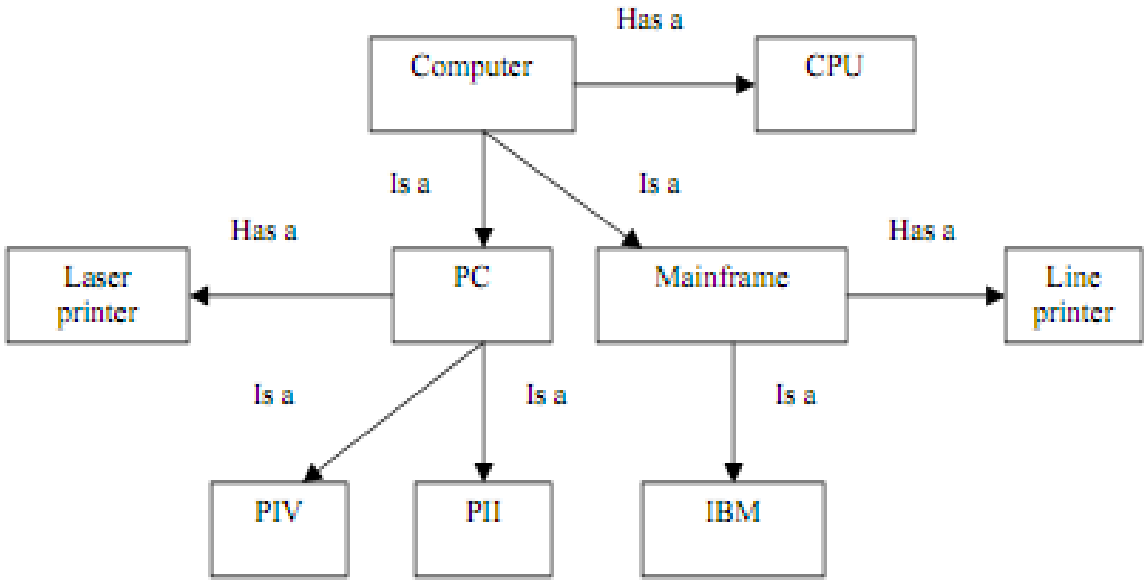
4- Prepositional and Predicates calculus.

5- Resolution.

1) Semantic Net

It is consist of a set of nodes and arcs , each node is represented as a rectangle to describe the objects, the concepts and the events. The arcs are used to connect the nodes and they divided to three parts which are (Is),(Is a), (Can), and (Has a).

Example1: Computer has much part like a CPU and the computer divided into two types, the first one is the mainframe and the second is the personal computer, Mainframe has line printer with large sheet but the personal computer has laser printer, IBM as example to the mainframe and PIII and PIV as example to the personal computer.

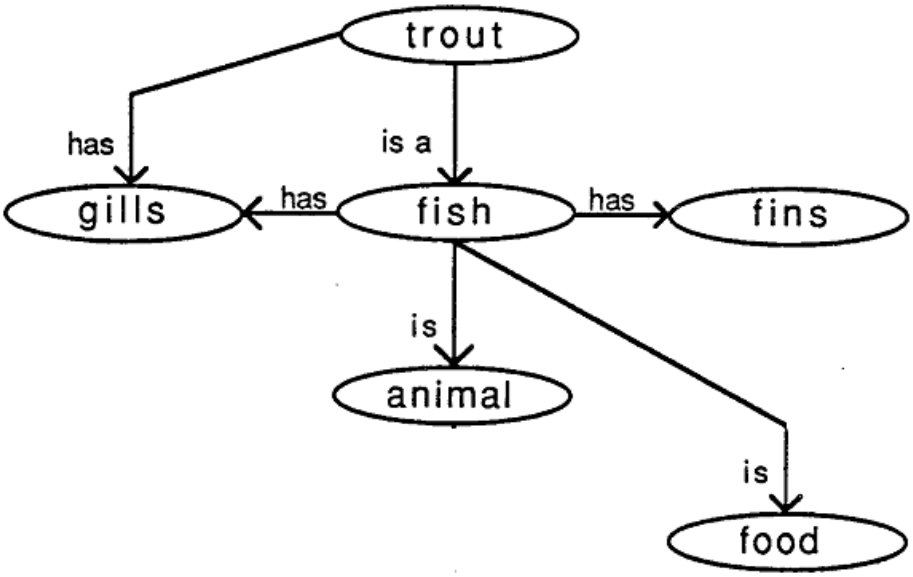


Example2: Create the semantic network for the following facts (Note: You must append new indirect facts if they exist):

1- A trout is a fish.
2- A fish has gills.
3- A fish has fins.
4- Fish is food.
5- Fish is animal.

Solution:

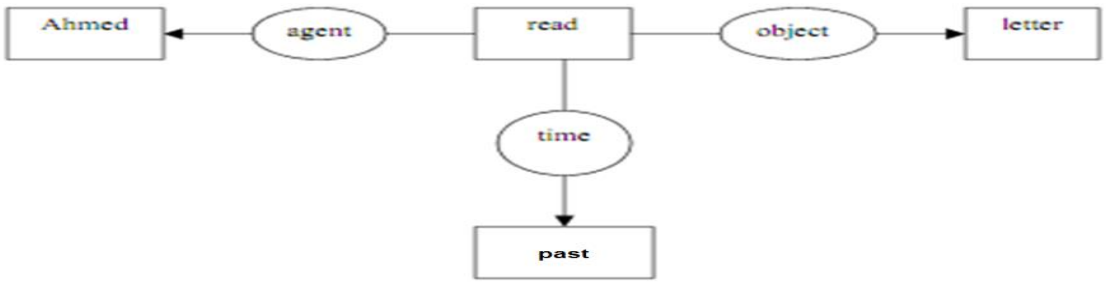
There is a fact must be added that is “A trout has gills” because all the fishes have gills. The semantic network is shown below:



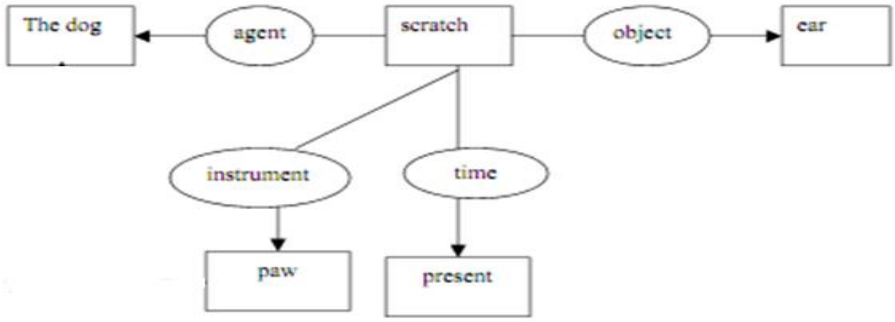
2) Conceptual Graphs:

Conceptual Graphs is a logical formalism that includes classes, relations, individuals and quantifiers. This formalism is based on semantic networks, but it has direct translation to the language of first order predicate logic, from which it takes its semantics. The main feature is standardized graphical representation that like in the case of semantic networks allows human to get quick overview of what the graph means. Conceptual graph is a bipartite orientated graph where instances of concepts are displayed as *rectangle* and conceptual relations are displayed as *ellipse*. Oriented edges then link these vertices and denote the existence and orientation of relation. A relation can have more than one edges, in which case edges are numbered.

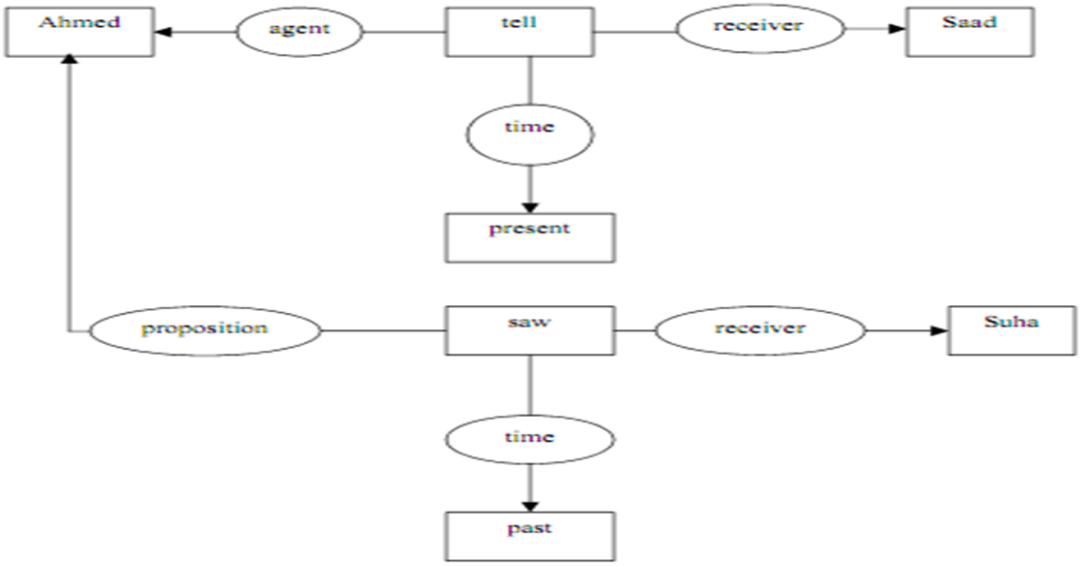
Example 1: Ahmed read a letter yesterday.



Example 2: The dog scratch its ear with its paw.



Example 3: Ahmed tell Saad that he saw Suha.

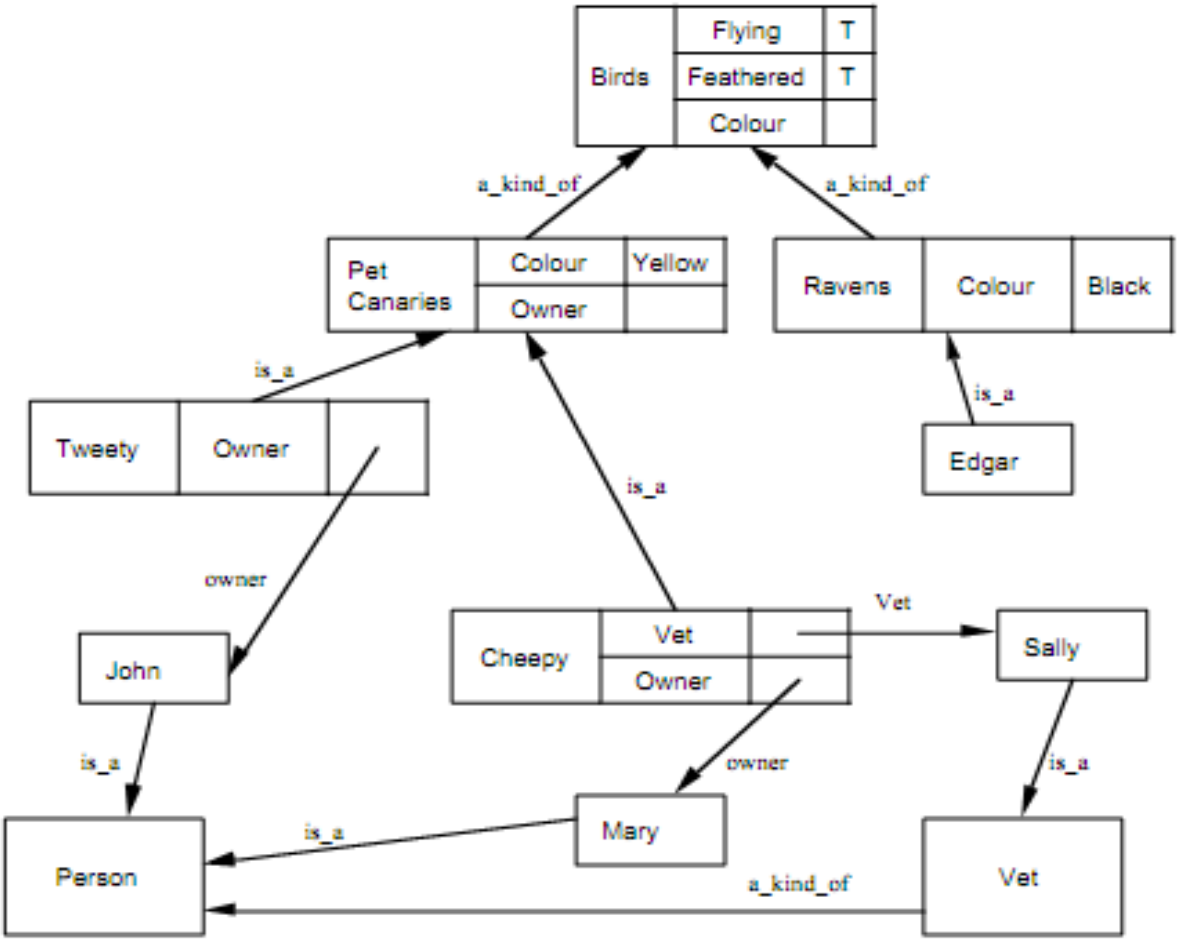


3) Frame:

Consideration of the use of cases suggests how we can tighten up on the semantic net notation to give something which is more consistent, known as the frame notation. In the place of an arbitrary number of arcs leading from a node there are a fixed number of slots representing attributes of an object.

