SRM VALLIAMMAI ENGINEERING COLLEGE

(An Autonomous Institution)

SRM Nagar, Kattankulathur – 603 203

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

LAB MANUAL

VII SEMESTER

1904009– ARTIFICIAL INTELLIGENCE LABORATORY

Regulation – 2019

Academic Year 2024 – 2025

ODD SEMESTER

Prepared by

Mr.Rajasekaran.T Assistant Professor (Sr.G)-CSE

Dr.Venkatesh.K, Assistant Professor (Sr.G)-CSE

Ms.S.Suma, Assistant Professor (Sel.G)-CSE

LIST OF PROGRAMS	
1904009-ARTIFICIAL INTELLIGENCE LABORATORY(Syllabus)	
S.NO	**PROGRAM**
1	Study of PROLOG.
2	Write a program to solve 8 queens problem
3	Solve any problem using depth first search.
4	Solve any problem using best first search.
5	Solve 8-puzzle problem using best first search.
6	Solve Robot (traversal) problem using means End Analysis.
7	Solve traveling salesman problem.

Books for References :

- Artificial Intelligence: A Modern Approach,. Russell & Norvig. 1995, Prentice Hall.
- Artificial Intelligence, Elain Rich and Kevin Knight, 1991, TMH.
- Artificial Intelligence-A modern approach, Staurt Russel and peter norvig, 1998, PHI.
- Artificial intelligence, Patrick Henry Winston:, 1992, Addition Wesley 3 Ed.,

LIST OF PROGRAMS	
1904009-ARTIFICIAL INTELLIGENCE LABORATORY	
S.NO	**PROGRAM**
1	Study of PROLOG.
2	Simple program using variables and operators
3	Simple program to handle list
4	Write a Prolog program to solve 8 queens problem
5	Write a Prolog program to solve any problem using depth first search.
6	Write a Prolog program to solve traveling salesman problem.
7	Write a Prolog program to solve 8-puzzle problem using best first search.
8	Write a Prolog program to solve Monkey banana problem
9	Write a Prolog program to solve Water jug problem
10	Write a Prolog program to solve Robot (traversal) problem using means End Analysis.

PROGRAM 1 : STUDY OF PROLOG.

Using *Turbo* Prolog
Topics:

a) Basics of Turbo Prolog
b) Intro to Prolog programming
c) Running a simple program

- Prolog is a logical programming language and stands for PROgramming in LOGic
- Created around 1972
- Preferred for AI programming and mainly used in such areas as:
  - Theorem proving, expert systems, NLP, …
- Logical programming is the use of mathematical logic for computer programming.

To start Turbo Prolog, open a MSDOS window and type:
N> prolog
followed by a carriage return.

The GUI:

- GUI is composed of four panels and a main menu bar.
- The menu bar lists **six** options – Files, Edit, Run, Compile, Options, Setup.
- The four panels are Editor, Dialog, Message and Trace.

MENU

- **Files** – Enables the user to load programs from disk, create new programs, save modified programs to disk, and to quit the program.
- **Edit** – Moves user control to the Editor panel.
- **Run** – Moves user control to the Dialog panel ; compiles the user program (if not already done so) in memory before running the program.
- **Compile** – Provides the user with choices on how to save the compiled version of the program.
- **Options** – Provides the user with choices on the type of compilation to be used.
- **Setup** – Enables the user to change panel sizes, colors, and positions.

Editor

Simple to use editor with support for common editing tasks. **Function**	**Command**
Character left/right	left arrow/right arrow
Word left/right	Ctrl-left arrow/Ctrl-right arrow
Line up/down	up arrow/down arrow
Page up/down	PgUp/PgDn
Beginning/End of line	Home/End
Delete character	Backspace/Delete
Delete line	Ctrl-Y
Search	Ctrl-QF
Replace	Ctrl-QA

Dialog

- When a Prolog program is executing, output will be shown in the Dialog Panel

Message

- The Message Panel keeps the programmer up to date on processing activity.

Trace

- The Trace Panel is useful for finding problems in the programs you create.

Prolog Clauses

Any factual expression in Prolog is called a clause.
There are two types of factual expressions: facts and rules

- **There are three categories of statements in Prolog:**
  - **Facts:** Those are true statements that form the basis for the knowledge base.
  - **Rules:** Similar to functions in procedural programming (C++, Java…) and has the form of if/then.
  - **Queries:** Questions that are passed to the interpreter to access the knowledge base and start the program.

What is a Prolog program?

❖ Prolog is used for solving problems that involve objects and the relationships between objects.

- A program consists of a database containing one or more facts and zero or more rules(next week).
- A fact is a relationship among a collection of objects. A fact is a one-line statement that ends with a full-stop.

```
parent (john, bart).
parent (barbara, bart).
male (john).
  dog(fido). >> Fido is a dog or It is true that fido is a dog
sister(mary, joe). >> Mary is Joe's sister.
  play(mary, joe, tennis). >> It is true that Mary and Joe play tennis.
```

- Relationships can have any number of objects.
- Choose names that are meaningful – because in Prolog names are arbitrary strings but people will have to associate meaning to them.

Facts… Syntax
rules:

1. The names of all relationships and objects must begin with a lower case letter. For example: likes, john, rachel.
2. The relationship is written first, and the objects are written separated by commas, and the objects are enclosed by a pair of round brackets.
3. The character '.' must come at the end of each fact.

