

BRCM CET, BAHAL



LAB MANUAL

INTELLIGENT SYSTEM LAB (CSE 308 F)

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

Check list for Lab Manual

S. No.	Particulars	Page Number
1	Mission and Vision	3
2	Guidelines for the student	4
3	List of Programs as per University	5
4	Practical Beyond Syllabus	6
5	Sample copy of File	7 – 26

Mission

- To develop BRCM College of Engineering & Technology into a “Center of Excellence” By :
- Providing State – of – the art Laboratories, Workshops, Research and instructional facilities
- Encouraging students to delve into technical pursuits beyond their academic curriculum.
- Facilitating Post – graduate teaching and research
- Creating an environment for complete personality development of students.
- Assisting in the best possible placement

Vision

To Nurture and Harness talent for empowerment towards self actualization in all technical domains – both existing for the future



Guidelines for the Students :

1. Students should be regular and come prepared for the lab practice.
2. In case a student misses a class, it is his/her responsibility to complete that missed experiment(s).
3. Students should bring the observation book, lab journal and lab manual. Prescribed textbook and class notes can be kept ready for reference if required.
4. They should implement the given Program individually.
5. While conducting the experiments students should see that their programs would meet the following criteria:
 - Programs should be interactive with appropriate prompt messages, error messages if any, and descriptive messages for outputs.
 - Programs should perform input validation (Data type, range error, etc.) and give appropriate error messages and suggest corrective actions.
 - Comments should be used to give the statement of the problem and every function should indicate the purpose of the function, inputs and outputs
 - Statements within the program should be properly indented
 - Use meaningful names for variables and functions.
 - Make use of Constants and type definitions wherever needed.
6. Once the experiment(s) get executed, they should show the program and results to the instructors and copy the same in their observation book.
7. Questions for lab tests and exam need not necessarily be limited to the questions in the manual, but could involve some variations and / or combinations of the questions.

LIST OF PROGRAMS(University Syllabus)	
Intelligent System Lab (CSE 308 F)	
Semester : VI CSE	
S.NO	PROGRAM
1	Study of PROLOG. Write the following programs using 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 Reference :

- 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.,
- Introduction to prolog.

Practical beyond Syllabus

P.No.	Program
1.	Program to add two numbers.
2.	Program to categorize animal characteristics.
3.	Program to read address of a person using compound variable.
4.	Program of fun to show concept of cut operator .
5.	Program to count number of elements in a list .
6.	Program to reverse the list.
7.	Program to append an integer into the list .
8.	Program to replace an integer from the list .
9.	Program to delete an integer from the list .
10.	Program to show concept of list.
11.	Program to demonstrate family relationship.
12.	Program to show how integer variable is used in prolog program .

Sample copy of file

Program 1 : Study of PROLOG. Write the following programs using 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:-

```
Goal: like(sita,ram)
Yes
Goal: like(a,X)
X=b
1 Solution
Goal: hate(c,X)
X=d
1 Solution
Goal: like(x,_)
Yes
Goal: _
```

PROGRAM 2 : Steps for 8-queen problem

STEP 1 : Represent the board positions as 8*8 vector , i.e., [1,2,3,4,5,6,7,8]. Store the set of queens in the list 'Q'.

STEP 2 : Calculate the permutation of the above eight numbers stored in set P.

STEP 3 : Let the position where the first queen to be placed be (1,Y), for second be (2,Y1) and so on and store the positions in Q.

STEP 4 : Check for the safety of the queens through the predicate , 'noattack ()'.

STEP 5 : Calculate Y1-Y and Y-Y1. If both are not equal to Xdist , which is the X – distance between the first queen and others, then go to Step 6 else go to Step 7.

STEP 6 : Increment Xdist by 1.

STEP 7 : Repeat above for the rest of the queens , until the end of the list is reached .

STEP 8 : Print Q as answer .

STEP 9 : Exit.

Write a program for 8-queen problem

domains

H=integer

T=integer*

predicates

safe(T)

solution(T)

permutation(T,T)

del(H,T,T)

noattack(H,T,H)

clauses

del(I,[I|L],L). /*to take a position from the permutation of list*/

del(I,[F|L],[F|L1]):-

del(I,L,L1).

permutation([],[]). /*to find the possible positions*/

permutation([H|T],PL):-

permutation(T,PT),\

del(H,PL,PT).

solution(Q):- /*final solution is stored in Q*/


```

permutation([1,2,3,4,5,6,7,8],Q),
safe(Q).
safe([]). /*Q is safe such that no queens attack each other*/
safe([Q|others]):-
safe(others),
noattack(Q,others,1).
noattack(_,[],_). /*to find if the queens are in same row, column or
diagonal*/
noattack(Y,[Y1|Ydist],Xdist):-
Y1-Y<>Xdist,
Y-Y1<>Xdist,
dist1=Xdist,
noattack(Y,Ydist,dist1).