Every object is a member or instance of a class, which it may be thought of as linking to with an is_a link as we saw before. The class indicates the number of slots that an object has, and the name of each slot. In the case of a giving object, for instance, the class of giving objects will indicate that it has at least three slots: the donor, the recipient and the gift. There may be further slots indicated as necessary in the class, such as ones to give the time and location of the action. The time slot may be considered a formalization of the tense of the verb in a sentence.

In our example we have a general class of birds, and all birds have attributes flying, feathered and color. The attributes flying and feathered are Boolean values and are fixed to true at this level, which means that for all birds the attribute flying is true and the attribute feathered is true. The attribute color, though defined at this level is not filled, which means that though all birds have a color, their color varies. Two subclasses of birds, pet_canaries and ravens are defined. Both have the color slot filled in, pet_canaries with yellow, ravens with black. The class pet_canaries has an additional slot, owner, meaning that all pet canaries have an owner, though it is not filled at this level since it is obviously not the case that all pet canaries have the same owner. We can therefore say that any instance of the class pet_canary has attributes color yellow, feathered true, flying true, and owner, the last of these varying among instances. Any instance of class raven has color black, feathered true, flying true, but no attribute owner. The two instances of pet_canary shown, Tweety and Cheepy have owners John and Mary who are separate instances of the class person, for simplicity no attributes have been given for class person. The instance of pet_canary Cheepy has an attribute which is restricted to itself, vet (since not all pet canaries have their own vet), which is a link to another person instance, but in this case we have subclass of person, vet. The frame diagram for this is:



We can define a general set of rules for making inferences on this sort of frame system. We can say that an object is an instance of a class if it is a member of that class, or if it is a member of a class which is a subclass of that class. A class is a subclass of another class if it is a kind of that class, or if it is a kind of some other class which is a subclass of that class. An object has a particular attribute if it has that attribute itself, or if it is an instance of a class that has that attribute.

4) The Prepositional and Predicates Calculus:

4.1 The Prepositional Calculus:

The propositional calculus and 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 produce the pieces that make it up: its set of symbols.

<u>Propositional Calculus Symbols</u>

- The symbols of propositional calculus are: {P, Q, R, S, …}
- Truth symbols: **{True, false}**
- Connectives: **{∧ ,∨ , ¬ , → , ≡}**

Propositional symbols denote ***propositions,*** or statements about the world that may be either true or false, Propositions are denoted by uppercase letters near the end of the English alphabet Sentences, **For example:**

P: It is sunny today.

Q: The sun shines on the window.

R: The blinds are down.

(P→Q): If it is sunny today, then the sun shines on the window

(Q→R): If the sun shines on the window, the blinds are brought down.

(¬R): The blinds are not yet down.

Propositional Calculus Sentence

- Every propositional 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,*** **AND***,* of two sentences is a sentence.

  For example: **P ∧ ¬P** is a sentence.

- The ***disjunction,*** **OR** of two sentences is a sentence.

  For example: **P ∨ ¬P** is a sentence.

- The ***implication*** of one sentence from another is a sentence.

  For example: **P → Q** is a sentence.

- The ***equivalence*** of two sentences is a sentence.

  For example: **P ∨ Q ≡ R** is a sentence.

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

In expressions of the form **P ∧ Q, P** and **Q** are called the ***conjuncts***. In **P ∨ Q**, **P** and **Q** are referred to as ***disjuncts***. In an implication, **P → Q, P** is the ***premise*** and ***Q***, the ***conclusion*** or ***consequent.***

In propositional calculus sentences, the symbols **( )** and **[ ]** are used to group symbols into sub-expressions and so to control their order of evaluation and meaning.

For Example: (P ∨ Q) ≡ R is quite different from **P ∨ (Q ≡ R)** as can be demonstrated using truth tables. An expression is a sentence, or well-

formed formula, of the propositional calculus if and only if it can be formed of legal symbols through some sequence of these rules.

For Example: $((P \wedge Q) \rightarrow R) \equiv \neg P \vee \neg Q \vee R$ is a well-formed sentence in the propositional calculus because:

P, Q, and **R** are propositions and thus sentences.

$P \wedge Q$, the conjunction of two sentences, is a sentence.

$(P \wedge Q) \rightarrow R$, the implication of a sentence for another, is a sentence.

$\neg P$ and $\neg Q$, the negations of sentences, are sentences.

$\neg P \vee \neg Q$ the disjunction of two sentences, is a sentence.

$\neg P \vee \neg Q \vee R$, the disjunction of two sentences, is a sentence.

$((P \wedge Q) \rightarrow R) \equiv \neg P \vee \neg Q \vee R$, the equivalence of two sentences, is a sentence.

This is our original sentence, which has been constructed through a series of applications legal rules and is therefore "***well formed***".

PROPOSITIONAL CALCULUS SEMANTICS

An ***interpretation*** of a set of propositions 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 sentences is determined by:

- The truth assignment of ***negation,*** $\neg P$, 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,*** $\wedge$, is **T** only when both **conjuncts** have truth value **T**; otherwise it is **F**.
- The truth assignment of ***disjunction,*** $\vee$, is **F** only when both **disjuncts** have truth value **F**; otherwise it is **T**.
- The truth assignment *of* ***implication,*** $\rightarrow$, 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,*** $\equiv$, is **T** only when both expressions have the same truth assignment for all possible interpretations; otherwise it is **F**.

The truth assignments of compound propositions are often described by ***truth tables.*** A truth table lists all possible truth value assignments to the atomic propositions of an expression and gives the truth value of the expression for each assignment. Thus, a truth table enumerates all possible worlds of interpretation that may be given to an expression. For Example, the truth table for **P** $\wedge$ **Q**, **Fig.(a)** , lists truth values for each possible truth assignment of the operands.

P	Q	P $\wedge$ Q
T	T	T
T	F	F
F	T	F
F	F	F

Fig. (a) : Truth table for AND $\wedge$ operator

P $\wedge$ **Q** is true only when both **P** and **Q** are both **T**. **Or ($\vee$), not($\neg$), implies ($\rightarrow$)**, and **equivalence ($\equiv$)** are defined in a similar fashion. The construction

of these truth tables is left as an exercise. Two expressions in the propositional calculus are equivalent if they have the same value under all truth value assignments.

This equivalence may be demonstrated using truth tables. For example, a proof of the equivalence of **P → Q** and **¬P ∨ Q** is given by the truth table **Fig.(b).**

P	Q	¬P	¬P∨Q	P→Q	(¬P∨Q) ≡(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

Fig. (b) : Truth table demonstrating the equivalence of (¬P∨Q) ≡ (P→Q)

By demonstrating that two different sentences in the propositional calculus have identical truth tables, we can prove the following equivalences. For propositional expressions **P, Q,** and **R**:

The double negation law : $\neg(\neg P) \equiv P$

The implication in terms of V law: $P \rightarrow Q \equiv \neg P \vee Q$

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

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

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

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

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

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

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

Identities such as these can be used to change propositional calculus expressions into a syntactically different but logically equivalent form. These identities may be used instead of truth tables to prove that two expressions are equivalent: find a series of identities that transform one expression into the other.

The ability to change a logical expression into a different form with equivalent truth values is also important when using ***inference rules*** (**modus ponens, and resolution**) that require expressions to be in a specific form.

Truth table then will list all possible truth value assignments to the propositions of an expression, the standard truth tables are shown the figure below:

And

p	q	$p \cdot q$
T	T	T
T	F	F
F	T	F
F	F	F

Or

p	q	$p \vee q$
T	T	T
T	F	T
F	T	T
F	F	F

If . . . then

p	q	$p \rightarrow q$
T	T	T
T	F	F
F	T	T
F	F	T

Not

p	$\sim p$
T	F
F	T

Example: Use a truth table to list all possible truth value assignments to the propositions of the expression **(P ∧ Q) ∨ (¬ Q ∨ P) .**

Answer:

P	Q	P∧Q	¬Q	¬Q∨P	(P∧Q)∨(¬Q∨P)
T	T	T	F	T	T
T	F	F	T	T	T
F	T	F	F	F	F
F	F	F	T	T	T

Example: Prove that **(P ∧ Q)** is not equivalent to $(P \rightarrow Q)$, in other word prove $(P \wedge Q) \not\equiv (P \rightarrow Q)$

P	Q	$(P \wedge Q)$	$(P \rightarrow Q)$
T	T	T	T
T	F	F	F
F	T	F	T
F	F	F	T

Example: True or false: $(P \wedge Q) \not\equiv (P \rightarrow Q)$. **Answer:** true.

Example: True or false: $((P \rightarrow Q) \wedge Q) \rightarrow P$. **Answer:** ???.

Example: Convert the following engilsh sentences to propositional calculus sentences:

- **It is hot.**
- **It is not hot.**
- **If it is raining, then will not go to mountain.**
- **The food is good and the service is good.**
- **If the food is good and the service is good then the restaurant is good**.

Answer:

- It is hot
  p
- It is not hot
  ¬p
- If it is raining, then will not go to mountain
  p → ¬q
- The food is good and the service is good
  x ∧ y
- If The food is good and the service is good then the restaurant is good
  x ∧ y → z

4.2 The Predicate Calculus (Also known as First-Order Logic):

In propositional 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. In other words, the propositional calculus has its limitations that you cannot deal properly with general statements because it represents each statement by using some symbols jointed with connectivity tools. To solve the limitations in the propositional calculus, you need to analyze propositions into predicates and arguments, and deal explicitly with quantification. Predicate calculus provides formalism for performing this analysis of propositions and additional methods for reasoning with quantified expressions. For example, instead of letting a single propositional symbol, **P**, denotes the entire sentence "**it rained on Tuesday**," we can create a predicate weather that describes a relationship between a date and the weather:

weather (rain, tuesday)

through inference rules we can manipulate predicate calculus expression accessing their individual components and inferring new sentences. Predicate calculus also allows expressions to 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 (rain, X) is true ;

I,e., it rains it rains every day. As with propositional calculus, we will first define the syntax of the language and then discuss its semantics.

Example: Convert the following engilsh sentences to predicate calculus sentences:

1. If it is raining, tom will not go to mountain
2. If it doesn't rain tomorrow, Tom will go to the mountains.
3. All basketball players are tall.