Terminology:

1. The names of the objects that are enclosed within the round brackets in each fact are called arguments.
2. The name of the relationship, which comes just before the round brackets, is called the predicate.
3. The arguments of a predicate can either be names of objects (constants) or variables.
4. When defining relationships between objects using facts, attention should be paid to the order in which the objects are listed. While programming the order is arbitrary, however the programmer must decide on some order and be consistent.
5. Ex. likes(tom, anna). >> The relationship defined has a different meaning if the order of the objects is changed. Here the first object is understood to be the "liker". If we wanted to state that Anna also likes Tom then we would have to add to our database – likes(anna, tom).
6. Remember that we must determine how to interpret the names of objects and relationships.

Constants & Variables

Constants

- Constants are names that begin with lower case letters.
- Names of relationships are constants

Variables

- Variables take the place of constants in facts.
- Variables begin with upper case letters.

Turbo Prolog Program

A Turbo Prolog program consists of two or more sections.

Clauses Section

- The main body of the prolog program.
- Contains the clauses that define the program – facts and rules.

Predicates Section

- Predicates (relations) used in the clauses section are defined.
- Each relation used in the clauses of the clauses section must have a corresponding predicate definition in the predicates section. Except for the built in predicates of Turbo Prolog.

Turbo Prolog requires that each predicate in the predicate section must head at least one clause in the clauses section.
A predicate definition in the predicates section does **not** end with a period.
Predicate definitions contain different names than those that appear in the clauses section. Make sure that the predicate definition contains the same number of names as the predicate does when it appears in the clauses section.

A Turbo Prolog may also have a domains section. In this section the programmer can define the type of each object.
Examples:
Clauses Section – likes(tom, anna).

Predicates Section – likes(boy, girl)
Domains Section – boy, girl = symbol
It is possible to omit the domains section by entering the data types of objects in the predicates section.
likes(symbol,symbol)
However, this might make the program harder to read especially if the predicate associates many objects.

Simple Program

```
domains
        disease,indication = symbol
predicates
        symptom(disease, indication)
clauses
        symptom(chicken_pox, high_fever).
        symptom(chicken_pox, chills).
        symptom(flu,  chills).
        symptom(cold, mild_body_ache).
        symptom(flu, severe_body_ache).
        symptom(cold, runny_nose).
        symptom(flu, runny_nose).
        symptom(flu, moderate_cough).
```

Executing Simple Program

- Start Turbo Prolog
- Select the Edit mode
- Type in the program
- Exit the Editor using Esc.
- Save the program
- Select Run (which compiles the program for you in memory)

Once you have followed these steps you will see the following prompt in the Dialog Panel:

Goal:

Using a Prolog program is essentially about asking questions. To ask the executing Prolog program a question you specify the Goal.
Ex -
Goal: symptom(cold, runny_nose)
True
Goal:
Turbo Prolog will respond with True and prompt for another goal.
Possible outcomes of specifying a goal:

1. The goal will succeed; that is, it will be proven true.
2. The goal will fail; Turbo Prolog will not be able to match the goal with any facts in the program.
3. The execution will fail because of an error in the program.

Execution is a matching process. The program attempts to match the goal with one of the clauses in the clauses section beginning with the first clause. If it does find a complete match, the goal succeeds and *True* is

displayed. In Prolog, False indicates a failure to find a match using the current database – not that the goal is untrue.

Variables Revisited

Variables are used in a clause or goal to specify an unknown quantity. Variables enable the user to ask more informative questions. For example, if we wanted to know for which diseases, runny_nose was a symptom – type in

```
Goal: symptom(Disease, runny_nose).
Turbo Prolog will respond
Disease = cold
Disease = flu
2 Solutions
Goal:
```

To find the two solutions Turbo Prolog began at the start of the clauses section and tried to match the goal clause to one of the clauses. When a match succeeded, the values of the variables for the successful match was displayed. Turbo prolog continued this process until it had tested all predicates for a match with the specified goal.

If you wish Prolog to ignore the value of one or more arguments when determining a goal's failure or success then you can use the anonymous variable "_" (underscore character).

Ex -

```
Goal: symptom(_, chills).
True
Goal:
```

Matching

Two facts match if their predicates are the same, and if their corresponding arguments are the same.

When trying to match a goal that contains an uninstantiated variable as an argument, Prolog will allow that argument to match any other argument in the same position in the fact.

If a variable has a value associated with a value at a particular time it is instantiated otherwise it is uninstantiated.

Heuristics

- A method to help solve a problem, commonly informal.
- It is particularly used for a method that often rapidly leads to a solution that is usually reasonably close to the best possible answer.
- Heuristics are "rules of thumb", educated guesses, intuitive judgments or simply common sense.

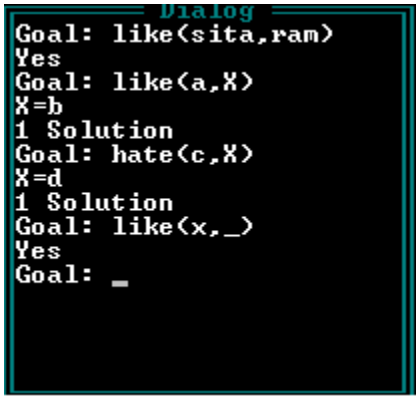
Program to demonstrate a simple prolog program.

```
predicates
        like(symbol,symbol)
        hate(symbol,symbol)
clauses
        like(sita,ram).
        like(x,y).
        like(a,b).

        hate(c,d).
        hate(m,n).
```

hate(f,g).