```

Output

goal: -solution(Q).

Q=["3","8","4","7","1","6","2","5"]

PROGRAM 3 : Write a program of depth first search

```

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"]

```

PROGRAM 4 : Solve any problem using best first search

In both breadth-first search and depth-first search, the purpose of search is to find a goal state from the initial state as quickly as possible. Since the nodes in the queue are examined from the front, the fastest solution occurs when the goal node very quickly finds its way to the front of the queue. If this happens during Depth or Breadth first search, it will be a matter of luck rather than design. To arrange for the goal node to be examined as soon as possible, we can add the successors to the queue and then sort it with the aim of placing the best successor, i.e. the successor that will lead soonest to the goal, at the front of the queue. This leads to a generalisation of both Depth-First and Breadth-First search that consists of arranging candidate nodes into a priority queue instead of a FIFO queue. This means that nodes are ordered according to some comparison operation, and the node with the highest priority or lowest cost is selected. In the AI literature, priority first search is known as best first search.

```
pfs(Origin, Visited) :- pfs3([Origin], % INITIAL PRIORITY QUEUE
[], % LIST OF NODES VISITED SO FAR
RevVisited), % SOLUTION LIST OF NODES
reverse(RevVisited, Visited).
pfs3([Node|_], History, [Node | History]) :- goal(Node).
pfs3([Node|RestQ], History, RevVisited) :-
not goal(Node),
findall(NextNode,
(successor(Node, NextNode),
not member(NextNode, History),
not member(NextNode, RestQ)),
Successors), %LIST OF SUCCESSORS OF Node
addPriorityQ(RestQ, Successors, PriorityQ), %MAKE NEW PRIORITY QUEUE
pfs3(PriorityQ, [Node | History], RevVisited).
addPriorityQ(L1, Q1, Q2) :- append(L1, Q1, L2),
sortQ(L2, Q2).
lessthan(STATE1, STATE2) :- %STATE1 IS BETTER THAN STATE2
... %COMPLETE ACCORDING TO PROBLEM
After each swap, the new list is closer to a sorted list.
bubblesort(List, Sorted):-
swap(List, List1), !,
bubblesort(List1,Sorted).
bubblesort(Sorted, Sorted). %LIST SORTED
swap([X, Y | Rest], [Y, X | Rest]):- %SWAP FIRST PAIR
21
lessthan(X, Y).
swap([Z | Rest],[Z | Rest1]):- %SWAP PAIR IN TAIL
swap(Rest, Rest1).
Insertion sort. To sort a non-empty list,
• Sort the tail.
• Insert the head into the sorted tail at such a position that the resulting list is sorted.
insertsort([], []).
insertsort([X|T], Sorted):-
insertsort(T, SortedT), %SORT THE TAIL
```

```

insert(X, SortedT, Sorted). %INSERT X
insert(X, [Y|Sorted], [Y|Sorted1]):-
lessthan(X,Y), !,
insert(X, Sorted, Sorted1).
insert(X, Sorted, [X|Sorted]).

```

PROGRAM 5 : Write a program to solve 8-Puzzle problem.

The title of this section refers to a familiar and popular sliding tile puzzle that has been around for at least forty years. The most frequent older versions of this puzzle have numbers or letters on the sliding tiles, and the player is supposed to slide tiles into new positions in order to realign a scrambled puzzle back into a goal alignment. For illustration, we use the 3 x 3 8-tile version, which is depicted here in goal configuration

7	2	3
4	6	5
1	8	

To represent these puzzle "states" we will use a Prolog term representation employing '/' as a separator. The positions of the tiles are listed (separated by '/') from top to bottom, and from left to right. Use "0" to represent the empty tile (space). For example, the goal is ...
goal(1/2/3/8/0/4/7/6/5).

Production Rules :-

$h_function(Puzz,H) \rightarrow p_fcn(Puzz,P), s_fcn(Puzz,S), H \text{ is } P + 3*S.$

The 'move' productions are defined as follows.

