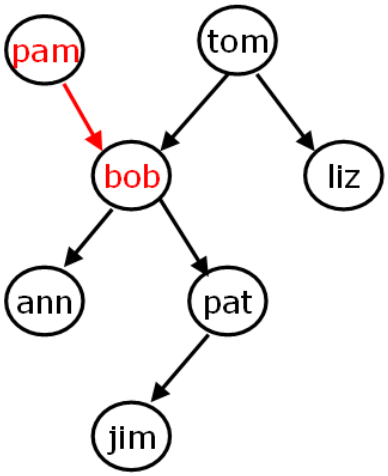
Prolog Programming for AI

Introduction to Prolog

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

It is a declarative programming language. i.e. we specify *what* the situation (*rules and facts*) and the goal (*query*) are and let the Prolog interpreter derive the solution for us.

Example: given the family tree.



Facts

```
parent( pam, bob).      % Pam is a parent of Bob
parent( tom, bob).
parent( tom, liz).
parent( bob, ann).
parent( bob, pat).
parent( pat, jim).
```

Note: % is comment. /* this is also comment */
Questions:
Is Bob a parent of Pat?

```
?- parent( bob, pat).
```

Is pam a parent?

```
?- parent( pam, _).
```

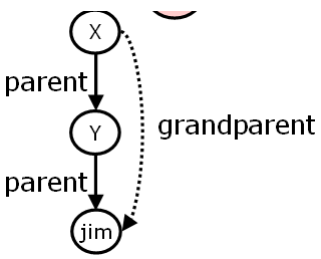
Who is Liz's parent?

```
?- parent( X, liz).
```

Who are Bob's children?

```
?- parent( bob, X).
```

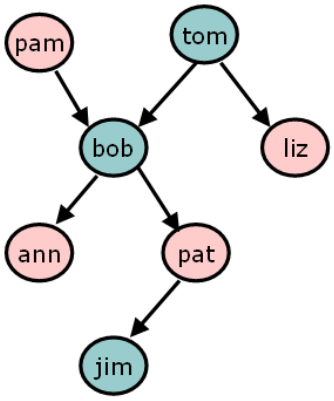
Who is a parent of whom?



Find X and Y such that X is a parent of Y.
?- parent(X, Y).

Who is a *grandparent* of Jim?
?- parent(Y, jim), parent(X, Y).
Who are Tom's grandchildren?
?- parent(tom, X), parent(X, Y).

Do Ann and Pat have a common parent?
?- parent(X, ann), parent(X, pat).

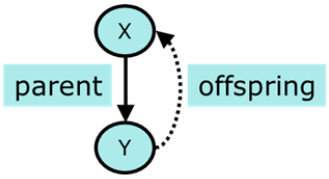



```
female( pam).    % Pam is female
female( liz).
female( ann).
female( pat).
male( tom).      % Tom is male
male( bob).
male( jim).
```

Rules

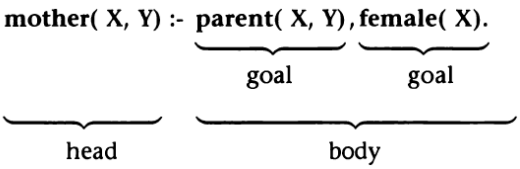
- Define the "offspring" relation:

offspring(Y, X) :- parent(X, Y).



For all X and Y,
Y is an offspring of X if
X is a parent of Y.

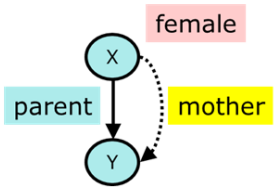
Rules have:
- A *condition* part (body): the right-hand side of the rule
- A *conclusion* part (head): the left-hand side of the rule



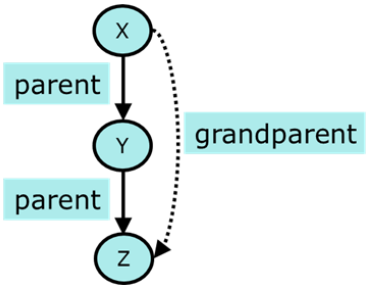
?- offspring(liz, tom).
?- offspring(X, Y).

- Define the "mother" relation:
  - **mother(X, Y) :- parent(X, Y), female(X).**
  - For all X and Y,

  X is the mother of Y if
  X is a parent of Y and
  X is a female.



- Define the "grandparent" relation:
  - **grandparent(X, Z) :-**
    parent(X, Y), parent(Y, Z).



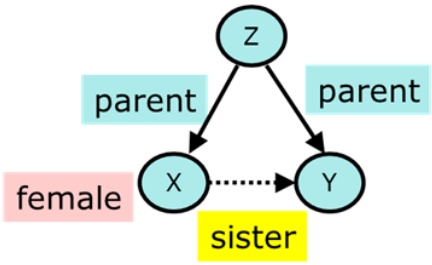
- Define the "sister" relation:

- **sister(X, Y) :-**
  parent(Z, X), parent(Z, Y), female(X).
- For any X and Y,

X is a sister of Y if
(1) both X and Y have the same parent, and
(2) X is female.

- ?- sister(ann, pat).
- ?- sister(X, pat).
- ?- sister(pat, pat).

- Pat is a sister to herself?!



- To correct the "sister" relation:
  - **sister(X, Y) :-**
    parent(Z, X), parent(Z, Y), female(X), X=\ Y.

- Prolog program consists of clauses.
- Prolog *clauses* consist of
  - Head
  - Body: a list of goal separated by commas (,)

- Prolog clauses are of three types:
  - *Facts*:
    - declare things that are always true.
    - facts are clauses that have a head and the empty body.
  - *Rules*:
    - declare things that are true depending on a given condition
    - rules have the head and the (non-empty) body
  - *Questions*:
    - the user can ask the program what things are true
    - questions only have the body

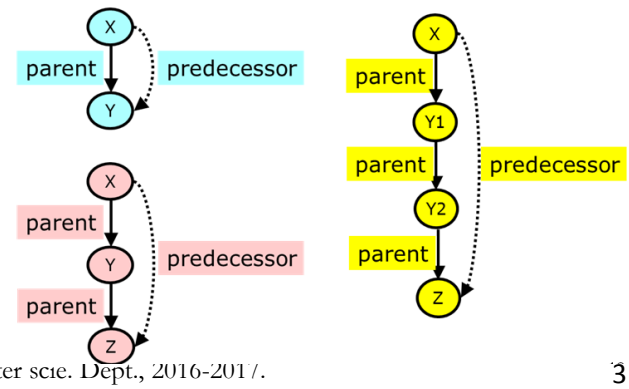
- A *variable* can be substituted by another object.

- Variables are assumed to be universally quantified and are read as "for all".
  - For example:

hasachild(X) :- parent(X, Y).
can be read in two way
(a) For all X and Y,
if X is a parent of Y then X has a child.
(b) For all X,
X has a child if there is some Y such that X is a parent of Y.

Define the "predecessor" relation

predecessor(X, Z):- parent(X, Z).
predecessor(X, Z):-
parent(X, Y), predecessor(Y, Z).



?- predecessor(pam, X).