Output:-



PROGRAM 2:SIMPLE PROGRAM USING VARIABLES AND OPERATORS

1.FOOD RELATION

PROGRAM

Facts
```
food(burger).
food(sandwich).
food(pizza).
lunch(sandwich).
dinner(pizza).
```

Rules
```
meal(X) :- food(X).
```

Queries / Goals
```
?- food(pizza).
?- meal(X), lunch(X).
?- dinner(sandwich).
```

OUTPUT

```
?- ['1.PL'].
true
?- food(pizza).
true
?- meal(X), lunch(X).
X= sandwich
?- dinner(sandwich).
false
```

2.STUDENT – TEACHER RELATION

PROGRAM

Facts
```
studies(charlie, csc135).
studies(olivia, csc135).
studies(jack, csc131).
studies(arthur, csc134).
teaches(kirke, csc135).
teaches(collins, csc131).
teaches(collins, csc171).
teaches(juniper, csc134).
```

Rules
```
professor(X, Y) :-
teaches(X, C), studies(Y, C).
```

Queries / Goals
?- studies(charlie, What).
?- professor(kirke, Students).
OUTPUT

?- [‘2.pl’].
true
?- studies(Charlie,What).
What = cse135.
?- professor(kirke,students).
false
?- professor(kirke,Students).
Students = Charlie;
Students = Olivia.

3: PROGRAM TO DEMONSTRATE FAMILY RELATIONSHIP

predicates
parent(symbol,symbol)
child(symbol,symbol)
mother(symbol,symbol)
brother(symbol,symbol)
sister(symbol,symbol)
grandparent(symbol,symbol)
male(symbol)
female(symbol)
clauses
parent(a,b).
sister(a,c).
male(a).
female(b).
child(X,Y):-parent(Y,X).
mother(X,Y):-female(X),parent(X,Y).
grandparent(X,Y):-parent(X,Z),parent(Z,Y).
brother(X,Y):-male(X),parent(V,X),parent(V,Y).

Output:
Goal: chilf(X,h)
No solution
Goal: female(b)
Yes
Goal: male(a)
Yes

4.PROGRAM TO CATEGORISE ANIMAL CHARACTERISTICS.
predicates
small(symbol)
large(symbol)
color(symbol,symbol)

clauses
small(rat).
small(cat).
large(lion).
color(dog,black).
color(rabbit,white).
color(X,dark):-
color(X,black);color(X,brown).

Output
Goal: small(X)
X=rat
X=cat
2 solutions

5: PROGRAM TO SHOW HOW INTEGER VARIABLE IS USED IN PROLOG PROGRAM
predicates
go
clauses
go:-X=10,
write(X),
nl,X=20,
write(X),nl.

Output:
Goal: go
10
No

6.ARITHMETIC OPERATIONS

PROGRAM & OUTPUT

?- X is 3+2. // expression on right side of 'is'

X = 5.

?- 3+2 is X. // expression on left side of 'is'

ERROR: is/2: Arguments are not sufficiently instantiated

?- X = 3+2. // just instantiate variable X to value 3+2

X = 3+2.

?- 3+2 = X.

X = 3+2.

?- X is +(3,2).

X = 5.

?- 5 is 3+2.

true.

?- 3+2 is 5.

false.

?- X is 3*2.

X = 6.

?- X is 3-2.

X = 1.

?- X is -(2,3).

X = -1.

?- X is 5-3-1.

X = 1.

?- X is -(5,3,1).

ERROR: is/2: Arithmetic: `(-)/3' is not a function

?- X is -(-(5,3),1).

X = 1.

?- X is 5-3-1.

X = 1

?- X is 3/5.

X = 0.6.

?- X is 3 mod 5.

X = 3.

?- X is 5 mod 3.

X = 2.

?- X is 5^3.

X = 125.

?- X is (5^3)^2.

X = 15625.

?- X = (5^3)^2.

X = (5^3)^2.

?- 25 is 5^2.

true.

?- Y is 3+2*4-1.

Y = 10.

?- Y is (3+2)*(4)-(1).

Y = 19.

?- Y is -(*(+(3,2),4),1).

Y = 19.

?- X is 3*2, Y is X*2.

X = 6,

Y = 12.

7. PROGRAM TO ADD TWO NUMBERS.

predicates
add
clauses
add:-write("input first number"),
readint(X),
write("input second number"),
readint(Y),
Z=X+Y,write("output=",Z).

Output:
Goal: add
Input first number 4
Input second number 7

Output=11 yes

8: PROGRAM TO READ ADDRESS OF A PERSON USING COMPOUND VARIABLE .

```
domains
person=address(name,street,city,state,zip)
name,street,city,state,zip=String
predicates
readaddress(person)
go
clauses
go:-
readaddress(Address),nl,write(Address),nl,nl,write("Accept(y/n)?"),readchar(Reply),Reply='y',!.
go:-
nl,write("please re-enter"),nl,go.
readaddress(address(N,street,city,state,zip)):-
write("Name:"),readln(N),
write("Street:"),readln(street),
write("City:"),readln(city),
write("State:"),readln(state),
write("Zip:"),readln(zip).
```

Output:

```
Goal: go
Name: Ram
Street: South Street
City: Trichy
State:Tamilnadu
Zip: 621000
Address(“Ram”,”South Street”,”Trichy”,”Tamilnadu”,”621000”).
```

9: PROGRAM OF FUN TO SHOW CONCEPT OF CUT OPERATOR .

```
predicates
fun(integer,integer)
clauses
fun(Y,1):-Y<3,!.
fun(Y,2):-Y>3,Y<=10,!.
fun(Y,3):-Y>10,!.
```

Output:

```
Goal: fun(4,1)
No.
Goal: fun(2,3)
Yes
```

10.TOWERS OF HANOI

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).

