

**Mechi Multiple Campus**

**Tribhuvan University**

**Bhadrapur, Jhapa**

**Lab Report of Artificial Intelligence(CACS-401)**

Faculty of Humanities and Social Sciences

Tribhuvan University

Kritipur, Nepal

**Submitted By**

Name: Rajeev Kamat

Roll: 22

**Submitted to**

Mechi Multiple Campus

Bachelor in Computer Application

Bhadrapur Jhapa, Nepal

|  |  |
| --- | --- |
| **S.No** | **Practical’s Name** |
| 1 | Study of Prolog. |
| 2 | Write simple fact for the statements using PROLOG. |
| 3 | Write the prolog program for the following predicate logic |
| 4 | Write a prolog program to find minimum maximum of two numbers. |
| 5 | Write a prolog program to verify whether a line segment is horizontal, vertical or oblique. |
| 6 | Write a prolog program to implement Tower of Hanoi problem. |
| 7 | Write a program to solve the Monkey Banana problem. |
| 8 | Write program to perform BFS and DFS search operation of the given graph. (any high level language) here we use python. |
| 9 | Write program to perform A\* search operation of the given graph. |

1. Study of Prolog.

**PROLOG-PROGRAMMING IN LOGIC**

PROLOG stands for Programming, In Logic — an idea that emerged in the early 1970’s to use logic as programming language. The early developers of this idea included Robert Kowaiski at Edinburgh (on the theoretical side), Marrten van Emden at Edinburgh (experimental demonstration) and Alian Colmerauer at Marseilles (implementation).

David D.H. Warren’s efficient implementation at Edinburgh in the mid -1970’s greatly contributed to the popularity of PROLOG. PROLOG is a programming language centered around a small set of basic mechanisms, Including pattern matching, tree based data structuring and automatic backtracking. This Small set constitutes a surprisingly powerful and flexible programming framework. PROLOG is especially well suited for problems that involve objects- in particular, structured objects- and relations between them.

**SYMBOLIC LANGUAGE**

PROLOG is a programming language for symbolic, non-numeric computation. It is especially well suited for solving problems that involve objects and relations between objects. For example, it is an easy exercise in prolog to express spatial relationship between objects, such as the blue sphere is behind the green one. It is also easy to state a more general rule: if object X is closer to the observer than object Y. and object Y is closer than Z, then X must be closer than Z. PROLOG can reason about the spatial relationships and their consistency with respect to the general rule. Features like this make PROLOG a powerful language for ArtJIcia1 LanguageA1,) and non- numerical programming.

There are well-known examples of symbolic computation whose implementation in other standard languages took tens of pages of indigestible code, when the same algorithms were implemented in PROLOG, the result was a crystal-clear program easily fitting on one page.

**FACTS, RULES AND QUERIES**

Programming in PROIOG is accomplished by creating a database of facts and rules about objects, their properties, and their relationships to other objects. Queries then can be posed about the objects and valid conclusions will be determined and returned by the program Responses to user queries are determined through a form of inference control known as resolution.

**FOR EXAMPLE:**

1. FACTS:

Some facts about family relationships could be written as:

sister( sue,bill)

parent( ann.sam)

male(jo)

female( riya)

1. RULES:

To represent the general rule for grandfather, we write:

grand father( X2)

parent(X,Y)

parent( Y,Z)

male(X) c)

1. QUERIES:

Given a database of facts and rules such as that above, we may make queries by typing after a query a

symbol’?’ statements such as:

?-parent(X,sam) Xann

?grandfather(X,Y)

X=jo, Y=sam

**PROLOG IN DISGINING EXPERT SYSTEMS**

An expert system is a set of programs that manipulates encoded knowledge to solve problems in a specialized domain that normally requires human expertise. An expert system’s knowledge is obtained from expert sources such as texts, journal articles. databases etc and encoded in a form suitable for the system to use in its inference or reasoning processes. Once a sufficient body of expert knowledge has been acquired, it must be encoded in some form, loaded into knowledge base, then tested, and refined continually throughout the life of the system PROLOG serves as a powerful language in designing expert systems because of its following features.