Recursive rules
The two "predecessor relation" clauses.

```
predecessor( X, Z)  :- parent( X, Z).
predecessor( X, Z)  :- parent( X, Y), predecessor( Y, Z).
```

Such a set of clauses is called a **procedure**.

Trace and Notrace
to trace a query:
?- trace.

Example: **predecessor(X, Z).**

To exit the trace mode:
?- notrace.

'Not' syntax in Prolog

We use the preidicate /+ to say 'not'
Example:
Mary likes animals but not the snake.

```
animal(beer).
animal(cat).
animal(dog).
animal(snake).
snake(snake).
snake(python).
like(X):-animal(X), \+snake(X).
```

```
?- like(X).
X = beer ;
X = cat ;
X = dog ;
false.
```

Run Prolog program

1- write your code in SWI editor, save it as family.pl
2- to run the program, use ?-[family]. Or ?- consult('family.pl')
3- Answer the program for example: ?- female(pat).

Prolog Syntax

Prolog language include terms:

1- Atoms: strings of letters, digits and the underscore character,'_', starting with a lower case letter like: elephant, b, abcXYZ, x_123, 'This is also a Prolog atom.' , + - * = < > : &.
2- Numbers: integers like 10, -5 and real numbers like 3.14.
3- Variables: start with capital letters or underscore like: X, X_variable, _x.
    - If the name X occurs in two clauses, then it signifies two different variables.

Hasachild(X) :- parent(X, Y).
isapoint(X) :- point(X, Y, Z).

- But each occurrence of X with in the same clause means the same variables.

Hasachild(X) :- parent(X, Y).

Matching
The terms is_bigger(X, dog) and is_bigger(elephant, dog) match, because the variable X can be instantiated with the atom elephant.
?- is_bigger(X, dog) = is_bigger(elephant, dog).
X = elephant
Yes

?- p(_, 2, 2) = p(1, Y, _).
Y = 2
Yes

How Prolog Serve Queries
Given the goal list G1,…,Gm.

Scan through the clauses in the program from top to bottom until the first clause, C, is found such that the head of C matches the first goal G1.

- If there is no such clause then terminate with failure.
- Match G1 and the head of C. Replace G1 with the body of C (except the facts) to obtain a new goal list.

Execute this new goal list.

Backtracking
If we reach a point where a goal can't be matched, or the body of a rule can't be matched, we *backtrack* to the last (most recent) clause where a choice of matching a particular fact or rule was made. We then try to match a different fact or rule. If this fails we go back to the next previous place where a choice was made and try a different match there. We try alternatives until we are able to solve all the goals in our query or until all possible choices have been tried and found to fail. If this happens, we answer "no" the query can't be solved. As we try to match facts and rules we try them in their order of definition.

Example:
Program:

```
big( bear).
Big( elephant).
Small( cat).
Brown( bear).
Black( cat).
Gray( elephant).
Dark(Z):- black(Z).
dark(Z):- brown(Z).
```

How to prevent Backtracking
To prevent the backtracking, we use the cut '!' predicate.
Example:

```
f( X, normal)  :-  X < 3, !.
f( X, alert1)  :-  3 =< X, X < 6, !.
f( X, alert2)  :-  6 =< X.
```

We use ! in the mutually exclusive rules:

Exampl1

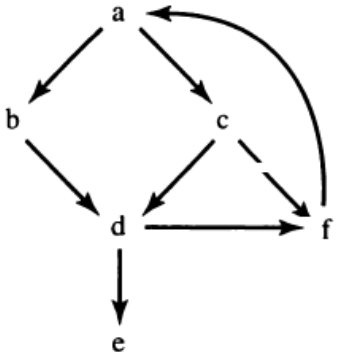
```
max( X, Y, X)   :-   X >= Y, !.

max( X, Y, Y).
```

Exampl2

```
beat( tom, jim).
beat( ann, tom).
beat( pat, jim).
class( X, fighter) :-
  beat( X, _ ),
  beat( _, X), !.

class( X, winner) :-
  beat( X, _ ), !.

class( X, sportsman) :-
  beat( _, X).
```

Assignment-1

1- Suppose you have the following directed graph:




```
link( a, b).    link( a, c).
link( b, d).    link( c, d).
link( c, f).    link( d, e).
link( d, f).    link( f, a).
```

Define the rule **path(X,Y)** to find the path from point **X** and point **Y**. *Hint*: use recursion. Then try to answer:
?path(a,d).

2- for the family problem: define the predicate **cousin(X,Y), and ant(X,Y), wife(X,Y), husband(X,Y), uncle(X,Y)**.
3- Define the predicate **happy(X)**, such that if **X** has a child then he is happy.
4- "***Map coloring problem***". Given number of countries on the map, we want to color them such that no two neighbors have the same color. Please write the following code to do that job.

```
% Map colouring

% Possible pairs of colours of neighbour countries

n( red, green).     n( red, blue).      n( red, yellow).
n( green, red).     n( green, blue).    n( green, yellow).
n( blue, red).      n( blue, green).    n( blue, yellow).
n( yellow, red).    n( yellow, green).  n( yellow, blue).

% Part of Europe (IT = Italy, SI = Slovenia, HR = Croatia,  CH = Switzerland, ...)

colours( IT, SI, HR, CH, AT, HU, DE, SK, CZ, PL, SEA) :-
    SEA = blue,                                  % Adriatic Sea has to be coloured blue
    n( IT, CH), n( IT, AT), n( IT, SI), n( IT, SEA),  % Italy and Switzerland are neighbours, etc.
    n( SI, AT), n( SI, HR), n( SI, HU), n( SI, SEA),
    n( HR, HU), n( HR, SEA),
    n( AT, CH), n( AT, DE), n( AT, HU), n( AT, SK), n( AT, CZ),
    n( CH, DE),
    n( HU, SK),
    n( DE, SK), n( DE, CZ), n( DE, PL),
    n( SK, CZ), n( SK, PL),
    n( CZ, PL).
```

?- colours(IT, SI, HR, CH, AT, HU, DE, SK, CZ, PL, SEA).

List Manipulation

A list is a sequence of any number of items.
For example:
[ann, tennis, tom, skiing]
A list is either empty or non-empty.
Empty: []
Non-empty:
- The first term, called the *head* of the list
- The remaining part of the list, called the *tail*

Example:
?[Head|Tail] = [ann, tennis, tom, skiing].
Head= ann
Tail=[tennis, tom, skiing]

?List=[a,1,b,[tom, bob],3].

Membership

The membership relation:
member(X, L)
where X is an object and L is list.
The goal **member(X, L)** is true if X occurs in L.

For example:
member(b, [a, b, c]) is true
member(b, [a, [b, c]]) is not true
member([b, c] , [a, [b, c]]) is true

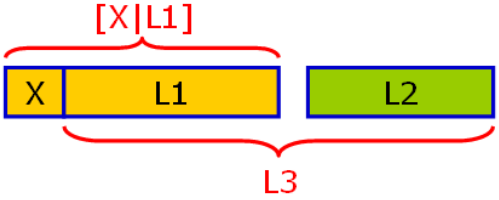
we can defined member relation as:
member1(X, [X| _]).
member1(X, [_| Tail]) :- member1(X, Tail).