$move(P,C,left) \rightarrow left(P,C).$

$move(P,C,up) \rightarrow up(P,C).$

$move(P,C,right) \rightarrow right(P,C).$

$move(P,C,down) \rightarrow down(P,C).$

$p_fcn(A/B/C/D/E/F/G/H/I, P) \rightarrow a(A,Pa), b(B,Pb), c(C,Pc),$
 $d(D,Pd), e(E,Pe), f(F,Pf),$
 $g(G,Pg), h(H,Ph), i(I,Pi),$
 $P \text{ is } Pa+Pb+Pc+Pd+Pe+Pf+Pg+Ph+Pi.$

$s_fcn(A/B/C/D/E/F/G/H/I, S) \rightarrow 1 \ s_aux(A,B,S1), s_aux(B,C,S2),$
 $s_aux(C,F,S3), s_aux(F,I,S4),$
 $s_aux(I,H,S5), s_aux(H,G,S6),$
 $s_aux(G,D,S7), s_aux(D,A,S8),$
 $s_aux(E,S9),$
 $S \text{ is } S1+S2+S3+S4+S5+S6+S7+S8+S9.$

$s_aux(0,0) \rightarrow cut$
 $s_aux(X,Y,0) :- Y \text{ is } X+1, !.$
 $s_aux(8,1,0) :- !.$

The heuristic function we use here is a combination of two other estimators: `p_fcn`, the Manhattan distance function, and `s_fcn`, the sequence function, all as explained in Nilsson (1980), which estimates how badly out-of-sequence the tiles are (around the outside).

```
h_function(Puzz,H) :- p_fcn(Puzz,P),
                      s_fcn(Puzz,S),
                      H is P + 3*S.
```

The 'move' predicate is defined as follows.

```
move(P,C,left) :- left(P,C).
move(P,C,up) :- up(P,C).
move(P,C,right) :- right(P,C).
move(P,C,down) :- down(P,C).
```

Here is the code for `p` and `s`.

```
%%% Manhattan distance
p_fcn(A/B/C/D/E/F/G/H/I, P) :-
    a(A,Pa), b(B,Pb), c(C,Pc),
    d(D,Pd), e(E,Pe), f(F,Pf),
    g(G,Pg), h(H,Ph), i(I,Pi),
    P is Pa+Pb+Pc+Pd+Pe+Pf+Pg+Ph+Pi.
```

```
a(0,0). a(1,0). a(2,1). a(3,2). a(4,3). a(5,4). a(6,3). a(7,2). a(8,1).
b(0,0). b(1,0). b(2,0). b(3,1). b(4,2). b(5,3). b(6,2). b(7,3). b(8,2).
c(0,0). c(1,2). c(2,1). c(3,0). c(4,1). c(5,2). c(6,3). c(7,4). c(8,3).
d(0,0). d(1,1). d(2,2). d(3,3). d(4,2). d(5,3). d(6,2). d(7,2). d(8,0).
e(0,0). e(1,2). e(2,1). e(3,2). e(4,1). e(5,2). e(6,1). e(7,2). e(8,1).
f(0,0). f(1,3). f(2,2). f(3,1). f(4,0). f(5,1). f(6,2). f(7,3). f(8,2).
g(0,0). g(1,2). g(2,3). g(3,4). g(4,3). g(5,2). g(6,2). g(7,0). g(8,1).
h(0,0). h(1,3). h(2,3). h(3,3). h(4,2). h(5,1). h(6,0). h(7,1). h(8,2).
i(0,0). i(1,4). i(2,3). i(3,2). i(4,1). i(5,0). i(6,1). i(7,2). i(8,3).
```

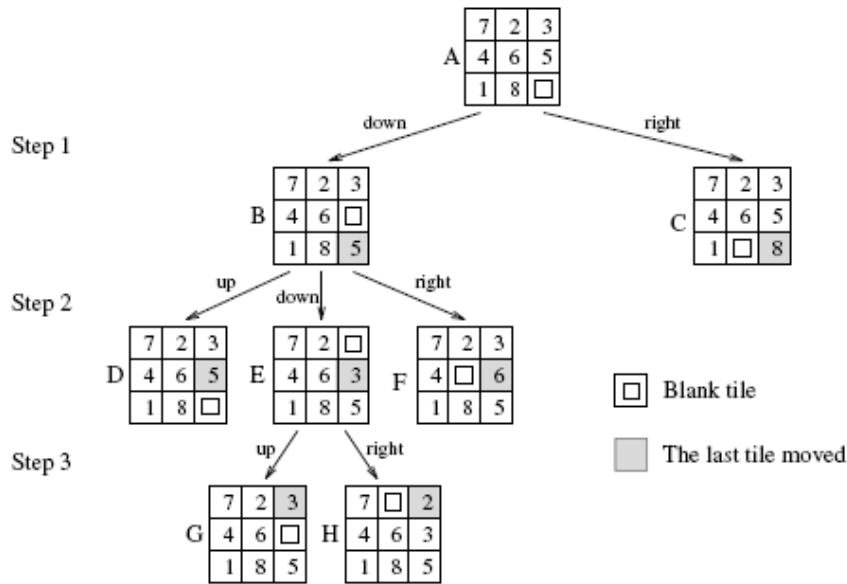
```
%%% the out-of-cycle function
s_fcn(A/B/C/D/E/F/G/H/I, S) :-
    s_aux(A,B,S1), s_aux(B,C,S2), s_aux(C,F,S3),
    s_aux(F,I,S4), s_aux(I,H,S5), s_aux(H,G,S6),
    s_aux(G,D,S7), s_aux(D,A,S8), s_aux(E,S9),
    S is S1+S2+S3+S4+S5+S6+S7+S8+S9.
```

```
s_aux(0,0) :- !.
s_aux(_,1).
```

```
s_aux(X,Y,0) :- Y is X+1, !.
s_aux(8,1,0) :- !.
s_aux(_,_,2).
```