* Use of knowledge rather than data
* Modification of the knowledge base without recompilation of the control programs.
* Capable of explaining conclusion.
* Symbolic computations resembling manipulations of natural language.
* Reason with meta-knowledge.

**META PROGRAMMING**

A meta-program is a program that takes other programs as data. Interpreters and compilers are examples of mela-programs. Meta-interpreter is a particular kind of meta-program: an interpreter for a language written in that language. So a PROLOG interpreter is an interpreter for PROLOG, itself written in PROLOG. Due to its symbol- manipulation capabilities, PROLOG is a powerful language for meta-programming. Therefore, it is often used as an implementation language for other languages. PROLOG is particularly suitable as a language for rapid prototyping where we are interested in implementing new ideas quickly. New ideas are rapidly implemented and experimented with.

**Outcome:**

Prolog is a programming language that is well-suited for tasks involving symbolic reasoning and manipulation. It is based on first-order logic and uses a declarative programming paradigm.

To program in Prolog, you will need a Prolog compiler or interpreter, such as SWI-Prolog or GNU Prolog. These can be downloaded and installed on your computer.

Once you have a Prolog environment set up, you can start writing Prolog programs. Prolog programs consist of facts and rules. Facts are statements that are true in the program's domain, and rules are logical statements that infer new facts based on existing ones.

For example, here is a simple Prolog program that defines a parent-child relationship between two people:

parent(john, jim).

parent(jane, jim).

parent(jim, joe).

This program defines three facts, stating that John is a parent of Jim, Jane is a parent of Jim, and Jim is a parent of Joe.

You can also write rules in Prolog, for example:

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

This rule states that X is a grandparent of Y if X is a parent of Z and Z is a parent of Y.

You can use the Prolog interpreter to query these facts and rules. For example, you can ask Prolog if John is a parent of Jim:

?- parent(john, jim).

Prolog will return true.

You can also ask Prolog to find all of the grandparents in the program:

Copy code

?- grandparent(X, Y).

Prolog will return all of the possible values of X and Y that make the rule grandparent(X, Y) true.

1. Write simple fact for the statements using PROLOG.

a. Ram likes mango.

b. Seema is a girl.

c. Bill likes Cindy.

d. Rose is red.

e. John owns gold.

**PROLOG Program**

likes(ram ,mango).

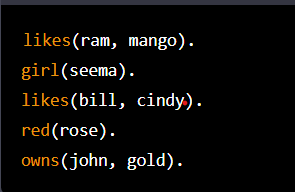
girl(seema).

red(rose).

likes(bill ,cindy).

owns(john ,gold).

**Output:**

****

1. Write the prolog program to execute the following predicate logic

Facts:

* Tom is male.
* John is male.
* Ann is female.
* Mary is female.
* Tom is a patent of john.
* Tom is a parent of Mary.
* Ann is a parent of John.
* Ann is a parent of Mary.

Rules:

* Someone is the father of a person if that someone is a parent of the person and that someone is a male.
* Someone is the mother of a person if that someone is a parent of the person and that someone is a female.

Question:

1. Who is John’s Father?
2. Is Tom the patent of Mary?
3. Is Ann the parent of Mary?

**Prolog Code**

male(tom).

male(john).

female(ann).

female(mary).

parent(tom,john).

parent(tom,mary).