Concatenation

The concatenation relation:
conc(L1, L2, L3)
here L1 and L2 are two lists, and L3 is their concatenation.

For example:
conc([a, b], [c, d], [a, b, c, d]) is true
conc([a, b], [c, d], [a, b, a, c, d]) is not true
conc relation is defined as:

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



?- conc([a, b], [c, d], A).

?- conc(L1, L2, [a,b,c]).

```
?- conc( Before, [may| After], [jan, feb, mar, apr, may, jun, jul, aug, sep, oct, nov, dec]).
After = [jun,jul,aug,sep,oct,nov,dec]
Before = [jan,feb,mar,apr] ? ;
false

 ?- conc( _, [Month1,may, Month2|_], [jan, feb, mar, apr, may, jun, jul, aug, sep, oct, nov, dec]).
Month1 = apr
Month2 = jun ? ;
false

 ?- L1 = [a,b,z,z,c,z,z,z,d,e], conc(L2,[z,z,z|_ ], L1).
L1 = [a,b,z,z,c,z,z,z,d,e]
L2 = [a,b,z,z,c] ? ;
false
```

Adding an item
Add an item at the beginning of a list.

add(X, L,[X|L]).
?- add(4, [1,2,3],Y).

Deleting an item
Deleting an item X form a list L can be programmed as:

del(X, [X| Tail], Tail).
del(X, [Y| Tail], [Y|Tail1]) :- del(X, Tail, Tail1).
?- del(4, [1,2,3,4,5,6],Y).
?- del(a,[a,b,a,a], L).
?- del(a, L, [1,2,3]).

Two applications of del:

1- Inserting X at any place in some list List giving BiggerList can be defined:

```
insert( X, List, BiggerList) :-
        del( X, BiggerList, List).
```

2- Use del to test for membership:

```
member3( X, List) :- del( X, List, _).
```

Sublist
The sublist relation has two arguments, a list L and a list S such that S occurs within L as its sublist.

For example:
sublist([c, d, e], [a, b, c, d, e]) is true
sublist([c, e], [a, b, c, d, e, f]) is not true

sublist(S, L) :- conc(L1, L2, L), conc(S, L3, L2).

?- sublist(S, [a,b,c]).

Permutations
Permutation examples:

```
| ?- permutation( [a, b, c], P).

P = [a,b,c] ? ;
P = [a,c,b] ? ;
P = [b,a,c] ? ;
P = [b,c,a] ? ;
P = [c,a,b] ? ;
P = [c,b,a] ? ;
```

Assignment-2

1- Write a goal, using **conc**, to delete the last three elements from a list L producing another list L1.

2- Write a goal to delete the first three elements and the last three elements from a list L producing list L2.

3- Define the relation **last1(Item, List)** so that **Item** is the last element of a list **List**.
Write two versions:
- Using the **conc** relation
- Without **conc**

4- Redefine the concatenation relation: **conc1(L1, L2, L3)** here L1 and L2 are two lists, L3 is their concatenation, and L2 is put before L1.

5- Define a relation **add_end(X, L, L1)** to add an item X to the end of list L.
For example,

```
? add_end(a, [b,c,d], L1).
L1 = [b, c, d, a].
```

6- Define a relation **del_all(X, L, L1)** to remove all items X (if any) from list L.

```
? del_all(a, [a,b,a,c,d,a], L1).
L1 = [b, c, d].
```

7- Define the relation **reverse1(List, ReversedList)** that reverses lists. For example,

```
? reverse1([a, b, c, d], L).
L= [d, c, b, a]
```

8- Define the predicate **palindrome(List).** A list is a palindrome if it reads the same in the forward and in the backward direction.

```
palindrome([m,a,d,a,m]).
true.
```

9- Define a predicate **element_at(L, K)** to Find the **K**'th element of a list **L**.

```
?- element_at([a,b,c,d,e],3).
X = c
```

10- Define a predicate **Split(L,L1,L2,N)** to split a list **L** into two parts **L1** and **L2**; the length of the first list is **N**.

?- split([a,b,c,d,e,f,g,h,i,k],L1,L2,3).
L1 = [a,b,c]
L2 = [d,e,f,g,h,i,k]

11- Define a predicate **union(L1,L2,L3)** to find the union of two lists.
12- Define a predicate **intersection(L1,L2,L3)** to find the intersection of two lists.
13- Define a predicate **evenlength(L)** to check whether **L** has even number of items.
14- Define a predicate **maxlist(L,X)** to find the maximum number **X** of the list **L**.

Arithmetic

Predefined basic arithmetic operators:

+	addition
-	subtraction
*	multiplication
/	division
**	power
//	integer division
mod	modulo, the remainder of integer division

?- X **=** 1+2.
X = 1+2
yes
?- X **is** 1+2.
X = 3
yes

?- X is 5/2, Y is 5//2, Z is 5 mod 2.
X = 2.5
Y = 2
Z = 1

Prolog implementations usually also provide standard functions such as sin(X), cos(X), atan(X), log(X), exp(X), etc.

?- X is sin(3).
X = 0.14112000805986721

Predefined comparison operators:

X > Y	X is greater than Y
X < Y	X is less than Y
X >= Y	X is greater than or equal to Y
X =< Y	X is less than or equal to Y
X =:= Y	the values of X and Y are equal
X =\= Y	the values of X and Y are not equal

| ?- 1+2 =:= 2+1.
yes
| ?- 1+2 = 2+1.
no
| ?- 1+A = B+2.
A = 2

B = 1
yes

GCD (greatest common divisor) problem:
gcd(X, X, X).
gcd(X, Y, D) :- X<Y, Y1 is Y-X, gcd(X, Y1, D).
gcd(X, Y, D) :- Y<X, gcd(Y, X, D).

```
?- gcd( 20, 25, D).
D=5
```

Length counting problem:
Define procedure **length1(List, N)** which will count the elements in a list **List** in variable **N.**
length1([], 0).
length1([_| Tail], N) :- length1(Tail, N1), N is 1 + N1.

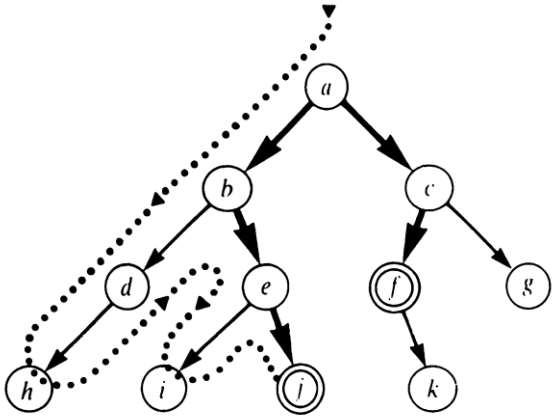
?- length1([a, b, [c, d], e], N).
N = 4

Assignment-3

1- Define the predicate **abs(X,Y)** to find the absolute value of **X**.
2- Define the predicate **cube(X,Y)** to find the cube of **X**. ?cube(2,Y). Y= 8.
3- Define the predicate **sumlist(List, Sum)** so that **Sum** is the sum of a given list of numbers **List**.
4- Define the predicate **ordered(List)** which is true if **List** is an ordered list of numbers. **ordered([1,5,6,6,9,12]).**
5- Define the predicate **max(X,Y,Z)** to find the maximum number between **X** and **Y**.
6- Define the predicate **range(X,Y,List)** to generate the integer numbers between **X** and **Y** and store them in **List**. ? range(3,7,L). L=[3,4,5,6,7]
7- Define the predicate **is_prime(X)** to determine whether **X** is prime.
8- Define a predicate **add_list(L1,L2,L)** to add two lists.
9- Define a predicate **even_odd(L,Odd,Even)** to split **L** into two lists, such that **Odd** includes the odd numbers, and **Even** includes the even numbers.
10- Define a predicate **power(X,N)** to compute **X^N**.
11- Define a predicate **fact(X)** to compute the factorial of **X**.
12- Define a predicate febo(N, X) to compute X=0+1+1+2+3+5+8+...+N