?-  move(3,left,right,center).
```

OUTPUT

```
Move top disk from left to right
    Move top disk from left to center
    Move top disk from right to center
    Move top disk from left to right
    Move top disk from center to left
    Move top disk from center to right
    Move top disk from left to right
```

PROGRAM 3:SIMPLE PROGRAM TO HANDLE LIST

1 : PROGRAM TO COUNT NUMBER OF ELEMENTS IN A LIST .

```
domains
x=integer
list=integer*
predicates
count(list,x)
clauses
count([],0).
count([_|T],N):-count(T,N1),N=N1+1.
```

Output:

```
Goal: count([],X)
X=0
1 solution
Goal: count([1,2,3,4,5,6,7,8,9,0,[],X)
X=11
1 solution
```

2 : PROGRAM TO REVERSE THE LIST .

```
domains
x=integer
list=integer*
predicates
append(x,list,list)
rev(list,list)
clauses
append(X,[],[X]).
append(X,[H|T],[H|T1]):-append(X,T,T1).
rev([],[]).
rev([H|T,rev):-rev(T,L),append(H,L,rev).
```

Output:

```
Goal:append(2,[3,4,5],X)
X=[3,4,5,2]
1 solution
Goal: rev([2,3,4],X).
X=[4,3,2] 1 solution
```

3: PROGRAM TO APPEND AN INTEGER INTO THE LIST .

```
domains
x=integer
list=integer*
predicates
append(x,list,list)
clauses
append(X,[],[X]).
```

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

Output:

```
Goal: append([1,[2,3,4,5],X)
X=[2,3,4,5,1]
1 solution
```

4: PROGRAM TO REPLACE AN INTEGER FROM THE LIST .

```
domains
list=integer*
predicates
replace(integer,integer,list,list)
clauses
replace(X,Y,[X|T],[Y|T]).
replace(X,Y,[H|T],[H|T1]):-replace(X,Y,T,T1).
```

Output:

```
Goal: replace(1,2[1,4,5,5,6],X)
X=[2,4,5,5,6]
1 solution
```

5: PROGRAM TO DELETE AN INTEGER FROM THE LIST .

```
domains
list=integer*
predicates
del(integer,list,list)
clauses
del(X,[X|T],T).
del(X,[H|T],[H|T1]):-
del(X,T,T1).
```

Output:

```
Goal:del(3,[2,3,4],X)
X=[2,4]
1 solution
```

6: PROGRAM TO SHOW CONCEPT OF LIST.

```
domains
name=symbol*
predicates
itnames(name)
clauses
itnames([ram,kapil,shweta]).
itnames([ram,shweta,kapil]).
```

Output:

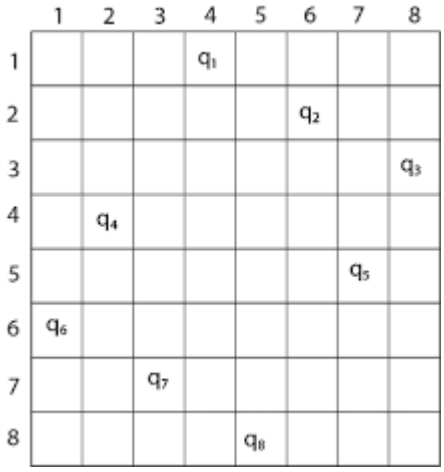
```
Goal: itnames(Y)
Y=[“ram,”kapil”,”shweta”]
Y=[“ram”,”shweta”,”kapil”]
2 solutions
```

Goal: itnames([ram|T])
T=[“kapil”,”shweta”]
T=[“shweta”,”kapil”]
2 solutions

PROGRAM 4: 8 – QUEENS PROBLEM

Procedure:
The objective of the puzzle is to place eight queens on an 8x8 chessboard in such a way that no two queens threaten each other, i.e., no two queens share the same row, column, or diagonal.

Solution:
One possible solution to the 8 Queens Problem is as follows:



Each row contains exactly one queen, and each column contains exactly one queen. Additionally, there are no queens threatening each other diagonally.

Solving the 8 Queens Problem programmatically can be done using various algorithms, including backtracking, depth-first search, genetic algorithms, etc. Backtracking is one of the commonly used methods to solve this problem efficiently.

The 8 Queens Problem is a specific case of the N-Queens Problem, where the objective is to place N queens on an NxN chessboard without any queen attacking another. The difficulty of the problem increases as N increases. For larger N, finding solutions might become computationally expensive and time-consuming

Algorithm:
1. Represent the board position a 8*8 vector i.e., [1,2,3,4,5,6,7,8]. Store the set of queens in the list ‘Q’
2. Calculate the permutation of the above eight numbers stored in set P.
3. Let the position where the first queen to be placed be (1,Y), for second be (2,Y1) and store the positions in Q
4. Check for the safety of queens through the predicate ‘nonattack’
5. Calculate Y1-Y and Y -Y1. If both are not equal to X dist, which is the X distance between the first queen and others then go to step 6 else go to step 7.
6. Increment X dist by 1
7. Repeat above step for the rest of the queens until the end of the list is reached.
8. Print Q as answer
9. Exit

PROGRAM

```
DOMAINS
cell=c(integer,integer)
list=cell*
int_list=integer*
PREDICATES
solution(list)
member(integer,int_list)
nonattack(cell,list)

CLAUSES
solution([]).
solution([c(X,Y)|Others]):-
solution(Others),
member(Y,[1,2,3,4,5,6,7,8]),
nonattack(c(X,Y),Others).
nonattack(_,[]).
nonattack(c(X,Y),[c(X1,Y1)|Others]):-
Y<>Y1,
Y1-Y<>X1-X,
Y1-Y<>X-X1,
nonattack(c(X,Y),Others).
member(X,[X|_]).
member(X,[_|Z]):-
member(X,Z).