4. Some people like anchovies.
5. John like anyone who likes books.
6. Nobody likes taxes.
7. There is a person who writes computer class.
8. All dogs are animals.
9. All cats and dogs are animals.
10. John did not study but he is lucky.
11. There are no two adjacent countries have the same color.
12. All blocks supported by blocks that have been moved have also been moved. Note: you can use the following predicates:

- block(X) means X is a block
- supports(X, Y) means X supports Y
- moved(X) means X has been moved

Answer:

1. weather (rain) → ¬ go(tom,mountain)
2. ¬ weather (rain, tomorrow) →go(tom, mountains).
3. ∀ X (basketball _ player(X) → tall (X))
4. ∃ X (person(X) ∧ likes(X, anchovies)).
5. ∃ X like(X,book) →like(john,X)
6. ¬ ∃ X likes(X,taxes).
7. ∃X write(X,computer_class)
8. ∀X dogs(X) →animals(X)
9. ∀X∀Y cats(X)∧dogs(Y) →animals(X)∧animals(Y).
10. ¬ study(john)∧ lucky(john)
11. ∀X∀Y (county(X) ∧ county(Y) ∧ adjacent(X,Y)) → ¬ (color(X) ≡ color(Y)). Or we say: ∀X∀Y ¬ county(X) ∨ ¬county(Y) ∨ ¬adjacent(X,Y) ∨ ¬ (color(X) ≡ color(Y)).
12. ∀X∀Y block(X) ∧ block(Y) ∧ supports(X,Y) ∧ moved(X) → moved(Y)

5. Resolution:

Resolution is a technique for proving theorems in the predicate calculus using the resolution by refutation algorithm. The resolution refutation proof procedure answers a query or deduces a new result by reducing the set of clauses to a contradiction.

<u>The Resolution by Refutation Algorithm includes the following steps:-</u>

a) Convert the statements to **predicate calculus** (predicate logic).

b) Convert the statements from **predicate calculus** to **clause forms**.

c) Add the negation of what is to be proved to the clause forms.

d) Resolve the clauses to producing new clauses and producing a contradiction by generating the empty clause.

Clause Forms

The statements that produced from **predicate calculus** method are nested and very complex to understand, so this will lead to more complexity in resolution stage , therefore the following algorithm is used to convert the **predicate calculus** to **clause forms**:-

1. Eliminate all (→) by replacing each instance of the form **(P → Q)** by expression **(¬P∨Q)**
2. Reduce the scope of negation.

   $\neg(\neg a) \equiv a$

   $\neg(\forall X)\ b(X) \equiv \exists X \quad \neg\ b(X)$

   $\neg(\exists X)\ b(X) \equiv \forall X \quad \neg b(X)$

   $\neg(a \wedge b) \equiv \neg a \vee \neg b$

   $\neg(a \vee b) \equiv \neg a \wedge \neg b$
3. Standardize variables: rename all variables so that each quantifier has its own unique variable name. *For example*,

$\forall X\ a(X) \vee \forall X\ b(X) \equiv \forall X\ a(X) \vee \forall Y\ b(Y)$

4. Move all quantifiers to the left without changing their order. *For example*,

   $\forall X\ a(X) \vee \forall Y\ b(Y)$

   $\forall X\ \forall Y\ a(X) \vee b(Y)$

5. Eliminate existential quantification by using the equivalent function. *For example*,

   $\forall X \quad \exists Y\ (mother(X,Y)) \equiv \forall X\ (mother(X,m(X)))$

   $\forall X\ \forall Y\ \exists Z\ (p(X,Y,Z) \equiv \forall X\ \forall Y\ (p(X,Y,f(X,Y))$

6. Remove universal quantification symbols. *For example*,

   $\forall X\ \forall Y\ (p(X,Y, f(X,Y))) \equiv p(X,Y, f(X,Y))$

7. Use the associative and distributive properties to get a conjunction of disjunctions called **conjunctive normal form**. *For example*,

   $a \vee (b \vee c) \equiv (a \vee b) \vee c$

   $a \wedge (b \wedge c) \equiv (a \wedge b) \wedge c$

   $a \vee (b \wedge c) \equiv (a \vee b) \wedge (a \vee c)$

   $a \wedge (b \vee c) \equiv (a \wedge b) \vee (a \wedge c)$

8. Split each conjunct into a separate **clause**. *For example*,

   $(\neg a(X) \vee \neg\ b(X) \vee e(W)) \wedge (\neg b(X) \vee \neg d(X,f(X)) \vee e(W))$

   $\neg a(X) \vee \neg\ b(X) \vee e(W)$

   $\neg b(X) \vee \neg d(X,f(X)) \vee e(W)$

9. Standardize variables apart again so that each clause contains variable names that do not occur in any other clause.

For example,

$(\neg a(X) \vee \neg b(X) \vee e(W)) \wedge (\neg b(X) \vee \neg d(X,f(X)) \vee e(W))$

$\neg a(X) \vee \neg b(X) \vee e(W)$

$\neg b(Y) \vee \neg d(X,f(X)) \vee e(V)$

Example: Use the Resolution Algorithm for proving that John is happy with regard the following story:

Everyone passing his AI exam and winning the lottery is happy. But everyone who studies or lucky can pass all his exams, John did not study but he is lucky. Everyone who is lucky wins the lottery. Prove that John is happy.

Solution:

a) **Convert all statement to predicate calculus.**

$\forall X\ pass(X,ai\_exam) \wedge win(X,lottery) \rightarrow happy(X)$

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

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

$\forall Z\ lucky(Z) \rightarrow win(Z,lottery)$

happy(john)?

b) **Convert the statements from predicate calculus to clause forms**.

1.

$\forall X \quad \neg(pass(X,ai\_exam) \wedge win(X,lottery)) \vee happy(X)$

$\forall Y \forall E \neg(study(Y) \vee lucky(Y)) \vee pass(Y,E)$

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

$\forall Z \quad \neg(lucky(Z)) \vee win(Z,lottery)$

$happy(john)?$

2.

$\forall X \quad (\neg pass(X,ai\_exam) \vee \neg win(X,lottery)) \vee happy(X)$

$\forall Y \forall E (\neg study(Y) \wedge \neg lucky(Y)) \vee pass(Y,E)$

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

$\forall Z \quad \neg lucky(Z) \vee win(Z,lottery)$

$happy(john)?$

3. Nothing to do here.
4. Nothing to do here.
5. Nothing to do here.
6.

$(\neg pass(X,ai\_exam) \vee \neg win(X,lottery)) \vee happy(X)$

$(\neg study(Y) \wedge \neg lucky(Y)) \vee pass(Y,E)$

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

$\neg lucky(Z) \vee win(Z,lottery)$

happy(john)?

7.

$\neg$pass(X,ai_exam)$\vee\neg$win(X,lottery)$\vee$ happy(X)

($\neg$study(Y)$\wedge\neg$lucky(Y))$\vee$ pass(Y,E) $\equiv$ (a$\wedge$b)$\vee$c $\equiv$ c$\vee$(a$\wedge$b)

The second statement become: pass(Y,E)$\vee\neg$study(Y)$\wedge$pass(Y,E)$\vee\neg$lucky(Y)

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

$\neg$lucky(Z)$\vee$win(Z,lottery)

happy(john)?

8.

$\neg$pass(X,ai_exam)$\vee\neg$win(X,lottery)$\vee$ happy(X)

pass(Y,E)$\vee\neg$study(Y)

pass(Y,E)$\vee\neg$lucky(Y)

$\neg$study(john)

lucky(john)

$\neg$lucky(Z)$\vee$win(Z,lottery)

happy(john)?

9.

$\neg$pass(X,ai_exam)$\vee\neg$win(X,lottery)$\vee$ happy(X)

pass(Y,E)$\vee\neg$study(Y)

pass(M,G)∨¬lucky(M)

¬study(john)

lucky(john)

¬lucky(Z)∨win(Z,lottery)

happy(john)?

c) Add the negation of what is to be proved to the clause forms.

¬happy(john).

d) Resolve the clauses to producing new clauses and producing a contradiction by generating the empty clause.

There are two ways to do this, the first is *backward resolution* and the second is *forward resolution*.

d_1) Backward Resolution

The proving for ***happy(john)*** using **Backward Resolution** is shown as follows:

1. ¬pass(X,ai_exam)∨¬win(X,lottery)∨happy(X)
2. pass(Y,E)∨¬study(Y)
3. pass(M,G)∨¬lucky(M)
4. ¬study(john)
5. lucky(john).
6.¬lucky(Z)∨win(Z,lottery).
7.¬happy(john).

7: ¬happy(john) **1**:¬pass(X,ai_exam)∨¬win(X,lottery)∨ happy(X) {X=john}

8: ¬pass(john, ai_exam)∨¬win(john,lottery) **6**: ¬lucky(Z)∨win(Z,lottery){Z=john}

9: ¬pass(john, ai_exam)∨¬lucky(john) **5**: lucky(john)

10: ¬pass(john, ai_exam) **3**: pass(M,G)∨¬lucky(M) {M=john, G=ai_exam}

11: ¬lucky(john) **5**: lucky(john)

12: □ {the empty clause}

∴ John is happy

d_2) Forward Resolution

The proving for ***happy(john)*** using **Backward Resolution** is shown as follows:

1. ┐pass(X,ai_exam)∨┐win(X,lottery)∨happy(X)
2. pass(Y,E)∨┐study(Y)
3. pass(M,G)∨┐lucky(M)
4. ┐study(john)
5. lucky(john).
6.┐lucky(Z)∨win(Z,lottery).
7.┐happy(john).

1:┐pass(X,ai_exam)∨┐~~win~~(X,lottery)∨happy(X) **6**: ┐lucky(Z)∨~~win~~(Z, lottery){Z=X}

8: ┐ pass(X,ai_exam)∨happy(X) ∨ ┐~~lucky~~(X) **5**: ~~lucky~~(john) {X=john}

9:┐ ~~pass~~(john, ai_exam)∨happy(john) **3**: ~~pass~~(M,G)∨┐lucky(M) {M=john, G=ai_exam}

10: happy(john) ∨ ┐~~lucky~~(john) **5**: ~~lucky~~(john)

11 : ~~happy~~(john) 7: ┐~~happy~~(john)

12: □ {the empty clause}

∴ John is happy

Another Example

All people that are not poor and smart are happy. Those people that read are not stupid. John can read and wealthy. Happy people have exciting life. Can anyone found with an exciting life.

a)

$\forall X(\neg poor(X) \wedge smart(X)) \rightarrow happy(X)$

$read(Y) \rightarrow \neg\, stupid(Y)$

$read(john) \wedge\ wealthy(john)$

$happy(Z) \rightarrow exciting(Z,life)$

$\exists W\ exciting(W,life)$

Note $\neg$ stupid$\equiv$ smart, wealthy$\equiv\neg$poor

$\forall X(\neg poor(X) \wedge smart(X)) \rightarrow happy(X)$

$\forall Y\ read(Y) \rightarrow smart\ (Y)$

$read(john) \wedge\ \neg poor(john)$

$\forall Z\ happy(Z) \rightarrow exciting(Z,life)$

$\exists W\ exciting(W,life)$

b)

1.

$\forall X \quad \neg(\neg poor(X) \wedge smart(X)) \vee happy(X)$

$\forall Y\ \neg\ read(Y) \vee smart\ (Y)$

read(john)∧ ¬poor(john)

∀Z ¬ happy(Z)∨ exciting(Z,life)

∃W exciting(W,life)

2.

∀X (poor(X)∨¬smart(X))∨ happy(X)

∀X ¬read(Y)∨smart(Y)

read(john)∧ ¬poor(john)

∀Z ¬ happy(Z)∨ exciting(Z,life)

∃W exciting(W,life)

3. Nothing to do here.

4. Nothing to do here.

5.

∀X (poor(X)∨¬smart(X))∨ happy(X)

∀Y ¬read(Y)∨smart(Y)

read(john)∧ ¬poor(john)

∀Z ¬ happy(Z)∨ exciting(Z,life)

exciting(W,life)

6.