Deep_first search

Consider the following graph



The following program, shows how to find the goal by using deepth_first search.

```
s(a,b). s(b,d). s(b,e). s(a,c). s(c,f). s(c,g). s(f,k).
s(d,h). s(e,i). s(e,j).
goal(j).
goal(f).
solve(N,[N]):-goal(N).
solve(N,[N|Sol1]):-s(N,N1),solve(N1,Sol1).
```

?- solve(a,Sol).

Deepth_limited search

```
s(a,b). s(b,d). s(b,e). s(a,c). s(c,f).s(c,g). s(f,k). s(d,h). s(e,i). s(e,j).
goal(j). goal(f).
solve(N,[N],_):-goal(N).
solve(N,[N,Sol], Maxdepth):-Maxdepth>0,s(N,N1), Max1 is Maxdepth-1, solve(N1,Sol,Max1).
```

Iterative deepening depth-first search

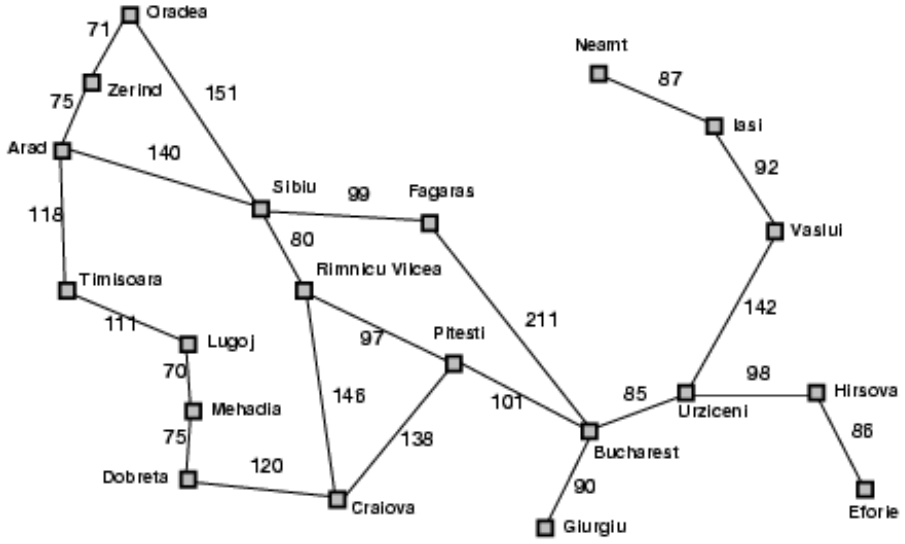
```
s(a,b). s(b,d). s(b,e). s(a,c). s(c,f). s(c,g). s(f,k). s(d,h). s(e,i). s(e,j). goal(j). goal(f).
  exactdeep( F, F , D , [F] )  :-  D = 0 .
  exactdeep( F, T , D , [F|P] ) :- D1 is D-1 ,
                                   s( F, U ) , exactdeep( U, T, D1 , P ) .
  iter( F, T , Maxdepth , P ) :- for( D, 1 , Maxdepth ) ,
                                 exactdeep( F, T, D, P ) .
  for( Low , Low, High ) :- Low =< High .
  for( I , Low , High ) :- Low1 is Low+1 ,
                           Low1 =< High ,
                           for( I , Low1, High ).
  go1(I) :- for( I, 1, 3 ) .
  go2(Piter) :- iter( a, _, 4 , Piter ) .
```

Breadth_first search

```
s(a,b). s(b,d). s(b,e). s(a,c). s(c,f). s(c,g). s(f,k). s(d,h). s(e,i). s(e,j). goal(j). goal(f).
solve(Start, Solution):-
        breadthfirst([[Start]|Z]-Z, Solution).
breadthfirst([[Node|Path]|_]-_,[Node|Path]):-goal(Node).
breadthfirst([Path|Paths]-Z, Solution):-
        extend(Path, Newpaths),
        conc(Newpaths, Z1, Z),
Paths\==Z1, breadthfirst(Paths-Z1,Solution).

extend([Node|Path], Newpaths):-
        findall([Newnode, Node|Path],
                (s(Node, Newnode),\+ member(newnode, [Node|Path])),
                Newpaths).
conc([],L,L).
conc([X|L1],L2,[X|L3]):-conc(L1,L2,L3).
```

A* Algorithm

The following program shows how the A* algorithm can find the best route for the following cities. The goal city is *Bucharest*.



Straight–line distance to Bucharest	
Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	176
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	10
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

```
s(oradea,zerind,71).
s(oradea,sibiu,151).
s(zerind,arad,75).
s(arad,timisoara,118).
s(arad,sibiu,140).
s(timisoara,lugoj,111).
s(lugoj,mehadia,70).
s(mehadia,dobreta,75).
s(dobreta,craiova,120).
s(craiova,pitesti,138).
s(sibiu,fagaras,99).
s(sibiu,rimnicu,80).
s(rimnicu,pitesti,97).
s(rimnicu,craiova,146).
s(pitesti,bucharest,101).
s(fagaras,bucharest,211).
s(giurgiu,bucharest,90).
s(urziceni,bucharest,85).
s(neamt,iasi,87).
s(iasi,vaslui,92).
s(vaslui,urziceni,142).
s(hirsova,urziceni,98).
s(efoie,hirsova,86).

goal(bucharest).
path(X,Y):-s(X,Y,_).
path(X,Y):-s(X,Z,_),path(Z,Y).

h(oradea,380).
```

```
h(zerind,374).
h(arad,366).
h(sibiu,253).
h(timisoara,329).
h(lugoj,244).
h(mehadia,241).
h(dobreta,242).
h(craiova,160).
h(fagaras,176).
h(rimnicu,193).
h(pitesti,10).
h(bucharest,0).
h(giurgiu,77).
h(urziceni,80).
h(neamt,234).
h(iasi,226).
h(vaslui,199).
h(hirsova,151).
h(eforie,161).


bestfirst(Start,Solution):-
        expand([],l(Start,0/0),9999,_,yes,Solution).

expand(P,l(N,_),_,_,yes,[N|P]):-
        goal(N).

expand(P,l(N,F/G),Bound,Tree1,Solved,Sol):-
        F=<Bound,
        (bagof(M/C,(s(N,M,C),\+member(M,P)),Succ),
        !,
        succlist(G,Succ,Ts),
        bestf(Ts,F1),
        expand(P,t(N,F1/G,Ts),Bound,Tree1,Solved,Sol)
        ;
        Solved=never
        ).


expand(P,t(N,F/G,[T|Ts]),Bound,Tree1,Solved,Sol):-
        F=<Bound,
        bestf(Ts,Bf),min(Bound,Bf,Bound1),
      expand([N|P],T,Bound1,T1,Solved1,Sol),
        continue(P,t(N,F/G,[T1|Ts]),Bound,Tree1,Solved1,Solved,Sol).

expand(_,t(_,_,[]),_,_,never,_):-!.
expand(_,Tree,Bound,Tree,no,_):-
        f(Tree,F),F>Bound.


continue(_,_,_,_,yes,yes,_).
```

```
continue(P,t(N,_/G,[T1|Ts]),Bound,Tree1,no, Solved,Sol):-
        insert(T1,Ts,Nts),
        bestf(Nts,F1),
        expand(P,t(N,F1/G,Nts),Bound,Tree1,Solved,Sol).

continue(P,t(N,_/G,[_|Ts]),Bound,Tree1,never,Solved,Sol):-
        bestf(Ts,F1),
        expand(P,t(N,F1/G,Ts),Bound,Tree1,Solved,Sol).

succlist(_,[],[]).
succlist(G0,[N/C|Ncs],Ts):-
        G is G0+C,
        h(N,H),
        F is G+H,
        succlist(G0,Ncs,Ts1),
        insert(l(N,F/G),Ts1,Ts).
insert(T,Ts,[T|Ts]):-
        f(T,F),bestf(Ts,F1),
        F=<F1,!.
insert(T,[T1|Ts],[T1|Ts1]):-
        insert(T,Ts,Ts1).
f(l(_,F/_),F).
f(t(_,F/_,_),F).

bestf([T|_],F):-
        f(T,F).
bestf([],9999).

min(X,Y,X):-X=<Y,!.
min(_,Y,Y).
```