GOAL
solution([c(1,A),c(2,B),c(3,C),c(4,D),c(5,E),c(6,F),c(7,G),c(8,H)]).
```

OUTPUT

A=4, B=2, C=7, D=3, E=6, F=8, G=5, H=1
A=5, B=2, C=4, D=7, E=3, F=8, G=6, H=1
A=3, B=5, C=2, D=8, E=6, F=4, G=7, H=1
A=3, B=6, C=4, D=2, E=8, F=5, G=7, H=1
A=5, B=7, C=1, D=3, E=8, F=6, G=4, H=2
A=4, B=6, C=8, D=3, E=1, F=7, G=5, H=2
A=3, B=6, C=8, D=1, E=4, F=7, G=5, H=2
A=5, B=3, C=8, D=4, E=7, F=1, G=6, H=2
A=5, B=7, C=4, D=1, E=3, F=8, G=6, H=2
A=4, B=1, C=5, D=8, E=6, F=3, G=7, H=2
A=3, B=6, C=4, D=1, E=8, F=5, G=7, H=2
A=4, B=7, C=5, D=3, E=1, F=6, G=8, H=2
A=6, B=4, C=2, D=8, E=5, F=7, G=1, H=3
A=6, B=4, C=7, D=1, E=8, F=2, G=5, H=3
A=1, B=7, C=4, D=6, E=8, F=2, G=5, H=3
A=6, B=8, C=2, D=4, E=1, F=7, G=5, H=3
A=6, B=2, C=7, D=1, E=4, F=8, G=5, H=3

A=4, B=7, C=1, D=8, E=5, F=2, G=6, H=3
A=5, B=8, C=4, D=1, E=7, F=2, G=6, H=3
A=4, B=8, C=1, D=5, E=7, F=2, G=6, H=3
A=2, B=7, C=5, D=8, E=1, F=4, G=6, H=3
A=1, B=7, C=5, D=8, E=2, F=4, G=6, H=3
A=2, B=5, C=7, D=4, E=1, F=8, G=6, H=3
A=4, B=2, C=7, D=5, E=1, F=8, G=6, H=3
A=5, B=7, C=1, D=4, E=2, F=8, G=6, H=3
A=6, B=4, C=1, D=5, E=8, F=2, G=7, H=3
A=5, B=1, C=4, D=6, E=8, F=2, G=7, H=3
A=5, B=2, C=6, D=1, E=7, F=4, G=8, H=3
A=6, B=3, C=7, D=2, E=8, F=5, G=1, H=4
A=2, B=7, C=3, D=6, E=8, F=5, G=1, H=4
A=7, B=3, C=1, D=6, E=8, F=5, G=2, H=4
A=5, B=1, C=8, D=6, E=3, F=7, G=2, H=4
A=1, B=5, C=8, D=6, E=3, F=7, G=2, H=4
A=3, B=6, C=8, D=1, E=5, F=7, G=2, H=4
A=6, B=3, C=1, D=7, E=5, F=8, G=2, H=4
A=7, B=5, C=3, D=1, E=6, F=8, G=2, H=4
A=7, B=3, C=8, D=2, E=5, F=1, G=6, H=4
A=5, B=3, C=1, D=7, E=2, F=8, G=6, H=4…
92 solutions

PROGRAM 5.DEPTH FIRST SEARCH

Procedure:
Depth-First Search (DFS) is used to find the shortest path in a graph because it explores paths as deeply as possible before backtracking.

Algorithm
1. Represent the positions using the conditions of childnodes
2. Calculate the permutation of the above child nodes using the path function
3. Let the path first to be traversed (a,e,L) and store the positions in path
4. Check the shortest distance of the goal node.
5. Repeat the above process and find the shortest distance to the goal node by path traversing
6. Exit

PROGRAM

```
domains
X=symbol
Y=symbol*

predicates
child(X,X)
childnode(X,X,Y)
path(X,X,Y)

clauses
child(a,b). /*b is child of a*/
child(a,c). /*c is child of a*/
child(a,d). /*d is child of a*/
child(b,e). /*b is child of b*/
child(b,f). /*f is child of b*/
child(c,g). /*g is child of c*/
path(A,G,[A|Z]):- /*to find the path from root to leaf*/
childnode(A,G,Z).
childnode(A,G,[G]):- /*to determine whether a node is child of other*/
child(A,G). childnode(A,G,[X|L]):- child(A,X), childnode(X,G,L).

goal:-path(a,e,L).
L=[“a”,”b”,”e”]
```

OUTPUT

```
Goal: path(a,b,L).
L=[“a”,”b”,”c”]
1 Solution
```

PROGRAM 6.TRAVELLING SALESMAN PROBLEM

Procedure:

The Traveling Salesman Problem (TSP) is a classic optimization problem in the field of computer science and operations research. It is a combinatorial optimization problem that seeks to find the shortest possible route that visits a set of given cities exactly once and returns to the starting city. The problem is known to be NP-hard, which means there is no known efficient algorithm to find the exact optimal solution for large instances of the problem**.**

Algorithm

1. Represent the routes using road
2. Calculate the permutation of road using nondeterm function
3. Let the first route to be traversed be route (“tampa”, “kansas_city”,x)
4. Check the shortest distance of the goal node
5. Repeat the above process and find the shortest cost of the path by traversing
6. Exit

PROGRAM