(poor(X)∨¬smart(X))∨ happy(X)

¬read(Y)∨smart(Y)

read(john)∧ ¬poor(john)

¬ happy(Z)∨ exciting(Z,life)

exciting(W,life)

7. Nothing to do here.

8.

poor(X)∨¬smart(X)∨ happy(X)

¬read(Y)∨smart(Y)

read(john)

¬poor(john)

¬ happy(Z)∨ exciting(Z,life)

exciting(W,life)

9. Nothing to do here.

c) ¬ exciting(W,life)

d)

d-1) Proving exciting(W,life) using ***Backward Resolution***

1. poor(X)∨¬smart(X)∨ happy(X)
2. ¬read(Y)∨smart(Y)
3. read(john)
4. ¬poor(john)
5. ¬ happy(Z)∨ exciting(Z,life)
6. ¬ exciting(W,life)

6: ¬exciting(W,life) 5: ¬ happy(Z)∨ exciting(Z,life) {W =Z}

7: ¬ happy(Z) 1: poor(X)∨¬smart(X)∨ happy(X) {Z=X}

8: poor(X)∨¬smart(X) 2: ¬read(Y)∨smart(Y) {Y=X}

9: poor(X)∨¬ read(X) 3: read(john) {X=john}

10: poor(john) 4: ¬poor(john)

11: □ {the empty clause}

∴ Anyone Can found with an exciting life.

d-2) Proving exciting(W,life)using ***Forward Resolution***

1. **poor(X)∨˥smart(X)∨ happy(X)**
2. **˥read(Y)∨smart(Y)**
3. **read(john)**
4. **˥poor(john)**
5. **˥ happy(Z)∨ exciting(Z,life)**
6. **˥ exciting(W,life)**

1: poor(X)∨˥smart(X)∨ happy(X) 2: ˥read(Y)∨smart(Y) {Y=X}

7: poor(X)∨ happy(X) ∨ ˥read(X) 3: read(john) {X=john}

8: poor(john)∨ happy(john) 5: ˥ happy(Z)∨ exciting(Z,life) {Z=**john**}

9: poor(john)∨ exciting(john,life) 6: ˥ exciting(W,life) {W=**john**}

10: poor(john) 4: ˥poor(john)

11: □ {the empty clause}

∴ Anyone Can found with an exciting life.

<u>Homework</u>

Everyone has a parent. The parent of a parent is a grandparent. Prove that Ali has a grandparent using Backward Resolution.

Example: Given the following two statements:

(1) **∀X. [(∀Y. s(Y) ∧ v(X, Y)) ⇒ ((∃Z. ¬ t(X, Z)) ∧ v(X, X))]**

(2) ∀X∀Y. s(Y) ⇒ t(X,Y) ∧ v(X,Y)

(A) Convert the statements to clause forms:

Solution:

(1) $\forall X.\ [\ (\forall Y.\ s(Y) \wedge v(X, Y)) \Rightarrow ((\exists Z.\ \neg\ t(X, Z)) \wedge v(X, X))\ ]$

1-1) **Remove ⇒**: $\forall X\ [\neg(\forall X.\ s(Y) \wedge v(X,Y)) \vee ((\exists Z.\ \neg\ t(X,Z)) \wedge v(X,X))\ ]$

1-2) **Reduce ¬:** $\forall X\ [\ (\exists Y.\ \neg s(Y) \vee \neg v(X,Y)) \vee ((\exists Z.\ \neg\ t(X,Z)) \wedge v(X,X))\ ]$

1-3) **Standardize:** Nothing.

1-4) **Move quantifiers:** $\forall X\ \exists Y\ \exists Z\ [\ (\neg s(Y) \vee \neg v(X,Y)) \vee (\neg t(X,Z)) \wedge v(X,X)]$

1-5) **Remove ∃:** $\forall X\ [(\neg s(f1(X)) \vee \neg v(X,f1(X)) \vee (\neg\ t(X,f2(X)\ \ \wedge v(X,X))\ ]$

1-6) **Remove ∀:** $(\neg s(f1(X)) \vee \neg v(X,f1(X)) \vee (\neg\ t(x,f2(X)\ \ \wedge v(X,X))$

1-7) **CNF:** $(\neg s(f1(X)) \vee \neg v(X,f1(X)) \vee \neg\ t(X,f2(X))) \wedge (\neg s(f1(X)) \vee \neg v(x,f1(X)) \vee v(X,X))$

1-8) **Split** ∧: $\neg s(f1(X)) \vee \neg v(X,f1(X)) \vee \neg\ t(X,f2(X))$

$\neg s(f1(X)) \vee \neg v(X,f1(X)) \vee v(X,X)$

1-9) **Standardize:** $\neg s(f1(X)) \vee \neg v(X,f1(X)) \vee \neg\ t(X,f2(X))$

$\neg s(f3(X3)) \vee \neg v(X3,f3(X3)) \vee v(X3,X3)$

(2) $\forall N \forall M.\ s(M) \Rightarrow t(N, M) \wedge v(N, M)$

2-1) **Remove ⇒:** $\forall N \forall M.\ \neg s(M) \vee (t(N, M) \wedge v(N, M))$

2-2) 2-3)2-4)2-5) Nothing

2-6) **Remove ∀**:$\neg s(M) \vee (t(N, M) \wedge v(N, M))$

2-7) **CNF:** $(\neg s(M) \vee t(N, M)) \wedge (\neg s(M) \vee v(N, M))$

2-8) **Split** ∧: $\neg s(M) \vee t(N, M)$

$\neg s(M) \vee v(N, M)$

2-9) **Standardize:** $\neg s(M) \vee t(N, M)$

¬s(M1) ∨ v(N, M1)

After applying the clause form method, the two sentences become four clauses as follows:

1. ¬s(f1(X)) ∨ ¬v(X,f1(X)) ∨ ¬ t(X,f2(X))
2. ¬s(f3(X3)) ∨ ¬v(X3,f3(X3)) ∨ v(X3,X3)
3. ¬s(M) ∨ t(N, M)
4. ¬s(M1) ∨ v(N1, M1)

(B) Prove by resolution refutation that: ∃ W. ¬ s(W)

Solution: We must add the negation of what is to be proved. Thus, ∃ W. ¬ s(W) become s(W) because ¬(∃ W. ¬ s(W))= ∀ W s(W) = s(W). Now we have a new clause (5. s(W).) added to the other four clauses and the proving is shown below:

1. ¬s(f1(X)) ∨ ¬v(X,f1(X)) ∨ ¬ t(X,f2(X))
2. ¬s(f3(X3)) ∨ ¬v(X3,f3(X3)) ∨ v(X3,X3)
3. ¬s(M) ∨ t(N, M)
4. ¬s(M1) ∨ v(N1, M1)
5. s(W)

5: s(W) 2: ¬s(f3(X3)) ∨ ¬v(X3,f3(X3)) ∨ v(X3,X3) {f3(X3)=W}

6: ¬v(X3,W) ∨ v(X3,X3) 4: ¬s(M1) ∨ v(N1, M1) { N1=X3, M1=W}

7: v(X3,X3) ∨ ¬s(W) 1: ¬s(f1(X)) ∨ ¬v(X,f1(X)) ∨ ¬ t(X,f2(X)) {X=X3, f1(X)=X3}

8: ¬s(W) ∨ ¬s(X3) ∨ ¬ t(X3,f2(X)) 3: ¬s(M) ∨ t(N, M) { N=X3,M=f2(X)}

9: ¬s(W) ∨ ¬s(X3) ∨ ¬s(f2(X)) 5. s(W) {f2(X)=W}

10: ¬s(W) ∨ ¬s(X3) 5. s(W) {X3=W}

11: ¬s(W) 5. s(W)

12: ( ) { Empty clause}

Example: Use the resolution to prove $\boldsymbol{b \wedge c}$ using the following sentences: $\boldsymbol{a \rightarrow (c \vee d)}$, $\boldsymbol{b \rightarrow a}$, $\boldsymbol{d \rightarrow c}$, and $\boldsymbol{b}$.

Solution:

(1) $\neg a \vee c \vee d$

(2) $\neg b \vee a$

(3) $\neg d \vee c$

(4) b

(5) $\neg b \vee \neg c$

(6) $\neg c$ (4) and (5)

(7) $\neg d$ (3) and (6)

(8) $\neg a \vee c$ (1) and (7)

(9) $\neg a$ (6) and (8)

(10) $\neg b$ (2) and (9)

(11) *nil* (4) and (10)

Control Strategy and Expert Systems

What is Control Strategy?

Control Strategy (also called production system) is a system based on IF… THEN… rules and consisting of three parts:

1. The production rules: A production rule is condition-action pair presented in the following form "IF condition THEN action", and it represents as a single chunk of problem-solving knowledge. The condition part of the rule is a pattern that determines when the rule may be applied to a problem. The action part defines the associated problem-solving step.

2. The working memory: It contains a description of the current state of the problem-solving.

3. The control structure: It implements search allowing the production system to move towards a goal within the set of rules. The control structure also called an interpreter or a recognize-act cycle.

Control Strategy types:

There are two broad kinds of control strategy: ***forward and backward chaining* systems**. In a forward chaining system you start with the initial facts, and keep using the rules to draw new conclusions (or take certain actions) given those facts. In a backward chaining system you start with some hypothesis (or goal) you are trying to prove, and keep looking for rules that would allow you to conclude that hypothesis, perhaps setting new subgoals to prove as you go. Forward chaining systems are primarily data-driven, while backward chaining systems are goal-driven.

Forward Chaining System

In a forward chaining system the facts in the system are represented in a working memory which is continually updated. Rules in the system

represent possible actions to take when specified conditions hold on items in the working memory - they are sometimes called condition-action rules. The conditions are usually patterns that must match items in the working memory, while the actions usually involve adding or deleting items from the working memory. The control structure will control the application of the rules, given the working memory, thus controlling the system's activity. It is based on a cycle of activity sometimes known as a recognize-act cycle. The system first checks to find all the rules whose conditions hold, given the current state of working memory. It then selects one and performs the actions in the action part of the rule. The selection of a rule to fire is based on fixed strategies, known as ***conflict*** resolution strategies. The actions will result in a new working memory, and the cycle begins again. This cycle will be repeated until either no rules ***fire***, or some specified goal state is satisfied. Rule-based systems vary greatly in their details and syntax, so the following examples are only illustrative.

First we'll look at a very simple set of rules:

1. IF *lecturing(X)* AND *marking-practicals(X)* THEN ADD (*overworked(X)*)
2. IF *month(february)* THEN ADD (*lecturing(john)*)
3. IF *month(february)* THEN ADD (*marking-practicals(john)*)
4. IF *overworked(X)* OR *slept-badly(X)* THEN ADD (*bad-mood(X)*)
5. IF *bad-mood(X)* THEN DELET (*happy(X)*)
6. IF *lecturing(X)* THEN DELET (*researching(X)*)

Let us assume that initially we have a working memory with the following facts:

month(february)

happy(john)

researching(john)

Production system will first go through all the rules checking which ones apply given the current working memory. **Rules 2** and **3** both apply, so the system has to choose between them, using its conflict resolution strategies. Let us say that **rule 2** is chosen. So, *lecturing(john)* is added to the working memory, which is now:

lecturing(john)

month(february)

happy(john)

researching(john)

Now the cycle begins again. This time **rule 3** and **rule 6** have their preconditions satisfied. Lets say **rule 3** is chosen and fires, so *marking-practicals(john)* is added to the working memory.

On the third cycle **rule 1** fires, so, with **X** bound to **john**, *overworked (john)* is added to working memory which is now:

overworked(john)

marking-practicals(john)

lecturing(john)

month(february)

happy(john)

researching(john)

Now **rules 4** and **6** can apply. Suppose **rule 4** fires, and *bad-mood(john)* is added to the working memory.

And in the next cycle **rule 5** is chosen and fires, with *happy(john)* removed from the working memory.

Finally, **rule 6** will fire, and *researching(john)* will be removed from working memory, to leave:

bad-mood(john)

overworked(john)

marking-practicals(john)

lecturing(john)

month(february)

The five facts in the working memory imply that there is a person called "john" that works as lecturing and marking-practical at the same time in the February month and this cause an overworked load and thus he has a bad-mood state.