The query like this: ? bestfirst(oradea, Solution).

Assignment-4

1- Apply the A* algorithm to find the best route to "Baghdad". Use Iraq map and google map to get the heuristic distances to the goal.

Constraint Logic Problems (CLP)

Given:
1- Set of *variables*.
2- *Domain* for each variable.
3- *Constraints* that variables have to satisfied.
Find:
An assignments of values to the variables, such that all constraints are satisfied.

We will focus on CLP over finite domains CLP(FD), where the domains are sets of *integers*.

- We should use

? **use_module(library(clbfd)).**

- The domain of variable is written as:
  X in set
  List ins set

Where set is : item1..item2 or set1 \/ set2.
Examples:
? X in 1..10.
X in 1..10.

? X in 1..5 \/ 3..12.
X in 1..12.

? [X,Y,Z] ins 0..3.
X in 0..3,
Y in 0..3,
Z in 0..3.

Constraints have the form:
Exp1 #=, #\=, #<, #>, #=<, #=>, Exp2
Example :
? 2*X #= 10.
X = 5.

? X*X #= 144.
X in -12\/12.

?- 4*X + 2*Y #= 24, X + Y #= 9, [X,Y] ins 0..sup.
X = 3,
Y = 6.

Some useful predicates:
1- indomain(X): assign through backtracking possible values to X.
Example:
? X in 1..3, indomain(X).
X=1;
X=2;
X=3

2- **labeling(Options, Vars):** assign value to each variable in Vars. Options is a list of options to select the variables and their values orders.

the variable selection strategy lets you specify which variable of *Vars* is labeled next and is one of:

leftmost
Label the variables in the order they occur in *Vars*. This is the default.

ff
First fail. Label the leftmost variable with smallest domain next, in order to detect infeasibility early. This is often a good strategy.

ffc
Of the variables with smallest domains, the leftmost one participating in most constraints is labeled next.

min
Label the leftmost variable whose lower bound is the lowest next.

max
Label the leftmost variable whose upper bound is the highest next.

The value order is one of:

up
Try the elements of the chosen variable's domain in ascending order. This is the default.

down
Try the domain elements in descending order.

The branching strategy is one of:

step
For each variable X, a choice is made between X = V and X #\= V, where V is determined by the value ordering options. This is the default.

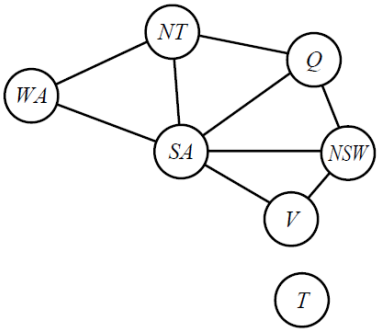
enum
For each variable X, a choice is made between X = V_1, X = V_2 etc., for all values V_i of the domain of X. The order is determined by the value ordering options.

bisect
For each variable X, a choice is made between X #=< M and X #> M, where M is the midpoint of the domain of X.

4- **all_different(Vars)**: all varibals in Vars must have different values.

?- **[X,Y] ins 1..3, labeling([],[X,Y]).**
?- L =[X,Y,Z], L ins 1..3, all_different(L), labeling([],L). (Permutation).

Solution of Map coloring problem by CLP



```
graph([wa,nt,sa,q,nsw,v,t],[[wa,nt],[wa,sa],[nt,sa],[sa,q],[nt,q],[q,nsw],[nsw,v],[v,sa]]).

coloring(K,Output) :- graph(Nodes, Edges),
create_output(Nodes, Colors, Output),Colors ins 1..K,
different(Edges, Output), labeling([ff], Colors).

create_output([],[],[]).
create_output([N|Nodes], [C|Colors], [N/C|Output]) :-
create_output(Nodes, Colors, Output).

different([],_).
different([[A,B]|R], Output) :- member(A/CA, Output),
member(B/CB, Output), CA #\= CB, different(R, Output).
```

?- coloring(K,Output). // K number of color

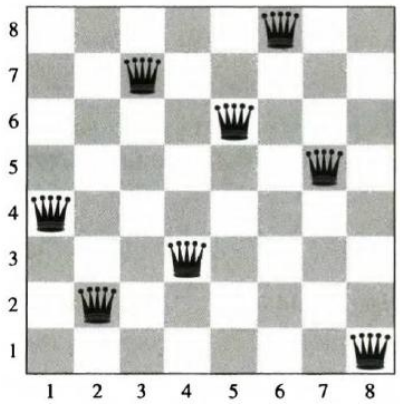
The solution of eight queen problem by CLP

The problem here is to place eight queens on the empty chessboard in such a way that no queen attacks any other queen.

- The solution will be programmed as a unary predicate:

Solution(Pos)

which is true if and only if **Pos** represents a position with eight queens that do not attack each other.




```
solution(Ys):-
        Ys=[_,_,_,_,_,_,_,_],
        Ys ins 1..8,
        all_different(Ys),
        safe(Ys),
        labeling([],Ys).
safe([]).
safe([Y|Ys]):-
        no_attack(Y,Ys,1),
        safe(Ys).
no_attack(_,[],_).
no_attack(Y1,[Y2|Ys],D):-
        D#\=Y1-Y2,
```

```
        D#\=Y2-Y1,
        D1 is D+1,
        no_attack(Y1,Ys,D1).
```

Assignment-5

Use map coloring program to pain the provinces of Iraq.