```
route(Town1,Town2,Distance)
 road(Town1,Town2,Distance).
 route(Town1,Town2,Distance)
road(Town1,X,Dist1),
 route(X,Town2,Dist2),
 Distance=Dist1+Dist2,
domains
 town = symbol
 distance = integer
predicates
 nondeterm road(town,town,distance)
 nondeterm route(town,town,distance)
clauses
 road("tampa","houston",200).
 road("gordon","tampa",300).
 road("houston","gordon",100).
 road("houston","kansas_city",120).
 road("gordon","kansas_city",130).
route(Town1,Town2,Distance):
 road(Town1,Town2,Distance).
 route(Town1,Town2,Distance):
 road(Town1,X,Dist1),
 route(X,Town2,Dist2),
 Distance=Dist1+Dist2,
 !.
goal
 route("tampa", "kansas_city", X),
 write("Distance from Tampa to Kansas City is ",X),nl.
```

OUTPUT

```
Distance from Tampa to Kansas City is 320
X=320
    1 Solution
```

PROGRAM7:8 PUZZLE PROGRAM

Procedure:

The 8 Puzzle (also known as the 8-Puzzle or 8-Puzzle Problem) is a classic sliding puzzle that consists of a 3x3 grid with eight numbered tiles and one empty tile (usually represented by the number 0). The objective of the puzzle is to rearrange the tiles from their initial state to a goal state by sliding them one at a time into the empty space.

The initial state and goal state of the 8 Puzzle can look like this:

Initial State:
1 2 3
4 0 5
6 7 8
Goal State:

1 2 3
4 5 6
7 8 0
The rules for sliding the tiles are as follows:

A tile can slide into the empty space if it is adjacent to it (vertically or horizontally).
1.The blank tile can only be moved to a position that is adjacent to it.
2.The 8 Puzzle is an example of a search problem, and it is often solved using search algorithms such as Breadth-First Search (BFS), Depth-First Search (DFS), A* Search, or Iterative Deepening A* (IDA*).

Algorithm

1. Get an integer INT
2. Get an ip of list 3*3 in PC(current positions) and the total positions in the list are stored in LPL.
3. By using move_left, move_right, move_up, move_down actions arrange the elements in list
4. Switch the list using BFS Strategy
5. Cut the repeating element positions with loops
6. After finding the optimal path
7. Print the Goal

PROGRAM

```
DOMAINS
INT=integer
PL=INT*
LPL=PL*
PREDICATES
puzzle(PL,LPL,INT,INT,INT)
BFS(PL,PL,LPL,LPL,INT,INT,INT)
move(PL,PL)
move_left(PL,PL)
move_right(PL,PL)
move_up(PL,PL)
move_down(PL,PL)
member(PL,LPL)
writel(PL)
out(LPL)
invord(LPL,LPL,LPL)
```

```
writepuz(LPL,INT)
CLAUSES
puzzle(L1,L2,I,N,_):-I<N,
L1=[0,1,2,3,4,5,6,7,8],out(L2),!.
puzzle(L1,L2,I,N,M):-I<N,K=I+1,BFS(L1,L3,L2,L4,K,J,M),puzzle(L3,L4,J,N,M),!.
BFS(L1,L2,L3,L4,I,J,M):-move(L1,L2),
not(member(L2,L3)),
L4=[L2|L3],
S=M*trunc(I/M),not(S=I),J=I.

move(L1,L2):-move_left(L1,L2).
move(L1,L2):-move_right(L1,L2).
move(L1,L2):-move_up(L1,L2).
move(L1,L2):-move_down(L1,L2).
move_left([0,A,B|T],[A,0,B|T]):-!.
move_left([A,0,B|T],[A,B,0|T]):-!.
move_left([A,B,C|T],[A,B,C|T1]):-!,move_left(T,T1),!.
move_right([A,B,0|T],[A,0,B|T]):-!.
move_right([A,0,B|T],[0,A,B|T]):-!.
move_right([A,B,C|T],[A,B,C|T1]):-!,move_right(T,T1),!.
move_up([0,A,B,C|T],[C,A,B,0|T]):-!.

move_up([H|T],[H|T1]):-!,move_up(T,T1),!.
move_down([A,B,C,0|T],[0,B,C,A|T]):-!.
move_down([H|T],[H|T1]):-!,move_down(T,T1),!.
out(L):-invord(L,[],L1),writepuz(L1,0),!.
writepuz([],_):-!.
writepuz([H|T],N):-!,N1=N+1,write("Step#",nl),writel(H),nl,writepuz(T,N1),!.
writel([]):-!.
writel([A,B,C|T]):-!,write(A,B,C),nl,writel(T),!.
member(M,[M|_]):-!.
member(M,[_|T]):-!,member(M,T),!.
invord([],L,L):-!.
invord([H|T],T1,L):-!,invord(T,[H|T1],L),!.
```

OUTPUT