Backward Chaining System

So far we have looked at how rule-based systems can be used to draw new conclusions from existing data, adding these conclusions to a working memory. This approach is most useful when you know all the initial facts, but don't have much idea what the conclusion might be. If you do know what the conclusion might be, or have some specific hypothesis to test, forward chaining systems may be inefficient. You could keep on forward chaining until no more rules apply or you have added your hypothesis to the working memory. But in the process the system is likely to do a lot of irrelevant work, adding uninteresting conclusions to working memory. For example, suppose we are interested in whether **john** is in a **bad-mood**. We could repeatedly fire rules, updating the working memory, checking each time whether (**bad-mood john**) is in the new working memory. But maybe we had a whole batch of rules for drawing conclusions about what happens when I'm **lecturing**, or what happens in **February** - we really don't care about this, so would rather only have to draw the conclusions that are relevant to the goal. This can be done by backward chaining from the goal

state (or on some hypothesized state that we are interested in). This is essentially what Prolog does, so it should be fairly familiar to you by now. Given a goal state to try and prove (e.g., *bad-mood(john)*) the system will first check to see if the goal matches the initial facts given. If it does, then that goal succeeds. If it doesn't the system will look for rules whose conclusions (previously referred to as actions) match the goal. One such rule will be chosen, and the system will then try to prove any facts in the preconditions of the rule using the same procedure, setting these as new goals to prove. Note that a backward chaining system does NOT need to update a working memory. Instead it needs to keep track of what goals it needs to prove to prove its main hypothesis. In principle we can use the same set of rules for both forward and backward chaining. However, in practice we may choose to write the rules slightly differently if we are going to be using them for backward chaining. In backward chaining we are concerned with matching the conclusion of a rule against some goal that we are trying to prove. So the 'then' part of the rule is usually not expressed as an action to take (e.g., add/delete), but as a state which will be true if the premises are true.

So, suppose we have the following rules:

1. IF *lecturing(X)* AND *marking-practicals(X)* THEN *overworked(X)*
2. IF *month(february)* THEN *lecturing(john)*
3. IF *month(february)* THEN *marking-practicals(john)*
4. IF *overworked(X)* THEN *bad-mood(X)*
5. IF *slept-badly(X)* THEN *bad-mood(X)*

And there is only one initial fact in the worked memory is: *month(february),* and we're trying to prove *bad-mood(john)*

First we check whether the goal state (*bad-mood(john)*) is in the initial facts in the working memory. As it isn't there, we will add the goal to working memory and then we try matching it against the conclusions of the rules in

the Production Rules. It matches **rules 4** and **5**. Let us assume that **rule 4** is chosen first - it will try to prove *overworked(john)*. **Rule 1** can be used, and the system will try to prove *lecturing(john)* and *marking-practicals(john)*. Trying to prove the first goal, it will match **rule 2** and try to prove *month(february)*. This is a fact in the working memory. We still have to prove *marking-practicals(john)*. **Rule 3** can be used, try to prove *month(february)*. This is a fact in the working memory and in this place we have proved the original goal *bad-mood(john).*

One way of implementing this basic mechanism is to use a stack of goals still to satisfy. You should repeatedly pop a goal of the stack, and try and prove it. If its in the set of initial facts then its proved. If it matches a rule which has a set of preconditions then the goals in the precondition are pushed onto the stack. Of course, this doesn't tell us what to do when there are several rules which may be used to prove a goal. If we were using Prolog to implement this kind of algorithm we might rely on its backtracking mechanism - it'll try one rule, and if that results in failure it will go back and try the other. However, if we use a programming language without a built in search procedure we need to decide explicitly what to do. One good approach is to use an agenda, where each item on the agenda represents one alternative path in the search for a solution. The system should try expanding each item on the agenda, systematically trying all possibilities until it finds a solution (or fails to). The particular method used for selecting items off the agenda determines the search strategy - in other words, determines how you decide on which options to try, in what order, when solving your problem. We'll go into this in much more detail in the section on search.

Forwards vs. Backwards Reasoning

Whether you use forward or backward reasoning to solve a problem depends on the properties of your rule set and initial facts.

Forward chaining is the best choice if:

1- All the facts are provided with the problem statement; or

2- There are many possible goals, and a smaller number of patterns of data; or
3- There isn't any sensible way to guess what the goal is at the beginning of the consultation.

Backward chaining is the best choice if:

1- The goal is given in the problem statement; or
2- The system has been built so that it asks for pieces of data rather than expecting all the facts to be presented to it.

Example1: Suppose you have the following production rules:

1. IF John is a student THEN John enjoys student's life

2. IF John enjoys student's life

   THEN John meets friends AND John participates in university's events

3. IF John meets friends THEN John needs money

4. IF John needs money THEN John has a job

5. IF John meets friends AND John participates in university's events

   THEN John has little free time

6. IF John has little free time AND John has a job

   THEN John is not successful in studies

   AND John does not receive scholarship

Trace the Forward Chaining Algorithm using the given production rules to get the goal "John does not receive scholarship" from the start sentence that is "John is a student". Show the contents of the Working Memory and the Conflict Set (i.e., all the rules that match the facts in the Working Memory) and the Rule Fired (i.e., select one member of conflict set to execute).

Note: You can rewrite all the sentences to atomic sentences (for example; John is a student become as john_is_a_student) or you can only write them as letters (for example; John is a student will be **A** and John enjoys student's life is **B** and so on).

Solution:

At the first, we can assume that each sentence just as a letter, so the production rules will become as follows:

[redacted] A B

1. **IF** John is a student **THEN** John enjoys student's life

B C

2. **IF** John enjoys student's life **THEN** John meets friends **AND**

D

John participates in university's events

C E

3. **IF** John meets friends **THEN** John needs money

E F

4. **IF** John needs money **THEN** John has a job

C D

5. **IF** John meets friends **AND** John participates in university's events

G

THEN John has little free time

G F H

6. **IF** John has little free time **AND** John has a job **THEN** John is not successful in studies **AND**

I

John does not receive scholarship

1. **IF** A **THEN** B
2. **IF** B **THEN** C **AND** D
3. **IF** C **THEN** E
4. **IF** E **THEN** F
5. **IF** C **AND** D **THEN** G
6. **IF** G **AND** F **THEN** H **AND** I

Start - A
Goal - I

The tracing for this algorithm using the given production rules will be shown as follows:

Iteration	Working memory	Conflict set	Fired rule
0	A	1	1
1	A, B	1, 2	2
2	A, B, C, D	1, 2, 3, 5	5
3	A, B, C, D, G	1, 2, 3, 5	3
4	A, B, C, D, G, E	1, 2, 3, 5, 4	4
5	A, B, C, D, G, E, F	1, 2, 3, 5, 4, 6	6
6	A, B, C, D, G, E, F, H, I	1, 2, 3, 5, 4, 6	goal

Then there are 8 facts which must to be found to reach to the goal I="John does not receive scholarship" and are:

1- A=John is a student.
2- B=John enjoys student's life.
3- C=John meets friends.
4- D=John participates in university's events.
5- G=John has little free time.
6- E=John needs money.
7- F=John has a job.
8- H=John is not successful in studies.

The production rules that are used to get these facts are {1,2,3,5,4,6}.

Example2: Suppose you have the following production rules:

1. **IF** p **AND** q **THEN** goal
2. **IF** r **AND** s **THEN** p
3. **IF** w **AND** r **THEN** p
4. **IF** t **AND** u **THEN** q
5. **IF** v **THEN** s
6. **IF** start **THEN** v **AND** r **AND** q

---

a) Write the Forward Chaining Algorithm and then trace this algorithm using the given production rules to verify the goal through the start fact.

b) Write the Backward Chaining Algorithm and then trace this algorithm using the given production rules to verify the start fact through the goal.

Solution:

<u>a) Forward Chaining Algorithm (Data-driven search algo.):</u>

1. Begins with a pattern (a problem description) added to the working memory.

2. The control structure compares matching of the pattern with IF part of rules in the production rules.

3. Firing a rule, its THEN part is added to the working memory and the process continues.

4. Search stops when the goal is found.

The tracing for this algorithm using the given production rules will be shown as follows:

Iteration #	Working memory	Conflict set	Rule fired
0	start	6	6
1	start, v, r, q	6, 5	5
2	start, v, r, q, s	6, 5, 2	2
3	start, v, r, q, s, p	6, 5, 2, 1	1
4	start, v, r, q, s, p, goal	6, 5, 2, 1	halt

b) Backward Chaining Algorithm (Goal-driven search algo.)::

1. A goal (a pattern) is added to the working memory.

2. The control structure compares matching of the pattern with THEN part of rules in the production rules.

3. Firing a rule, its IF part is added to the working memory and the process continues.

4. Search stops when facts on problem are found.

The tracing for this algorithm using the given production rules will be shown as follows:

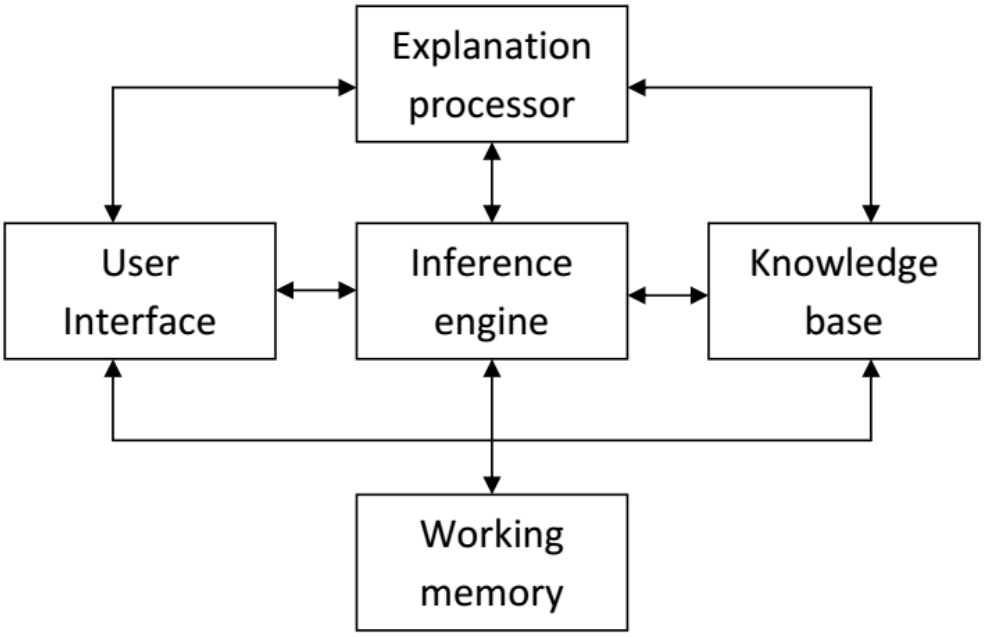
Iteration #	Working memory	Conflict set	Rule fired
0	goal	1	1
1	goal, p, q	1, 2, 3, 4	2
2	goal, p, q, r, s	1, 2, 3, 4, 5	3
3	goal, p, q, r, s, w	1, 2, 3, 4, 5	4
4	goal, p, q, r, s, w, t, u	1, 2, 3, 4, 5	5
5	goal, p, q, r, s, w, t, u, v	1, 2, 3, 4, 5, 6	6
6	goal, p, q, r, s, w, t, u, v, start	1, 2, 3, 4, 5, 6	halt

What are 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.

What is Expert System Architecture and Components?

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



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.

2-Explanation Processor

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).

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.