Tic tac toe Game

We have three programs:

1- minimax.pl

```
:- module(minimax, [minimax/3]).

% minimax(Pos, BestNextPos, Val)
% Pos is a position, Val is its minimax value.
% Best move from Pos leads to position BestNextPos.
minimax(Pos, BestNextPos, Val) :-                    % Pos has successors
    bagof(NextPos, move(Pos, NextPos), NextPosList),
    best(NextPosList, BestNextPos, Val), !.
minimax(Pos, _, Val) :-                          % Pos has no successors
    utility(Pos, Val).

best([Pos], Pos, Val) :-
    minimax(Pos, _, Val), !.

best([Pos1 | PosList], BestPos, BestVal) :-
    minimax(Pos1, _, Val1),
    best(PosList, Pos2, Val2),
    betterOf(Pos1, Val1, Pos2, Val2, BestPos, BestVal).

betterOf(Pos0, Val0, _, Val1, Pos0, Val0) :-   % Pos0 better than Pos1
    min_to_move(Pos0),                         % MIN to move in Pos0
    Val0 > Val1, !                             % MAX prefers the greater value
    ;
    max_to_move(Pos0),                         % MAX to move in Pos0
    Val0 < Val1, !.                            % MIN prefers the lesser value
betterOf(_, _, Pos1, Val1, Pos1, Val1).        % Otherwise Pos1 better than Pos0
```

2- tictactoe.pl

```
:- module(tictactoe,
[move/2,min_to_move/1,max_to_move/1,utility/2,winPos/2,drawPos/2]).
% move(+Pos, -NextPos)
% True if there is a legal (according to rules) move from Pos to NextPos.
move([X1, play, Board], [X2, win, NextBoard]) :-
   nextPlayer(X1, X2),
   move_aux(X1, Board, NextBoard),
   winPos(X1, NextBoard), !.
move([X1, play, Board], [X2, draw, NextBoard]) :-
   nextPlayer(X1, X2),
   move_aux(X1, Board, NextBoard),
   drawPos(X1,NextBoard), !.
move([X1, play, Board], [X2, play, NextBoard]) :-
   nextPlayer(X1, X2),
   move_aux(X1, Board, NextBoard).
% move_aux(+Player, +Board, -NextBoard)
% True if NextBoard is Board whith an empty case replaced by Player mark.
move_aux(P, [0|Bs], [P|Bs]).

move_aux(P, [B|Bs], [B|B2s]) :-
```

```
    move_aux(P, Bs, B2s).

% min_to_move(+Pos)
% True if the next player to play is the MIN player.
min_to_move([o, _, _]).
% max_to_move(+Pos)
% True if the next player to play is the MAX player.
max_to_move([x, _, _]).
% utility(+Pos, -Val) :-
% True if Val the the result of the evaluation function at Pos.
% We will only evaluate for final position.
% So we will only have MAX win, MIN win or draw.
% We will use  1 when MAX win
%             -1 when MIN win
%              0 otherwise.
utility([o, win, _], 1).       % Previous player (MAX) has win.
utility([x, win, _], -1).      % Previous player (MIN) has win.
utility([_, draw, _], 0).

% winPos(+Player, +Board)
% True if Player win in Board.
winPos(P, [X1, X2, X3, X4, X5, X6, X7, X8, X9]) :-
    equal(X1, X2, X3, P) ;    % 1st line
    equal(X4, X5, X6, P) ;    % 2nd line
    equal(X7, X8, X9, P) ;    % 3rd line
    equal(X1, X4, X7, P) ;    % 1st col
    equal(X2, X5, X8, P) ;    % 2nd col
    equal(X3, X6, X9, P) ;    % 3rd col
    equal(X1, X5, X9, P) ;    % 1st diag
    equal(X3, X5, X7, P).     % 2nd diag

% drawPos(+Player, +Board)
% True if the game is a draw.
drawPos(_,Board) :-
    \+ member(0, Board).


% equal(+W, +X, +Y, +Z).
% True if W = X = Y = Z.
equal(X, X, X, X).
```

3- tictactoe_game.pl

```
:- use_module(minimax).
:- use_module(tictactoe).

% bestMove(+Pos, -NextPos)
% Compute the best Next Position from Position Pos
% with minimax or alpha-beta algorithm.
bestMove(Pos, NextPos) :-
    minimax(Pos, NextPos, _).
```

```
% play
% Start the game.
play :-
   nl,
   write('====================='), nl,
      write('= Prolog TicTacToe ='), nl,
      write('====================='), nl, nl,
      write('Rem : x starts the game'), nl,
      playAskColor.



% playAskColor
% Ask the color for the human player and start the game with it.
playAskColor :-
      nl, write('Color for human player ? (x or o)'), nl,
      read(Player), nl,
      (
        Player \= o, Player \= x, !,     % If not x or o -> not a valid color
        write('Error : not a valid color !'), nl,
        playAskColor                     % Ask again
        ;
        EmptyBoard = [0, 0, 0, 0, 0, 0, 0, 0, 0],
        show(EmptyBoard), nl,

        % Start the game with color and emptyBoard
        play([x, play, EmptyBoard], Player)
      ).


% play(+Position, +HumanPlayer)
% If next player to play in position is equal to HumanPlayer -> Human must play
% Ask to human what to do.
play([Player, play, Board], Player) :- !,
   nl, write('Next move ?'), nl,
   read(Pos), nl,                        % Ask human where to play
   (
    humanMove([Player, play, Board], [NextPlayer, State, NextBoard], Pos), !,
    show(NextBoard),
    (
      State = win, !,                    % If Player win -> stop
      nl, write('End of game : '),
      write(Player), write(' win !'), nl, nl
      ;
      State = draw, !,                    % If draw -> stop
      nl, write('End of game : '),
      write(' draw !'), nl, nl
      ;
      play([NextPlayer, play, NextBoard], Player) % Else -> continue the game
```

```
    )
    ;
    write('-> Bad Move !'), nl,              % If humanMove fail -> bad move
    play([Player, play, Board], Player)      % Ask again
  ).



% play(+Position, +HumanPlayer)
% If it is not human who must play -> Computer must play
% Compute the best move for computer with minimax or alpha-beta.
play([Player, play, Board], HumanPlayer) :-
  nl, write('Computer play : '), nl, nl,
  % Compute the best move
  bestMove([Player, play, Board], [NextPlayer, State, BestSuccBoard]),
  show(BestSuccBoard),
  (
    State = win, !,                          % If Player win -> stop
    nl, write('End of game : '),
    write(Player), write(' win !'), nl, nl
    ;
    State = draw, !,                         % If draw -> stop
    nl, write('End of game : '), write(' draw !'), nl, nl
    ;
    % Else -> continue the game
    play([NextPlayer, play, BestSuccBoard], HumanPlayer)
  ).



% nextPlayer(X1, X2)
% True if X2 is the next player to play after X1.
nextPlayer(o, x).
nextPlayer(x, o).

% When human play
humanMove([X1, play, Board], [X2, State, NextBoard], Pos) :-
  nextPlayer(X1, X2),
  set1(Pos, X1, Board, NextBoard),
  (
    winPos(X1, NextBoard), !, State = win ;
    drawPos(X1,NextBoard), !, State = draw ;
    State = play
  ).



% set1(+Elem, +Pos, +List, -ResList).
% Set Elem at Position Pos in List => Result in ResList.
% Rem : counting starts at 1.
set1(1, E, [X|Ls], [E|Ls]) :- !, X = 0.
```

```
set1(P, E, [X|Ls], [X|L2s]) :-
    number(P),
    P1 is P - 1,
    set1(P1, E, Ls, L2s).


% show(+Board)
% Show the board to current output.
show([X1, X2, X3, X4, X5, X6, X7, X8, X9]) :-
    write('   '), show2(X1),
    write(' | '), show2(X2),
    write(' | '), show2(X3), nl,
    write('  -----------'), nl,
    write('   '), show2(X4),
    write(' | '), show2(X5),
    write(' | '), show2(X6), nl,
    write('  -----------'), nl,
    write('   '), show2(X7),
    write(' | '), show2(X8),
    write(' | '), show2(X9), nl.



% show2(+Term)
% Write the term to current outupt
% Replace 0 by ' '.
show2(X) :-
    X = 0, !,
    write(' ').

show2(X) :-
    write(X).
```