```
Goal: L=[1,0,2,3,4,5,6,7,8], puzzle(L,[L],0,100,20).
Step#nl
102
345
678
Step#nl
012
345
678
L=[1,0,2,3,4,5,6,7,8]
1 Solution
```

PROGRAM 8:MONKEY BANANA PROBLEM

Procdeure

Problem Setup:
There is a room with a banana hanging from the ceiling.
There is a chair in the room.
There is a stick in the room.
Rules:
The monkey can move around freely in the room.
The monkey can climb onto the chair to reach higher positions.
The monkey can use the stick to hit the bananas and make them fall.
Objective:
The monkey must figure out how to use the chair and stick to reach the bananas and obtain them.

Algorithm:

1. Monkey can reach the chair if both of them are at same level. From the given code we can see both monkey and chair on the same level.
2. If the chair position is not at the center then monkey can drag it to center
3. If the monkey and chair both are on the floor the chair is at center, then the monkey can cimb upto chair. So the vertical position of the monkey can be changed.
4. When the monkey is on the chair and chair is at center then the monkey hit the banana with the sticks and get it

PROGRAM

```
domains
X=string
predicates
take(X,X)
move(X,X)
get_on(X,X)
hit(X,X,X)
go clauses
go:-
take("Monkey","Stick"),
move("Monkey","Chair"),
get_on("Monkey","Chair"),
hit("Monkey","Stick","Banana"),
write("The monkey has hit the banana").
go:-
write("The monkey couldn't reach the banana").
take(Animal,Object):-
write("Does the ",Animal," take the ",Object,"?(y/n)"),
readchar(Reply),
Reply='y'.
move(Animal,Object):-
write("Does the ",Animal," move the ",Object,"?(y/n)"),
readchar(Reply),
Reply='y'.
```

```
get_on(Animal,Object):-
write("Does the ",Animal," get on ",Object,"?(y/n)"),
readchar(Reply),
Reply='y'.
hit(Animal,Object,Fruit):-
write("Does the ",Animal," hit the ",Fruit,"with the",Object,"?(y/n)"),
readchar(Reply),
Reply='y'.
```

OUTPUT

```
Goal:go
Does the Monkey take the Stick?(y/n)
y
Does the Monkey move the Chair?(y/n)

y
Does the Monkey get on Chair?(y/n)
y
Does the Monkey hit the Banana with the Stick?(y/n)
y
```

PROGRAM 9: WATER JUG PROBLEM

Procedure

In the water jug problem we are provided with 2 jugs: one has the capacity to hold 3 gallons of water and the other has the capacity to hold 4 gallons of water. There is no other measuring equipment available and the jugs also do not have any kind of marking on them. So, the agent's task here is to fill the 4-gallon jug with 2 gallons of water by using only these two jugs and no other material

Initially both jugs are empty

S.No	Initial State	Condition	Final State	Description of action taken
1	(x,y)	If x<4	(4,y)	Fill the 4-gallon jug completely
2	(x,y)	If y<3	(x,3)	Fill the 3-gallon jug completely
3	(x,y)	If x>0	(x-d,y)	Pour some part from the 4-gallon jug
4	(x,y)	If y>0	(x,y-d)	Pour some part from the 3-gallon jug
5	(x,y)	If x>0	(0,y)	Empty the 4-gallon jug
6	(x,y)	If x>0	(x,0)	Empty the 3-gallon jug
7	(x,y)	If (x+y)<7	(4,y-[4-x])	Pour some water from the 3-gallon jug to fill the 4-gallon jug
8	(x,y)	If (x+y)<7	(x-[3-y],y)	Pour some water from the 4-gallon jug to fill the 3-gallon jug
9	(x,y)	If (x+y)<4	(x+y,0)	Pour some water from the 3-gallon jug to fill the 4-gallon jug
10	(x,y)	If (x+y)<3	(0,x+y)	Pour some water from the 4-gallon jug to fill the 3-gallon jug

Program:

```
database
 rstate(integer,integer)
predicates
 state(integer,integer)
clauses
 state(2,_).
 state(0,0):-
 not(rstate(0,0)),
 assert(rstate(0,0)),
 state(0,0).
state(X,Y):-
 X < 4,
 not(rstate(4,Y)),
 assert(rstate(4,Y)),
 write("\n Rule 1 => (4,",Y,")"),
 state(4,Y).
state(X,Y):-
 Y < 3,
 not(rstate(X,3)),
 assert(rstate(X,3)),
 write("\n Rule 2 => (",X,",3)"),
```

```
state(X,3).
state(X,Y):-
X>0,
not(rstate(0,Y)),
assert(rstate(0,Y)),
write("\n Rule 5 => (0,",Y,")"),
state(0,Y).
state(X,Y):-
Y>0,
not(rstate(X,0)),
assert(rstate(X,0)),
write("\n Rule 6 => (",X,",0)"),
state(X,0).
state(X,Y):-
X+Y >= 4,
Y > 0,
Z=Y-(4-X),
not(rstate(4,Z)),
assert(rstate(4,Z)),
write("\n Rule 7 => (4,",Z,")"),
state(4,Z).
state(X,Y):-
X+Y >= 3,
X>0,
Z=X-(3-Y),
not(rstate(Z,3)),
assert(rstate(Z,3)),
write("\n Rule 8 => (",Z,",3)"),
state(Z,3).
state(X,Y):-
X+Y <= 4,
Y > 0,
Z=X+Y,
not(rstate(Z,0)),
assert(rstate(Z,0)),
write("\n Rule 9 => (",Z,",0)"),
state(Z,0).
state(X,Y):-
X+Y <= 3,
X>0,
Z=X+Y,
not(rstate(0,Z)),
assert(rstate(0,Z)),
write("\n Rule 10 => (0,",Z,")"),
state(0,Z).
state(X,Y):-
X=0,
Y=2,
not(rstate(2,0)),
assert(rstate(2,0)),
write("\n Rule 11 => (2,0)"),
state(2,0).
state(X,Y):-
X=2,
assert(rstate(0,Y)),
```