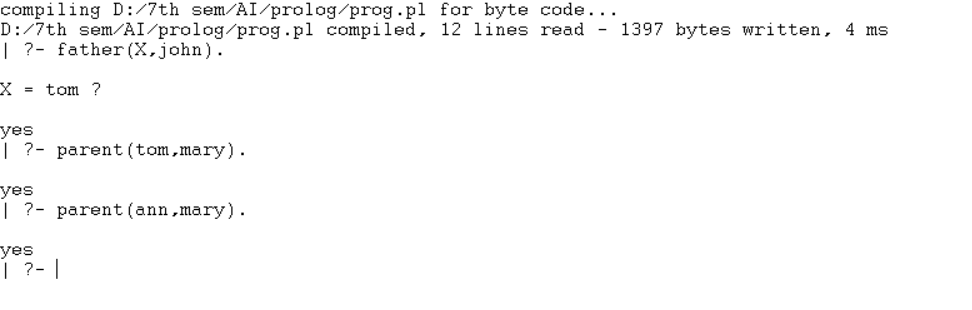
parent(ann,john).

parent(ann,mary).

father(X,Y) :- parent(X,Y),male(X).

mother(X,Y) :- parent(X,Y),female(X).

**Output:**



1. Write a prolog program to find minimum maximum of two numbers.

Here we will see one Prolog program, that can find the minimum of two numbers and the maximum of two numbers. First, we will create two predicates, **find\_max(X,Y,Max)**. This takes X and Y values, and stores the maximum value into the Max. Similarly **find\_min(X,Y,Min)** takes X and Y values, and store the minimum value into the Min variable.

**Prolog program:**

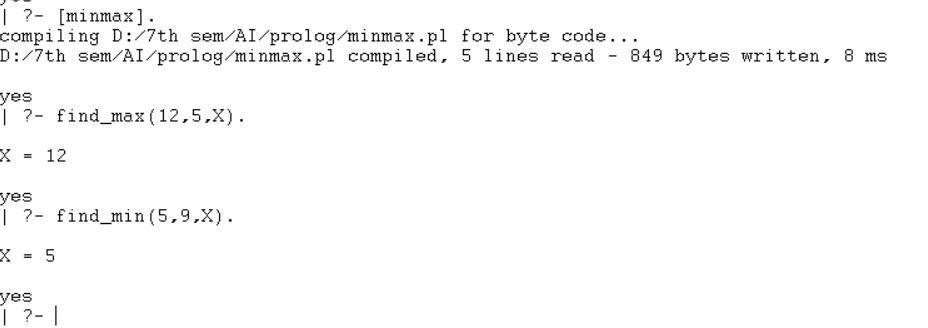
find\_max(X, Y, X) :- X >= Y, !.

find\_max(X, Y, Y) :- X < Y.

find\_min(X, Y, X) :- X =< Y, !.

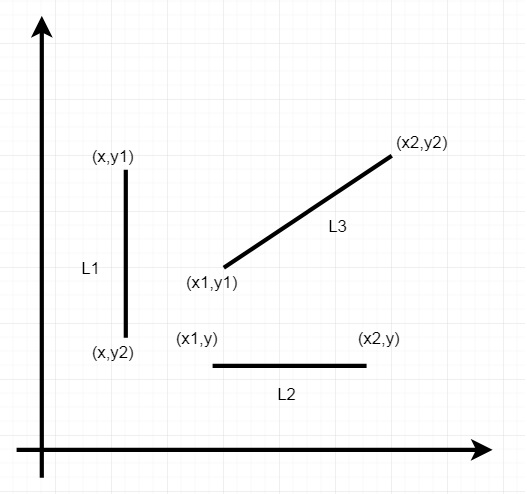
find\_min(X, Y, Y) :- X > Y.

**Output :**



1. Write a prolog program to verify whether a line segment is horizontal, vertical or oblique.

There are three types of line segments, horizontal, vertical or oblique. This example verifies whether a line segment is horizontal, vertical or oblique.



From this diagram we can understand that −

* For Horizontal lines, the y coordinate values of two endpoints are same.
* For Vertical lines, the x coordinate values of two endpoints are same.
* For Oblique lines, the (x,y) coordinates of two endpoints are different.

Now let us see how to write a program to check this.

Prolog program:

vertical(seg(point(X,\_),point(X,\_))).