- the query: **?- play.**

```
====================
= Prolog TicTacToe =
====================

Rem : x starts the game

Color for human player ? (x or o)
|: x.

     |   |
  -----------
     |   |
  -----------
     |   |


Next move ?
|: 1.

   x |   |
  -----------
     |   |
  -----------
     |   |

Computer play :

   x |   |
  -----------
     | o |
```

```
  -----------
     |   |

Next move ?
|: 2.

   x | x |
  -----------
     | o |
  -----------
     |   |

Computer play :

   x | x | o
  -----------
     | o |
  -----------
     |   |

Next move ?
|: 4.

   x | x | o
  -----------
   x | o |
  -----------
     |   |

Computer play :

   x | x | o
  -----------
   x | o |
  -----------
   o |   |

End of game : o win !

true.
```

Planning

The following program explain the planner work to find a plane for blocks world problem.

```
can(move(Block,From,To),[clear(Block),clear(To),on(Block,From)]):-
     block(Block),
     object(To),
     To\=Block,
     object(From),
     From\=To,
     Block\=From.
adds(move(X,From,To),[on(X,To),clear(From)]).
deletes(move(X,From,To),[on(X,To),clear(From)]).
object(X):-
     place(X);
     block(X).
block(a).
block(b).
block(c).

place(1).
place(2).
place(3).
place(4).

state1([clear(2),
clear(4),
clear(b),
clear(c),
on(a,1),
on(b,3),
on(c,a)]).

plan(State,Goals,[],State):-
     satisfied(State,Goals).
plan(State,Goals,Plan,Finalstate):-
       conc(Plan,_,_),
       conc(Preplan,[Action|Postplan],Plan),
     select(State,Goals,Goal),
     achievs(Action,Goal),
     can(Action,Condition),
     plan(State,Condition,Preplan,Midstate1),
     apply(Midstate1,Action,Midstate2),
     plan(Midstate2,Goals,Postplan,Finalstate).

satisfied(_,[]).
satisfied(State,[Goal|Goals]):-
     member(Goal,State),
     satisfied(State,Goals).
select(State,Goals,Goal):-
     member(Goal,Goals),
     \+member(Goal,State).
```

```
achievs(Action,Goal):-
      adds(Action,Goals),
      member(Goal,Goals).

apply(State,Action,Newstate):-
      deletes(Action,Dellist),
      delete_all(State,Dellist,State1),!,
      adds(Action,Addlist),
      conc(Addlist,State1,Newstate).
delete_all([],_,[]).
delete_all([X|L1],L2,Diff):-
      member(X,L2),!,
      delete_all(L1,L2,Diff).

delete_all([X|L1],L2,[X|Diff]):-
      delete_all(L1,L2,Diff).

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

?- state1(State), plane(State, [on(a,b),on(b,c)],Plan,_).

Note: the program work in iterative deepening fashion: all plans of length n are tried before plans of length n+1.

Probabilistic reasoning by Bayesian networks

The following program shows how we can use the Bayesian network to calculate the probabilities of some events based on some given events.

The main predicate is **pred(Proposition, Cond, P)**.
Where **Proposition** and **Cond** are lists in the form [X1, X2, …]. **P** is the output.

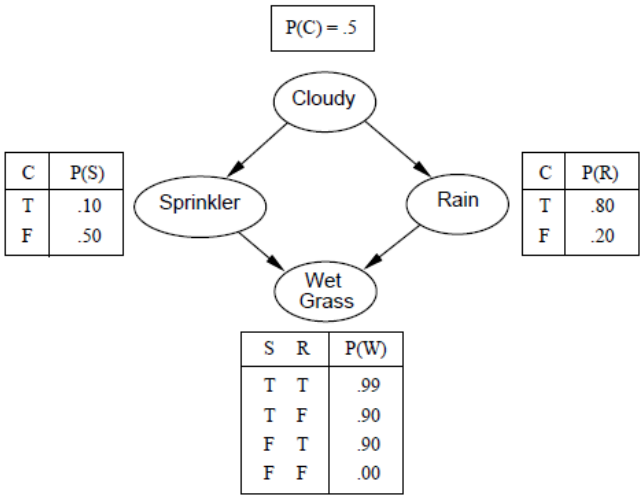
Example:
?- pred(burglary, alarm, P).
?-pred(burglary, [call, not lightning, P).

```
% the program
:- op(900,fy,not).
prob([X|Xs],Cond,P):-!,
        prob(X,Cond,Px),
        prob(Xs,[X|Cond],Prest),
        P is Px*Prest.
prob([],_,1):-!.
prob(X,Cond,1):-
member(X,Cond),!.
prob(X,Cond,0):-
        member(not X,Cond),!.
prob(not X,Cond,P):-!,
     prob(X,Cond,P0),
```

```
        P is 1-P0.
prob(X,Cond0,P):-
        delete(Y,Cond0,Cond),

pred(X,Y),!,
        prob(X,Cond,Px),
        prob(Y,[X|Cond],Pygivenx),
        prob(Y,Cond,Py),
        P is Px * Pygivenx /Py.
prob(X,_,P):-
        p(X,P),!.
prob(X,Cond,P):-!,
        findall((Condi,Pi),p(X,Condi,Pi),Cplist),
        sum_probs(Cplist,Cond,P).
sum_probs([],_,0).
sum_probs([(Cond1,P1)|Condsprobs],Cond,P):-

prob(Cond1,Cond,Pc1),
        sum_probs(Condsprobs,Cond,Prest),
        P is P1*Pc1+Prest.
pred(X, not Y):-!,
        pred(X,Y).
pred(X,Y):-
        parent(X,Y).
pred(X,Z):-
        parent(X,Y),
        pred(Y,Z).