```
 write("\n Rule 12 => (0,",Y,")"),
 state(0,Y).
```

Output:

Goal:
write(“initial state(0,0)”),state(0,0).
Initial state(0,0).
Rule 1 =>(4,0)
Rule 2 =>(4,3)
Rule 5 =>(0,3)
Rule 9 =>(3,0)
Rule 2 =>(3,3)
Rule 7 =>(4,2)
Rule 5 =>(0,2)
Rule 9 =>(2,0) Yes.
Goal:
write(“initial state(0,0)”),state(1,3).
Initial state(0,0).
Rule 6 =>(1,0)
Rule 2 =>(1,3)
Rule 10 =>(0,1)
Rule 1 =>(4,1)
Rule 8 =>(2,3) Yes.
Goal:
write(“initial state(0,0)”),state(2,3).
Initial state(0,0). Yes

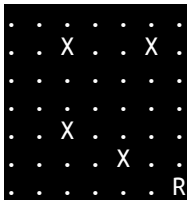
PROGRAM 10:MEANS-END ANALYSIS

Procedure:

Means-End Analysis is a problem-solving technique used in artificial intelligence and cognitive psychology. It involves breaking down a problem into subgoals and finding the means (actions or steps) to achieve those subgoals, eventually leading to the overall goal. Let's apply Means-End Analysis to a simple robot traversal problem.

Problem:
Imagine a robot is placed in a grid-like environment and needs to reach a specific destination cell while avoiding obstacles. The robot can move up, down, left, or right, but it cannot move diagonally.

Environment:



(R represents the robot's starting position, X represents obstacles, and the destination cell is notated by a dot.)

Objective:
The robot must find a sequence of actions (moves) to reach the destination cell (marked as '.'). The robot cannot move through the obstacles (marked as 'X').

Means-End Analysis Solution:

1.Identify the overall goal: The robot needs to reach the destination cell.
2.Identify the subgoals:
- a. Move closer to the destination horizontally (left or right).
- b. Move closer to the destination vertically (up or down).

3.Find the means (actions) to achieve each subgoal:
- a. To achieve subgoal (a), the robot needs to:
  Check if the destination cell is to the left or right of its current position.
  Move left or right accordingly until it is horizontally aligned with the destination cell.
- b. To achieve subgoal (b), the robot needs to:
  Check if the destination cell is above or below its current position.
  Move up or down accordingly until it is vertically aligned with the destination cell.

4.Plan the sequence of actions:
Use the means (actions) found for subgoals (a) and (b) to plan the robot's moves step-by-step, alternating between horizontal and vertical movements, until the robot reaches the destination cell.

Program:

```
% Define the environment as a 7x7 grid.
% 'r' represents the robot's starting position.
% 'x' represents obstacles.
% '.' represents empty cells.
% 'd' represents the destination cell.
```

```
environment([
    [x, x, x, x, x, x, x],
    [x, ., ., ., ., x, x],
    [x, ., x, ., x, ., x],
    [x, ., ., ., ., ., x],
    [x, ., x, ., ., x, x],
    [x, ., ., ., x, ., x],
    [x, ., ., ., ., ., r]
]).

% Define the robot's movements.
% The robot can move up, down, left, or right.
move(up, X, Y, NewX, Y) :- NewX is X - 1.
move(down, X, Y, NewX, Y) :- NewX is X + 1.
move(left, X, Y, X, NewY) :- NewY is Y - 1.
move(right, X, Y, X, NewY) :- NewY is Y + 1.

% Define the predicate to check if a cell is empty (not an obstacle).
isEmpty(Cell) :- Cell = '.' ; Cell = 'r' ; Cell = 'd'.

% Define the main predicate to find a path to the destination using Means-End Analysis.
find_path(X, Y, Path) :- find_path(X, Y, [], Path).

% Base case: Reached the destination cell (d).
find_path(X, Y, _, [(X,Y)]) :- environment(Grid), nth1(X, Grid, Row), nth1(Y, Row, 'd').

% Recursive case: Move towards the destination.
find_path(X, Y, Visited, Path) :-
    environment(Grid),
    nth1(X, Grid, Row),
    nth1(Y, Row, Cell),
    isEmpty(Cell),
    \+ member((X,Y), Visited), % Avoid revisiting cells.
    move(Direction, X, Y, NewX, NewY),
    find_path(NewX, NewY, [(X,Y) | Visited], PathRest),
    Path = [(X,Y) | PathRest].

% Predicate to display the path.
display_path([]).
display_path([(X,Y)|T]) :- format("Move to cell (~w, ~w)~n", [X, Y]), display_path(T).
```

OUTPUT

```
?- find_path(7, 7, Path), display_path(Path).
Move to cell(7, 6)
Move to cell(6, 6)
Move to cell(6, 5)
Move to cell(6, 4)
Move to cell(5, 4)
Move to cell(5, 3)
```

Move to cell(4, 3)
Move to cell(4, 2)
Move to cell(3, 2)
Move to cell(2, 2)
Move to cell(1, 2)
Move to cell(1, 3)
Move to cell(1, 4)
Move to cell(1, 5)
Move to cell(2, 5)
Move to cell(2, 6)
Move to cell(2, 7)
Move to cell(3, 7)
This output represents the sequence of moves the robot should take to reach the destination ('d') while avoiding obstacles ('x'). The numbers indicate the row and column coordinates of each cell. The robot follows this sequence of moves to reach its goal.