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-Working Memory

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

Appendix-A

Introduction to Prolog Language

1. Simple prolog programs (Facts & Rules):

Example1:

```
domains
  st=string
predicates
  father(st,st)
  grandfather(st,st)
clauses
  father("Ali","Zaki").
  father("Zaki","Suha").
  grandfather(X,Y):-
    father(X,Z),
    father(Z,Y).
goal:grandfather(X,Y).%X=Ali, Y=Suha
```

Example2:

```
domains
  i=integer
predicates
  sum(i,i,i).
clauses
  sum(0,S,S):-!.
  sum(N,X,S):-
    X1=X+N,
    N1=N-1,
    sum(N1,X1,S).
goal: sum(7,0,S). S=28
```

Example3:

```
domains
  i=integer
predicates
  sum(i,i,i).
  run.
clauses
  run:-
    write("Enter any number:\n"),
    readint(N),
    sum(N,0,S),
    write("Sumation =",S),nl.
```

```
  sum(0,S,S):-!.
  sum(N,X,S):-
    N1=N-1,
    X1=X+N,
    sum(N1,X1,S).
goal
    run.
Enter any number:
5
Sumation =15
yes
```

Example4:

```
domains
  i=integer
predicates
  sum(i,i,i).
clauses
  sum(0,S,S):-!.
  sum(N,X,S):-
    write("enter digit:\n"),
    readint(I),
    X1=X+I,
    N1=N-1,
    sum(N1,X1,S).
goal
    sum(5,0,S).
enter digit:
2
enter digit:
3
enter digit:
6
enter digit:
4
enter digit:
5
S=20
1 Solution
```

Example5:

```
domains
  i=integer
predicates
  sum(i,i,i).
  g.
```

```
clauses
  g:-write("Enter number of digit for sumation:\n"),
    readint(N),sum(N,0,S),
    write("Sumation=",S),nl.
  sum(0,S,S):-!.
  sum(N,X,S):-
    write("enter digit:\n"),
    readint(I),
    X1=X+I,
    N1=N-1,
    sum(N1,X1,S).
goal
    g.
Enter number of digit for sumation:
2
enter digit:
4
enter digit:
6
Sumation=10
yes
```

Example6:

```
domains
  i=integer
predicates
  sum(i,i).
clauses
  sum(0,S):-write(S),nl,!.
  sum(N,S):-
    write("Enter digit"),nl,
    readint(I),
    S1=S+I,
    N1=N-1,
    sum(N1,S1).
Goal:     sum(3,0).
```

Example7:

```
domains
  i=integer
predicates
  sum(i,i).
  g.
clauses
  g:-write("Enter number of digit for sumation:\n"),
    readint(N),sum(N,0).
```

```
  sum(0,S):-write("Sumation=",S),nl,!.
  sum(N,S):-
    write("Enter digit"),nl,
    readint(I),
    S1=S+I,
    N1=N-1,
    sum(N1,S1).
Goal:   g.
Enter number of digit for sumation:
3
Enter digit
4
Enter digit
1
Enter digit
5
Sumation=10
yes
```

---

2. Lists.

```
Example1
domains
  i=integer
  l=i*
predicates
  member(i,l).
clauses
     member(X,[X|_]):-!.
     member(X,[_|T]):-
        member(X,T).
goal:
   member(4,[1,2,4,6,8]).%yes
-------------------------------------------------------
Example2
domains
l=i*.
i=integer.
predicates
 length(l,i,i).
clauses
  length([],N,N):-!.
  length([_|T],N1,N):-
      N2=N1+1,
```

```
    length(T,N2,N).
goal:
length([1,2,7,3,4],0,Length).Length=5
-------------------------------------------------------
Example3
domains
l=i*.
i=integer.
predicates
  delOnce(i,l,l).
clauses
  delOnce(_,[],[]):-!.
  delOnce(X,[X|T],T):-!.
   delOnce(X,[H|T1],[H|T2]):-
      delOnce(X,T1,T2).
goal:
  delOnce(3,[1,3,3,5,3],X).%X=[1,3,5,3]
```

```
Example4
domains
l=i*.
i=integer.
predicates
  delAll(i,l,l).
clauses
  delAll(_,[],[]):-!.
  delAll(X,[X|T1],T2):-delAll(X,T1,T2),!.
   delAll(X,[H|T1],[H|T2]):-
      delAll(X,T1,T2).
goal:
delAll(3,[1,3,3,5,3],X).%X=[1,5]
-----------------------------------------------------------------
Example5
domains
i=integer
l=i*
predicates
delAtLoc(i,l,l)
clauses
   delAtLoc(1,[_|T],T):-!.
   delAtLoc(N,[H|T],[H|T1]):-
      N1=N-1,
      delAtLoc(N1,T,T1).

goal
    delAtLoc(2,[1,9,5,2,8],X).%X=[1,5,2,8]
```

```
-----------------------------------------------------------------
Example6
% find an element from a list in a specific location
Domains
i=integer
s=symbol
l=s*
Predicates
n_th(i,l,s)
Clauses
n_th(1,[X|_],X).
n_th(N,[_|T],X):-
  N>1,N1=N-1,
  n_th(N1,T,X).

goal
   n_th(3,[a,b,c,d],X).%X=c
-----------------------------------------------------------------
Example7
domains
l=i*.
i=integer.
predicates
append(l,l,l).
clauses
append([],L,L):-!.
append([H|T1],X,[H|T2]):-
    append(T1,X,T2).
goal:
 append([1,2],[3,4],X).%X=[1,2,3,4].
-----------------------------------------------------------------
domains
l=i*.
i=integer.
predicates
getout(l,l).
member(i,l).
clauses
 getout([],[]):-!.
 getout([H|T1],[H|T2]):-
    not(member(H,T1)),!,
    getout(T1,T2).
 getout([_|T],L):-
    getout(T,L).
 member(H,[H|_]):-!.
 member(H,[_|T]):-
```

```
    member(H,T).
goal:
  %getout([1,3,1,2,5,3],X).%X=[1,2,5,3]
  getout([1,2,3,4,1,3,5,2]).X=[4,1,3,5,2]
 ---------------------------------------------------------
Example8
domains
l=i*.
i=integer.
predicates
dif(l,l,l).
member(i,l).
clauses
  dif([],_,[]).
  dif([H|T1],L,[H|T2]):-
    not(member(H,L)),!,
    dif(T1,L,T2).
  dif([_|T],L1,L2):-
    dif(T,L1,L2).
  member(H,[H|_]):-!.
  member(H,[_|T]):-
     member(H,T).
goal:
   dif([1,2,3,1],[5,1,3,9,3],X).%X=[2]
---------------------------------------------------------
Example9
domains
l=i*.
i=integer.
predicates
 max(l,i,i).
clauses
 max([],M,M):-!.
 max([H|T1],N,M):-
      H>N,!,
      max(T1,H,M).
 max([_|T],N,M):-
      max(T,N,M).
goal:
   max([1,4,15,3,1,7],-2147483648,Max).%Max=15
-------------------------------------------------------

Example10
domains
l=i*.
i=integer.
```

```
predicates
  member(i,l).
  intersection(l,l,l).
clauses
  intersection([],_,[]).
  intersection([H|T1],L,[H|T2]):-
     member(H,L),!,
     intersection(T1,L,T2).
  intersection([_|T],L1,L2):-
     intersection(T,L1,L2).
   member(H,[H|_]):-!.
  member(H,[_|T]):-
      member(H,T).
goal:
  intersection([3,1,5,2],[2,1,6,3],X).X=[3,1,2]
-----------------------------------------------------
Example11
domains
l=i*.
i=integer.
predicates
  member(i,l).
  union(l,l,l).
clauses
  union([],L,L).
  union([H|T],L1,L2):-
     member(H,L1),!,
     union(T,L1,L2).
  union([H|T1],L,[H|T2]):-
     union(T1,L,T2).
  member(H,[H|_]):-!.
  member(H,[_|T]):-
      member(H,T).
goal:
  union([3,1,5,2],[2,1,6,3],X).X=[5,2,1,6,3]
----------------------------------------------------
Example12
domains
l=i*.
i=integer.
predicates
sort(l,l).
min(l,i,i).
delOnce(i,l,l).
clauses
  sort([],[]):-!.
```

```
  sort(L,[M|T]):-
    min(L,2147483647,M),
    delOnce(M,L,L1),
    sort(L1,T).
  min([],M,M):-!.
  min([H|T1],N,M):-
      H<N,!,
      min(T1,H,M).
  min([_|T],N,M):-
      min(T,N,M).
  delOnce(X,[X|T],T):-!.
  delOnce(X,[H|T1],[H|T2]):-
       delOnce(X,T1,T2).
goal:
  sort([3,1,7,5,2,6,5,10],Sort).%Sort=[1,2,3,5,5,6,7,10]
```

---

3. Strings

```
Example1
domains
  s=string.
  i=integer.
predicates
  str_length(s,i).
clauses
  str_length(S,N):-
    str_len(S,N).
goal:
    str_length("Computer Science",N).%N=16
```

```
Example2
domains
  s=string.
  i=integer.
predicates
  char_count(s,i,i).
clauses
  char_count("",N,N):-!.
  char_count(S,N,X):-
    frontchar(S,_,R),
         N1=N+1,
    char_count(R,N1,X).
goal:
    char_count("Computer Science",0,N).%N=16
```

Example3

```
domains
  s=string.
  i=integer.
predicates
  token_count(s,i,i).
clauses
  token_count("",N,N):-!.
  token_count(S,N,X):-
    fronttoken(S,_,R),
        N1=N+1,
    token_count(R,N1,X).
goal:    token_count("Computer Science",0,N).%N=2
```

Example4

```
domains
  s=string.
  i=integer.
predicates
  token_count(s,i).
clauses
  token_count("",0):-!.
  token_count(S,N):-
    fronttoken(S,_,R),
    token_count(R,N1),
    N=N1+1.
goal:
    token_count("Computer Science",N).%N=2
```

Example5

```
domains
  s=string.
  i=integer.
predicates
  sentence_count(s,i).
clauses
  sentence_count("",0):-!.
  sentence_count(S,N):-
    frontchar(S,C,R),
    C='.',!,
    sentence_count(R,N1),
    N=N1+1.
  sentence_count(S,N):-
    frontchar(S,_,R),
    sentence_count(R,N).
goal:
    sentence_count("Computer Science.Information system.Third stage.",N).%N=3
```

Example6

```
domains
  s=string.
predicates
  rev(s,s).
clauses
  rev("","")):-!.
  rev(S,S1):-
      str_len(S,L),
      N=L-1,
      frontstr(N,S,P,R),
      rev(P,S2),
      concat(R,S2,S1).
goal:
    rev("Artificial Intelligence",S).
```

Example7

```
predicates
split_tokens(string)
run(string)
clauses
run(S):-
  split_tokens(S).

split_tokens(S):-
   fronttoken(S,W,R),
   isname(W),!,
   write("String=",W),nl,
   split_tokens(R).

split_tokens(S):-
   fronttoken(S,W,R),
   str_int(W,N),!,
   write("Integer=",N),nl,
   split_tokens(R).

split_tokens(S):-
   fronttoken(S,W,R),
   str_real(W,N),!,
   write("Real=",N),nl,
   split_tokens(R).
split_tokens("").
 goal
    run("He A$& bought 7 oranges  their total  weight 1.5 kg").
```

```
/* String=He
String=bought
Integer=7
String=oranges
String=their
String=total
String=weight
Real=1.5
String=kg     Yes*/
```

---

4. Database

Example1:

1- **Assert predicate:**

- **assert(X) or assertz(X) :Adds a new fact to the database. Term is asserted as the last fact with the same key predicate.**
  - ✓ **For example;**

```
domains
s=string.
ls=s*.
database
person(s)
predicates
list_preson(ls)
clauses
list_preson(L):-
    assert(person ("Ali")),
    assert(person ("Zaki")),
    assert(person ("Suha")),
    findall(X,person(X),L).

goal: list_preson(L).
%L=["Ali","Zaki","Suha"]
```

- **asserta(X) :Same as assert, but adds a fact X at the beginning of the database.**
  - ✓ **For example;**

```
domains
s=string.
ls=s*.
database
person(s)
predicates
list_preson(ls)
clauses
list_preson(L):-
        asserta(person ("Ali")),
```

```
        asserta(person ("Zaki")),
        asserta(person ("Suha")),
        findall(X,person(X),L).

goal: list_preson(L).  %L=["Suha","Zaki","Ali"]
```

2- Retract predicate:

- **retract(X): Removes a fact X from the database.**
  - ✓ **For example;**

```
domains
s=string.
ls=s*.
database
person(s)
predicates
list_preson(ls)
clauses
list_preson(L):-
        assert(person ("Ali")),
        assert(person ("Zaki")),
        assert(person ("Suha")),
        retract(person ("Zaki")),
        findall(X,person(X),L).

goal: list_preson(L).
%L=["Ali","Suha"]
```

- **retractall(X): Removes all facts from the database for which the head unifies with X.**
  - ✓ **For example;**

```
domains
s=string.
ls=s*.
database
person(s)
predicates
list_preson(ls)
clauses
list_preson(L):-
        assert(person ("Ali")),
        assert(person ("Zaki")),
        assert(person ("Suha")),
        retractall(person (_)),% retractall(_),
        findall(X,person(X),L).

goal: list_preson(L).
%L=[]
```

Example2: Insert the following facts to a database then put them in a list. The facts are:

f(1).
f(2).
f(3).
Solution:

```
domains
 i=integer
 f=f(i).
 lf=f*.
database
 f(i).
predicates
 run(lf).
 g(f).
 test
clauses
 test:-
 assertz(f(1)),
 assertz(f(2)),
 assertz(f(3)).

g(f(X)):-f(X).
run(L):-test,
      findall(S,g(S),L).

goal:
   run(X).  %X=[f(1),f(2),f(3)]
```