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


delete(X,[X|L],L).
delete(X,[Y|L],[Y|L2]):-
        delete(X,L,L2).

parent(burglary,sensor).
parent(lightning,sensor).
parent(sensor,alarm).
parent(sensor,call).
p(burglary,0.001).
p(lightning,0.02).
p(sensor,[burglary,lightning],0.9).
p(sensor,[burglary,not lightning],0.9).
p(sensor,[not burglary,lightning],0.1).
p(sensor,[not burglary,not lightning],0.001).
p(alarm,[sensor],0.95).
p(alarm,[not sensor],0.001).
p(call,[sensor],0.9).
p(call,[not sensor],0.0).
```

Assessment-6

Given the following Bayesian network of four variables (cloudy, sprinkler, rain, and wetgrass).



Answer the following queries:

1- prob(wetgrass, cloudy,P).

2- prob(rain,[],P).

3- prob(sprinkler,[rain],P).

Decision tree classifier

First we declare the data file. For example weather.pl

% weather.pl

```
attribute(sky, [sunny, cloudy, rain]).

attribute(temperature, [heat, normal, cold]).

attribute(humidity,[high, normal]).

attribute(wind, [weak, strong]).

example(good_day, [ sky = suny, temperature=heat, humidity = high, wind = weak]).

example(good_day,  [sky= suny, temperature=heat, humidity = high, wind = strong]).

example(bad_day, [sky= cloudy, temperature = heat, humidity = high, wind = weak]).

example(bad_day, [sky = rain, temperature=normal, humidity = high, wind = weak]).

example(bad_day, [sky=rain, temperature = cold, humidity = normal, wind = weak]).

example(good_day,  [sky=rain, temperature = cold, humidity = normal, wind = strong]).
```

```
example(bad_day, [sky =cloudy, temperature = cold, humidity = normal, wind = wtrong]).
example(bad_day,  [sky= sunny, temperature=normal, humidity = high, wind = weak]).
example(bad_day, [sky =sunny, temperature = heat, humidity = normal, wind = weak]).
example(bad_day, [sky=rain, temperature=normal, humidity=normal, wind = weak]).
example(bad_day, [sky= sunny, temperature=normal, humidity = normal, wind = strong]).
example(good_day,  [sky =cloudy, temperature=normal, humidity = high, wind = strong]).
example(bad_day,  [sky =cloudy, temperature = heat, humidity = normal, wind = weak]).
example(good_day,  [sky = rain, temperature=normal, humidity = high, wind = strong]).
```

Then we use decision tree classifier program

```
:-consult('weather.pl').
induce_tree(Tree)  :-
   findall( example( Class, Obj), example( Class, Obj), Examples),
   findall( Att, attribute( Att, _ ), Attributes),
   induce_tree( Attributes, Examples, Tree).
% induce_tree( Attributes , Examples, Tree)
induce_tree(_,[], null ) :- !.
induce_tree(_,[example( Class,_) | Examples], leaf( Class)) :-
   \+ ((member(example( ClassX,_), Examples), ClassX \== Class)), !.
induce_tree(Attributes , Examples, tree( Attribute , SubTrees)) :-
   choose_attribute( Attributes , Examples, Attribute/_), ! ,
   del( Attribute , Attributes , RestAtts),
   attribute( Attribute , Values),
   induce_trees( Attribute , Values, RestAtts, Examples, SubTrees).
induce_tree(_, Examples, leaf( ExClasses)) :-
   findall(Class, member( example( Class, _), Examples), ExClasses).
induce_trees(_, [],_,_, [] ).
induce_trees( Att , [Val1 | Vals ], RestAtts, Exs, [Val1 : Tree1 | Trees]) :-
   attval_subset( Att = Val1, Exs, ExampleSubset),
   induce_tree( RestAtts, ExampleSubset, Tree1),
   induce_trees( Att , Vals, RestAtts, Exs, Trees).
```

```
attval_subset( AttributeValue, Examples, ExampleSubset) :-
    findall(example(Class, Obj),
        (member( example( Class, Obj), Examples), satisfy( Obj, [ AttributeValue ])), ExampleSubset).
satisfy( Object, Conj) :- \+ ((member( Att = Val, Conj), member( Att = ValX, Object), ValX \== Val)).
choose_attribute([], _, 0/0).
choose_attribute([Att], Examples, Att/Gain):-!, gain(Examples, Att, Gain).
choose_attribute([Att|Atts], Examples, BestAtt/BestGain):-
    choose_attribute(Atts,Examples,BestAtt1/BestGain1),
    gain(Examples, Att, Gain),
    (Gain>BestGain1,!,BestAtt=Att,BestGain=Gain;
    BestAtt=BestAtt1,BestGain=BestGain1).
gain( Exs, Att , Gain) :- attribute( Att , AttVals ),
                length(Exs, Total),
                setof(Class, X^example(Class,X), Classes),
                findall(Nc, (member(C,Classes), cntclass(C,Exs,Nc)), CCnts),
                info(CCnts,Total,I),
                rem(Att, AttVals,Exs,Classes,Total,Rem),
                Gain is I-Rem.
info([], _, 0).
info([VC|ValueCounts], Total, I) :-
                info(ValueCounts,Total,I1),
                (VC = 0, !, I is I1;
                Pvi is VC / Total,
                log2(Pvi, LogPvi), I is - Pvi * LogPvi + I1).
rem( _, [], _, _, _, 0).
rem( Att, [V | Vs], Exs, Classes, Total, Rem) :-
                findall(1, (member(example(_, AVs),Exs), member(Att = V, AVs)), L1), length(L1, Nv),
                findall(Ni, (member(C, Classes), cntclassattv(Att,V,C,Exs,Ni)), VCnts),
                Pv is Nv / Total,  % P(v)
                info(VCnts,Nv,I),
```

```
                rem(Att,Vs,Exs,Classes,Total,Rem1),
                Rem is Pv * I + Rem1.
cntclass( Class, Exs, Cnt) :-
                findall(1, member(example(Class,_),Exs), L), length(L, Cnt).
cntclassattv( Att, Val, Class, Exs, Cnt) :-
                findall(1, (member(example(Class,AVs),Exs), member(Att = Val, AVs)), L), length(L, Cnt).
log2(X, Y) :- Y is log(X) / log(2).
% show(+X,+L,-L1)
del(A,[A|T],T).
del(A,[H|T1],[H|T2]) :- del(A,T1,T2).
% show(+Tree)
show(Tree) :-
                show(Tree, 0).
% show(+Tree, +Ind)
show(leaf(Class), Ind) :-
   tab(Ind), write(Class), nl.
show(tree(A, SubTrees), Ind) :-
   tab(Ind), write(A), write('?'), nl,
   NI is Ind+2, show(SubTrees, NI).
show([], _).
show([_ : null | SubTrees], Ind) :- !, show(SubTrees, Ind).
show([V1 : ST1 | SubTrees], Ind) :-
   tab(Ind), write('= '), write(V1), nl,
   NI is Ind+2, show(ST1, NI),
   show(SubTrees, Ind).
:-induce_tree(T),show(T).
```

The output: the decision tree.

```
wind?
  = weak
    sky?
      = sunny
        bad_day
      = cloudy
        bad_day
      = rain
        bad_day
  = strong
    sky?
      = sunny
        bad_day
      = cloudy
        good_day
      = rain
        good_day
```

Assignment-7

Use the program of decision tree to build the tree of *restaurant* problem.