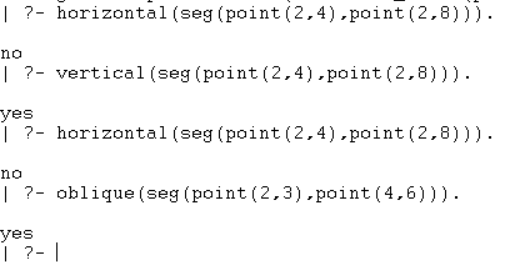
horizontal(seg(point(\_,Y),point(\_,Y))).

oblique(seg(point(X1,Y1),point(X2,Y2)))

:-X1 \== X2,

Y1 \== Y2.

**Output :**

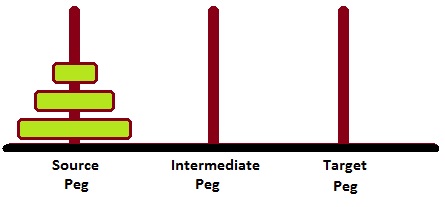


1. Write a prolog program to implement Tower of Hanoi problem.

Towers of Hanoi Problem is a famous puzzle to move N disks from the source peg/tower to the target peg/tower using the intermediate peg as an auxiliary holding peg. There are two conditions that are to be followed while solving this problem −

* A larger disk cannot be placed on a smaller disk.
* Only one disk can be moved at a time.

The following diagram depicts the starting setup for N=3 disks.



To solve this, we have to write one procedure move(N, Source, Target, auxiliary). Here N number of disks will have to be shifted from Source peg to Target peg keeping Auxiliary peg as intermediate.

For example – move(3, source, target, auxiliary).

* Move top disk from source to target
* Move top disk from source to auxiliary
* Move top disk from target to auxiliary
* Move top disk from source to target
* Move top disk from auxiliary to source
* Move top disk from auxiliary to target
* Move top disk from source to target.

**Prolog Program:**

move(1,X,Y,\_) :-

write('Move top disk from '), write(X), write(' to '), write(Y), nl.

move(N,X,Y,Z) :-

N>1,

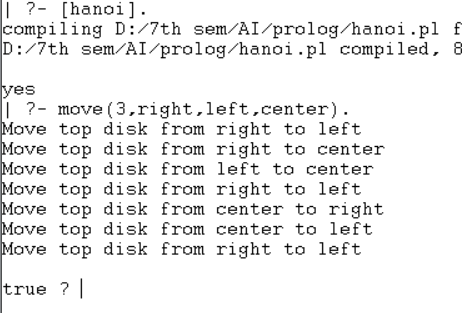
M is N-1,

move(M,X,Z,Y),

move(1,X,Y,\_),

move(M,Z,Y,X).

**Output:**



1. Write a program to solve the Monkey Banana problem.

**Problem Statement**

Suppose the problem is as given below −

* A hungry monkey is in a room, and he is near the door.
* The monkey is on the floor.
* Bananas have been hung from the center of the ceiling of the room.
* There is a block (or chair) present in the room near the window.
* The monkey wants the banana, but cannot reach it.



## So how can the monkey get the bananas?

So if the monkey is clever enough, he can come to the block, drag the block to the center, climb on it, and get the banana. Below are few observations in this case −

* Monkey can reach the block, if both of them are at the same level. From the above image, we can see that both the monkey and the block are on the floor.
* If the block position is not at the center, then monkey can drag it to the center.
* If monkey and the block both are on the floor, and block is at the center, then the monkey can climb up on the block. So the vertical position of the monkey will be changed.
* When the monkey is on the block, and block is at the center, then the monkey can get the bananas.

Now, let us see how we can solve this using Prolog. We will create some predicates as follows −

We have some predicates that will move from one state to another state, by performing action.

* When the block is at the middle, and monkey is on top of the block, and monkey does not have the banana (i.e. ***has not*** state), then using the ***grasp*** action, it will change from ***has not*** state to ***have*** state.
* From the floor, it can move to the top of the block (i.e. ***on top*** state), by performing the action ***climb***.
* The **push** or **drag** operation moves the block from one place to another.
* Monkey can move from one place to another using **walk** or **move** clauses.