Example3: Use a database concept to perform the following goal:
Goal: run("He bought 7 oranges their total weight 1.5 kg").
And give the following output:

```
String=   He      length=   2
String=   bought      length=   6
String=   oranges      length=   7
String=   their      length=   5
String=   total      length=   5
String=   weight      length=   6
String=   kg      length=   2
```

Solution:

```
database
db_string(String,integer)
predicates
```

```
split_tokens(string)
run(string)
print_string
clauses
run(S):-retractall(_),
  split_tokens(S),
  print_string.

split_tokens(S):-
   fronttoken(S,W,R),
   isname(W),!,str_len(W,N),
   assert(db_string(W,N)),
   split_tokens(R).
split_tokens(S):-
   fronttoken(S,_,R),!,split_tokens(R).
split_tokens("").

print_string:-
db_string(S,N),write("String=   ",S,"      length=   ",N),nl,fail.
print_string.

goal
    run("He bought 7 oranges  their total  weight 1.5 kg").
/*String=   He      length=   2
String=   bought      length=   6
String=   oranges      length=   7
String=   their      length=   5
String=   total      length=   5
String=   weight      length=   6
String=   kg      length=   2
yes
 */
```

5. Compound Object

Example1:

```
domains
   predecessor=parent(father,son);child(string).
   father=father(string).
   son=son(predecessor).
predicates
  father(string,string).
  grandfather(predecessor).
clauses
  father("Ali","Zaki").
  father("Zaki","Suha").
```

```
 grandfather(parent(X,son(parent(Y,son(Z))))):-
   father(X1,Y1),
   father(Y1,Z1),
   X=father(X1),
   Y=father(Y1),
   Z=child(Z1).

goal:
   grandfather(X).
   %X=parent(father("Ali"),son(parent(father("Zaki"),son(child("Suha")))))
```

Example2:

```
domains
st=s(symbol,integer)
l=st*
predicates
member(st,l)
clauses
member(s(A,_),[s(A,_)|_]):-!.
member(X,[_|T]):-
      member(X,T).

goal
  member(s(a,5),[s(c,5),s(a,3),s(d,6)]).%yes
```

Example3:

```
domains
st=s(symbol,integer)
l=st*
predicates
member(st,l,integer,integer)
clauses

member(s(A,_),[s(A,_)|_],N,N):-!.
member(X,[_|T],N1,N):-
      N2=N1+1,member(X,T,N2,N).

goal
% member(s(a,5),[s(c,5),s(a,3),s(d,6)],1,N).%N=2
member(s(a,5),[s(c,5),s(g,3),s(d,6),s(a,7)],1,N).%N=4
%member(s(a,2),[s(a,5),s(g,3),s(d,6),s(c,7)],1,N).%N=1
%member(s(f,2),[s(a,5),s(g,3),s(d,6),s(c,7)],1,N).%No Solution
```

Example4:

```
domains
```

```
st=s(symbol,integer)
l=st*
predicates
member(st,l,st)
clauses
member(s(A,_),[s(A,X)|_],s(A,X)):- !.
member(X,[_|T],Z):-
      member(X,T,Z).

goal
  member(s(a,5),[s(c,5),s(a,3),s(d,6)],X).%X=s("a",3)
  %member(s(a,3),[s(c,5),s(a,5),s(d,6)],X).%X=s("a",5)
  %member(s(d,1),[s(c,5),s(a,5),s(d,6)],X).%X=s("d",6)
  %member(s(c,1),[s(c,5),s(a,5),s(d,6)],X).%X=s("c",5)
```

Example5:

```
domains
st=s(symbol,integer)
l=st*
i=integer
predicates
del(st,l,l)
clauses
del(s(A,_),[s(A,_)|L],L):-!.
del(X,[H|T],[H|Z]):-
  del(X,T,Z).
goal:
%del(s(g,9),[s(a,5),s(g,3),s(d,6),s(c,7)],X).%X=[s("a",5),s("d",6),s("c",7)]
```

Example6:

```
domains
st=s(symbol,integer)
l=st*
predicates
replace_lesser(st,l,l)
clauses

replace_lesser(s(A,X),[s(A,Y)|T],[s(A,X)|T]):-X<Y,!.
replace_lesser(X,[H|T],[H|Z]):-
    replace_lesser(X,T,Z).

goal

replace_lesser(s(c,2),[s(a,3),s(c,6),s(d,8)],X).%X=[s("a",3),s("c",2),s("d",8)]
```

Example7:

```
domains
st=s(symbol,integer)
l=st*
predicates
difference(l,l,l)
member(st,l)
clauses
difference([],_,[]):-!.
difference([H|T],Z,[H|T1]):-
                not(member(H,Z)),!,
                difference(T,Z,T1).

difference([_|T],X,Y):-
  difference(T,X,Y).

member(s(A,_),[s(A,_)|_]):-!.
member(X,[_|T]):-
      member(X,T).

goal
 difference([s(k,3),s(a,0),s(f,2),s(b,6)],[s(d,10),s(b,8),s(a,5),s(g,9)],X).
 % X=[s("k",3),s("f",2)]
```

Example8:

```
Domains
state=s(symbol,integer).
l=state*
l1=symbol*
l2=integer*
Predicates
sort(l,l).
min(l,integer,integer).
delet(l,integer,l).
member(state,l).
Clauses
sort([],[]):-!.
sort(L,[s(A,M)|T]):-min(L,100,M),member(s(A,M),L),
          delet(L,M,Z),sort(Z,T),!.
min([],M,M):-!.
min([s(_,V)|T],X,M):-V<X,min(T,V,M),!.
min([_|T],X,M):-min(T,X,M),!.

delet([s(_,X)|T],X,T):-!.
delet([H|T],X,[H|T1]):-delet(T,X,T1).
```

```
member(s(A,X),[s(A,X)|_]):-!.
member(X,[_|T]):-
      member(X,T).

goal sort([s("f",8),s("g",4),s("b",6),s(c,4)],X).%X=[s("g",4),s("c",4),s("b",6),s("f",8)]
```

HW: Try to use the compound object concept to solve any other programs or from that you have already taken it in the previous lecture.

6. Files

Example1:

```
domains
   file=m
   s=string
predicates
   readfile(s)
   writefile(s)
   start
clauses
start:-
   write("Save any string in your file(D:\\test.pro):"),nl,
   readln(X),
   writefile(X),
   readfile(Y),
   write("The string in your file is:",Y),nl.
writefile(X):-
   openwrite(m,"D:\\test.pro"),
   writedevice(m),
   write(X),
   closefile(m).
readfile(X):-
   openread(m,"D:\\test.pro"),
   readdevice(m),
   readln(X),
   closefile(m).
goal:start.
    %Save any string in your file(D:\test.pro):
    %abcdefg
    %The string in your file is:abcdefg
```

Example2(a): Given a program to obtain a digit and its item from a string as shown below:

```
domains
    s=string
    i=integer
predicates
```

```
    digit(s,i,s)
    change(s,i)
clauses
    digit(S,D,I):-
        fronttoken(S,W,R),change(W,D),
        fronttoken(R,I,_).
    digit(S,D,I):-
       fronttoken(S,_,R),digit(R,D,I).
    change("five",5).
    change("ten",10).
Goal:
    digit("I have five pens and ten books.", Digit, Item).
 %Digit =5, Item=pens
 %Digit =10, Item=books
```

Example2(b): Preform the above problem using file concept.

Solution:

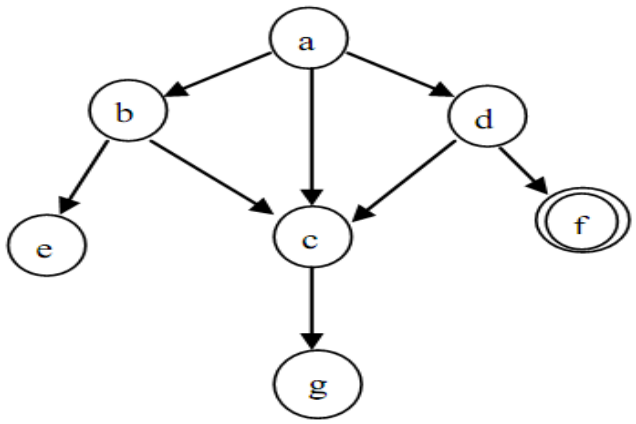
```
domains
    file = myfile
    s=string
    i=integer
predicates
    digit(s,i,s)
    change(s,i)
    writefile
    readfile(s)
    chk_digit(i,s)
clauses
writefile:-
   openwrite(myfile,"d:\\f1.pro"),
   write("Enter sentence:"),nl,
   readln(S),
   writedevice(myfile),
   write(S),
   closefile(myfile).

readfile(S):-
    openread(myfile,"d:\\f1.pro"),
    readdevice(myfile),
    readln(S),
    closefile(myfile).

chk_digit(D,I):-
    readfile(S),
    write("Source sentence is:\n",S),nl,
    write("The result is:\n"),
```

```
    digit(S,D,I).

    digit(S,D,I):-
        fronttoken(S,W,R),change(W,D),
        fronttoken(R,I,_).
    digit(S,D,I):-
       fronttoken(S,_,R),digit(R,D,I).
    change("five",5).
    change("ten",10).
goal
     writefile,chk_digit(Digit,Item).
/*Enter sentence:
I have five pens and ten books.
Source sentence is:
I have five pens and ten books.
The result is:
Digit=5, Item=pens
Digit=10, Item=books
2 Solutions*/
```

Appendix-B

Artificial Intelligence Programs:

Depth first search program

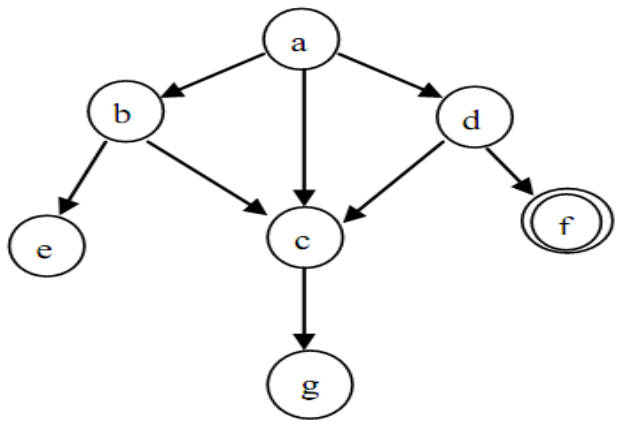



```
Goal:
        depth(['a'],[],'f').

Open=['a']          Closed=[]
Open=['b','c','d']  Closed=['a']
Open=['e','c','d']  Closed=['a','b']
Open=['c','d']      Closed=['a','b','e']
Open=['g','d']      Closed=['a','b','e','c']
Open=['d']          Closed=['a','b','e','c','g']
Open=['f']          Closed=['a','b','e','c','g','d']
      Goal is found
```

```
domains
    c=char .l=c*.
predicates
   path(c,c).
   depth(l,l,c).
   difference(l,l,l).
   append(l,l,l).
   member(c,l).
   print(l,l).
clauses
   path('a','b').
   path('a','c').
   path('a','d').
   path('b','e').
   path('b','c').
   path('d','c').
   path('d','f').
   path('c','g').

   depth([],_,_):-!,write("Goal is not found ").
   depth([G|T_Open],Closed,G):-!,
     print([G|T_Open],Closed),write("Goal is found "),nl.
   depth([H|T_Open],Closed,G):-
     print([H|T_Open],Closed), %print Open & Closed.
     findall(X,path(H,X),Children), %find children of H.
     append(Closed,[H],Closed1), %Put H in Closed.
     difference(Children,T_Open,Children1), %Ignore children of H if already on Open or
     difference(Children1,Closed1,Children2), %on closed
     append(Children2,T_Open,Open1), %Put remaining children on left of Open.
     depth(Open1,Closed1,G).

   print(Open,Closed):- write("Open=",Open,"      ","Closed=",Closed),nl.

   difference([],_,[]):- !.
   difference([H|T],Z,[H|T1]):- not(member(H,Z)),!, difference(T,Z,T1).
   difference([_|T],Z,T1):- difference(T,Z,T1).
```

```
member(H,[H|_]):-!.
member(H,[_|T]):- member(H,T).

append([],L,L):-!.
append([H|T],L,[H|M]):- append(T,L,M).
```