Another predicate will be canget(). Here we pass a state, so this will perform move predicate from one state to another using different actions, then perform canget() on state 2. When we have reached to the state ‘**has>**’, this indicates ‘**has banana**’. We will stop the execution.

**Prolog Program.**

move(state(middle,onbox,middle,hasnot),

grasp,

state(middle,onbox,middle,has)).

move(state(P,onfloor,P,H),

climb,

state(P,onbox,P,H)).

move(state(P1,onfloor,P1,H),

drag(P1,P2),

state(P2,onfloor,P2,H)).

move(state(P1,onfloor,B,H),

walk(P1,P2),

state(P2,onfloor,B,H)).

canget(state(\_,\_,\_,has)).

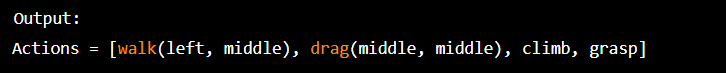
canget(State1) :-

move(State1,\_,State2),

canget(State2).

Output:



****

1. Write program to perform BFS and DFS search operation of the given graph.

A

B C

D E F

*Students work: (explain bfs and dfs theoretically with algorithm)*

import networkx as nx

import matplotlib.pyplot as plt

graph={

"A":['B','C'],

"B":['D','E'],

"C":['F'],

"D":[],

"E":['F'],

"F":[]

}

#depth first search

visited=set() # set to keep track of visited node

def dfs(visited,graph,node):

if node not in visited:

print(node)

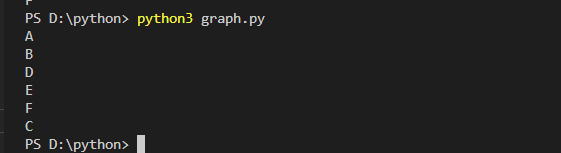
visited.add(node)

for neighbour in graph[node]:

dfs(visited,graph,neighbour)

#call

dfs(visited,graph,'A')



#breadth first search

visited=[] # list to keep track of visited node

queue=[] # initialize the queue

def bfs(visited,graph,node):

visited.append(node)

queue.append(node)

while queue:

s=queue.pop(0)

print(s,end="\n")

for neighbour in graph[s]:

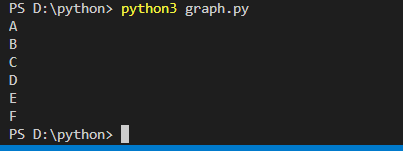
if neighbour not in visited:

visited.append(neighbour)

queue.append(neighbour)

#call

bfs(visited,graph,"A")



1. Write a program to implement A\* search operation on the given graph.

Start

A B

C D E F

Goal

|  |  |
| --- | --- |
| Node | Heuristic |
| Start | 10 |
| A | 5 |
| B | 6 |
| C | 8 |
| D | 5 |
| E | 4 |
| F | 6 |
| Goal | 0 |

**Python code:**

import networkx as nx,matplotlib.pyplot as plt

G=nx.Graph()

G.add\_edge("start","a",weight=10)

G.add\_edge("start","b",weight=6)

G.add\_edge("a","c",weight=3)

G.add\_edge("a","d",weight=8)

G.add\_edge("c","goal",weight=6)

G.add\_edge("d","goal",weight=9)

G.add\_edge("b","f",weight=4)

G.add\_edge("b","e",weight=7)

G.add\_edge("e","goal",weight=2)

G.add\_edge("f","goal",weight=10)

pos = nx.spring\_layout(G)

nx.draw\_networkx\_nodes(G,pos,node\_size=700)

plt.figure(figsize=(5,4))

nx.draw\_networkx(G)

labels = nx.get\_edge\_attributes(G,'weight')

pos = nx.spring\_layout(G)

plt.figure(figsize=(10.7))

nx.draw\_networkx(G,pos)

nx.draw\_networkx\_edge\_labels(G,pos,edge\_labels=labels,font\_color='b')

print(nx.astar\_path(G,'start','goal',heuristic=None,weight="weight"))

**Output:**

['start', 'b', 'e', 'goal']