Breadth first search program



<Paths for this tree>

```
path('a','b').
path('a','c').
path('a','d').
path('b','e').
path('b','c').
path('d','c').
path('d','f').
path('c','g').
```

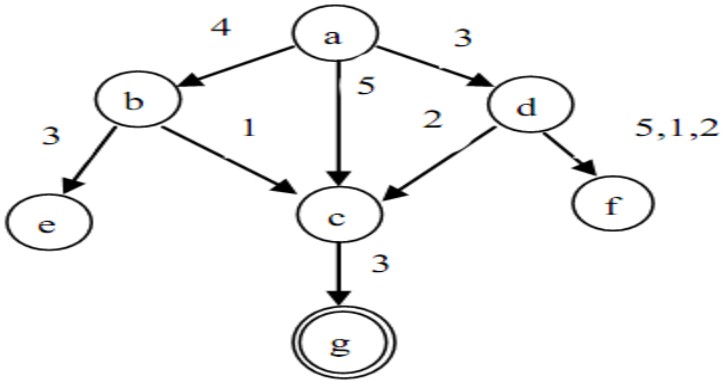
<Goal & Outputs for this tree>

```
goal:breadth(['a'],[],'f').
  Open=['a']            Closed=[]
  Open=['b','c','d']     Closed=['a']
  Open=['c','d','e']     Closed=['a','b']
  Open=['d','e','g']     Closed=['a','b','c']
  Open=['e','g','f']     Closed=['a','b','c','d']
  Open=['g','f']        Closed=['a','b','c','d','e']
  Open=['f']           Closed=['a','b','c','d','e','g']
  Goal is found
```

<Some necessary rules>

```
clauses
 breadth([],_,_):-!,write("Goal is not found ").
 breadth([G|T_Open],Closed,G):-!,
   print([G|T_Open],Closed),write("Goal is found "),nl.
 breadth([H|T_Open],Closed,G):-print([H|T_Open],Closed),
   findall(X,path(H,X),Children), append(Closed,[H],Closed1),
   difference(Children,T_Open,Children1),difference(Children1,Closed1,Children2),
   append(T_Open,Children2,Open1),%Put remaining children on rigth of Open.
   breadth(Open1,Closed1,G).
```

Hill climbing program



Goal Part

```
goal: hill([s('a',0)],[],'g').          %path(s('d',3),s('f',5))
Open=[s('a',0)]      Closed=[]
Open=[s('d',3),s('b',4),s('c',5)]      Closed=[s('a',0)]
Open=[s('c',2),s('f',5),s('b',4),s('c',5)] Closed=[s('a',0),s('d',3)]
Open=[s('g',3),s('f',5),s('b',4),s('c',5)]Closed=[s('a',0),s('d',3),s('c',2)]
Goal is found & the resulted path is[s('a',0),s('d',3),s('c',2),s('g',3)]
  Total cost=8
```

```
goal: hill([s('a',0)],[],'g').          %path(s('d',3),s('f',1)).
  Open=[s('a',0)]      Closed=[]
  Open=[s('d',3),s('b',4),s('c',5)]      Closed=[s('a',0)]
  Open=[s('f',1),s('c',2),s('b',4),s('c',5)]   Closed=[s('a',0),s('d',3)]
  Search is stopped because there is dead end= s('f',1)
```

```
goal: hill([s('a',0)],[],'g').          %path(s('d',3),s('f',2)).
Open=[s('a',0)]      Closed=[]
Open=[s('d',3),s('b',4),s('c',5)]      Closed=[s('a',0)]
Search is stopped because there are equal costs for the states= s('c',2)&s('f',2)
```

Domains Part

```
domains
   f=s(char,integer).
   l=f*.
   c=char.
   i=integer.
```

Predicates Part

```
predicates
   hill(l,l,c).
   append(l,l,l).
   sort(l,l).
   sum(l,i).
   min(l,f).
   del(f,l,l).
   print(l,l).
   dead_end(l,f).
   equal_cost(l).
   path(f,f).
```

Clauses Part<Facts>

```
clauses
  path(s('a',0),s('b',4)).
  path(s('a',0),s('c',5)).
  path(s('a',0),s('d',3)).
  path(s('b',4),s('e',3)).
  path(s('b',4),s('c',1)).
  path(s('d',3),s('c',2)).
  path(s('d',3),s('f',5)).  %path(s('d',3),s('f',1)).path(s('d',3),s('f',2)).
  path(s('c',1),s('g',3)).
  path(s('c',5),s('g',3)).
  path(s('c',2),s('g',3)).
```

Clauses Part<Rules>

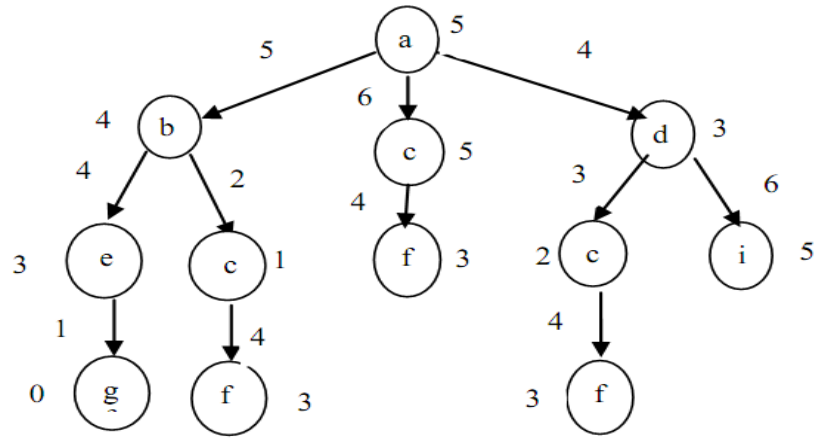
```
hill([],_,_):-!,write("The goal is not found").
hill([s(G,H)|T_Open],Closed,G):-!,
  print([s(G,H)|T_Open],Closed),
  append(Closed,[s(G,H)],Path),
  write("Goal is found & the resulted path is  ",Path),nl,
  sum(Path,N),write("Total cost=",N).
hill([H|T_Open],Closed,G):-
  print([H|T_Open],Closed),
  findall(X,path(H,X),Children),not(dead_end(Children,H)),
  append(Closed,[H],Closed1),
  sort(Children,S_children),not(equal_cost(S_children)),
  append(S_children,T_Open,Open1),
  hill(Open1,Closed1,G).
```

```
dead_end([],H):-
  write("Serch is stopped because there is dead end= ",H),nl.

equal_cost([s(A,X),s(B,X)|_]):-S1=s(A,X),S2=s(B,X),
  write("Serch is stopped because there are equal costs for
                                    the states= ",S1,"&",S2),nl.

print(Open,Closed):-
  write("Open=",Open,"      ","Closed=",Closed),nl.
```

Other necessary rules

```
append([],L,L):-!.
append([H|T],L,[H|T1]):-
    append(T,L,T1).

sum([],0).
sum([s(_,H)|T],N):-
    sum(T,N1),N=N1+H.

sort([],[]):-!.
sort(L,[M|T]):-
    min(L,M),
    del(M,L,X),
    sort(X,T).

min([M],M):-!.
min([s(A,X),s(_,Y)|T],M):-
    X<=Y,!,
    min([s(A,X)|T],M).
min([_|T],M):-
    min(T,M).

del(X,[X|T],T):-!.
del(X,[H|T],[H|T1]):-
    del(X,T,T1).
```

Best First Search Program



<Goal of this tree>

```
goal:   best([s('a',5)],[],'g').
Open=[s('a',5)]                          Closed=[]
Open=[s('d',3),s('b',4),s('c',5)]     Closed=[s('a',5)]
Open=[s('c',2),s('b',4),s('i',5)]     Closed=[s('a',5),s('d',3)]
Open=[s('f',3),s('b',4),s('i',5)]     Closed=[s('a',5),s('d',3),s('c',2)]
Open=[s('b',4),s('i',5)]              Closed=[s('a',5),s('d',3),s('c',2),s('f',3)]
Open=[s('c',1),s('e',3),s('i',5)]     Closed=[s('a',5),s('d',3),s('f',3),s('b',4)]
Open=[s('e',3),s('i',5)]     Closed=[s('a',5),s('d',3),s('f',3),s('b',4),s('c',1)]
Open=[s('g',0),s('i',5)]     Closed=[s('a',5),s('d',3),s('f',3),s('b',4),s('c',1),s('e',3)]
The goal is found &The resulted path is
[s('a',0),s('d',4),s('f',4),s('b',5),s('c',2),s('e',4),s('g',1)]
Total cost=20
```

<Paths of the given tree>

```
path(s('a',5),s('b',4)).
path(s('a',5),s('c',5)).
path(s('a',5),s('d',3)).
path(s('b',4),s('e',3)).
path(s('b',4),s('c',1)).
path(s('e',3),s('g',0)).
path(s('c',1),s('f',3)).
path(s('c',5),s('f',3)).
path(s('d',3),s('c',2)).
path(s('d',3),s('i',5)).
path(s('c',2),s('f',3)).
```

< Necessary clauses for Best First Search Program>

```
clauses
  best([],_,_):-!,write("The goal is not found").
  best([s(G,H)|T],Closed,G):-!,
        print([s(G,H)|T],Closed),
        append(Closed,[s(G,H)],Path),
        original_cost(Path,Path1),
        write("The goal is found &The resulted path is  ",Path1),nl,
        sum(Path1,N),write("Total cost=",N),nl.
  best([H|T_Open],Closed,G):-
        print([H|T_Open],Closed),
        findall(X,path(H,X),Children),
        append(Closed,[H],Closed1),
        check(Children,T_Open,Closed1,Open1,Closed2),
        sort(Open1,Open2),
        best(Open2,Closed2,G).

   check(Children,Open,Closed,New_Open,Closed):-
        difference(Children,Open,X),
```

```
    difference(X,Closed,Y),
    Children=Y,!,%Chidren aren't in Open or in the Closed.
    append(Children,Open,New_Open).%Add children to the Open.
check(Children,Open,Closed,New_Open,Closed):-
    difference(Children,Open,X),not(Children=X),!,%There is a Child or more in the Open.
    best_open(Children,Open,Open1),%Make the Open is the best by replace the state by the best.
    append(X,Open1,New_Open).%Add dissimilar child to the Open.
check(Children,Open,Closed,New_Open,New_closed):-%There is Child or more in the Closed.
    best_closed(Children,Closed,New_closed),%Make the Closed is the best by delete the worst.
    best_children(Closed,Children,Best_child),%Make the Children is the best by ignore the not best
    append(Best_child,Open,New_Open).%Add the pure children to the Open.

best_open([],Z,Z):-!.
best_open([X|T],Y,Z):-
    set_best(X,Y,Z1),
    best_open(T,Z1,Z).
set_best(_,[],[]):-!.
set_best(s(A,X),[s(A,Y)|T],[s(A,X)|T]):-
    X<Y,!.
  set_best(X,[H|T],[H|Z]):-
    set_best(X,T,Z).

best_closed([],Z,Z):-!.
best_closed([X|T],Y,Z):-
    remove_worst(X,Y,Z1),
    best_closed(T,Z1,Z).
remove_worst(_,[],[]):-!.
remove_worst(s(A,X),[s(A,Y)|T],T):-
    Y>X,!.
remove_worst(X,[H|T],[H|Z]):-
    remove_worst(X,T,Z).

best_children([],Z,Z):-!.
```

```
best_children([X|T],Y,Z):-
    ignore_worst(X,Y,Z1),
    best_children(T,Z1,Z).
ignore_worst(_,[],[]):-!.
ignore_worst(s(A,X),[s(A,Y)|T],T):-
    Y>=X,!.
ignore_worst(X,[H|T],[H|Z]):-
ignore_worst(X,T,Z).
```