The Prolog program from the previous section and the program outlined in this section can be used as an 8-puzzle solver.

```
?- solve(0/8/1/2/4/3/7/6/5, S).
```

Solution:

left(A/0/C/D/E/F/H/I/J , 0/A/C/D/E/F/H/I/J).
left(A/B/C/D/0/F/H/I/J , A/B/C/0/D/F/H/I/J).
left(A/B/C/D/E/F/H/0/J , A/B/C/D/E/F/0/H/I/J).
left(A/B/0/D/E/F/H/I/J , A/0/B/D/E/F/H/I/J).
left(A/B/C/D/E/0/H/I/J , A/B/C/D/0/E/H/I/J).
left(A/B/C/D/E/F/H/I/0 , A/B/C/D/E/F/H/0/I).

up(A/B/C/0/E/F/H/I/J , 0/B/C/A/E/F/H/I/J).
up(A/B/C/D/0/F/H/I/J , A/0/C/D/B/F/H/I/J).
up(A/B/C/D/E/0/H/I/J , A/B/0/D/E/C/H/I/J).
up(A/B/C/D/E/F/0/I/J , A/B/C/0/E/F/D/I/J).
up(A/B/C/D/E/F/H/0/J , A/B/C/D/0/F/H/E/I/J).
up(A/B/C/D/E/F/H/I/0 , A/B/C/D/E/0/H/I/F).

right(A/0/C/D/E/F/H/I/J , A/C/0/D/E/F/H/I/J).
right(A/B/C/D/0/F/H/I/J , A/B/C/D/F/0/H/I/J).
right(A/B/C/D/E/F/H/0/J , A/B/C/D/E/F/H/J/0).
right(0/B/C/D/E/F/H/I/J , B/0/C/D/E/F/H/I/J).
right(A/B/C/0/E/F/H/I/J , A/B/C/E/0/F/H/I/J).
right(A/B/C/D/E/F/0/I/J , A/B/C/D/E/F/I/0/J).

down(A/B/C/0/E/F/H/I/J , A/B/C/H/E/F/0/I/J).
down(A/B/C/D/0/F/H/I/J , A/B/C/D/I/F/H/0/J).
down(A/B/C/D/E/0/H/I/J , A/B/C/D/E/J/H/I/0).
down(0/B/C/D/E/F/H/I/J , D/B/C/0/E/F/H/I/J).
down(A/0/C/D/E/F/H/I/J , A/E/C/D/0/F/H/I/J).
down(A/B/0/D/E/F/H/I/J , A/B/F/D/E/0/H/I/J).

PROGRAM 6 :

Means-Ends Analysis (MEA) is a technique used in Artificial Intelligence for controlling search in problem solving computer programs. It is also a technique used at least since the 1950s as a creativity tool, most frequently mentioned in engineering books on design methods. Means-Ends Analysis is also a way to clarify one's thoughts when embarking on a mathematical proof.

Problem-solving as search

An important aspect of intelligent behavior as studied in AI is goal-based problem solving, a framework in which the solution of a problem can be described by finding a sequence of actions that lead to a desirable goal. A goal-seeking system is supposed to be connected to its outside environment by sensory channels through which it receives information about the environment and motor channels through which it acts on the environment. (The term "afferent" is used to describe "inward" sensory flows, and "efferent" is used to describe "outward" motor commands.) In addition, the system has some means of storing in a memory information about the state of the environment (afferent information) and information about actions (efferent information). Ability to attain goals depends on building up associations, simple or complex, between particular changes in states and particular actions that will bring these changes about. Search is the process of discovery and assembly of sequences of actions that will lead from a given state to a desired state. While this strategy may be appropriate for machine learning and problem solving, it is not always suggested for humans (e.g. cognitive load theory and its implications).

How MEA works

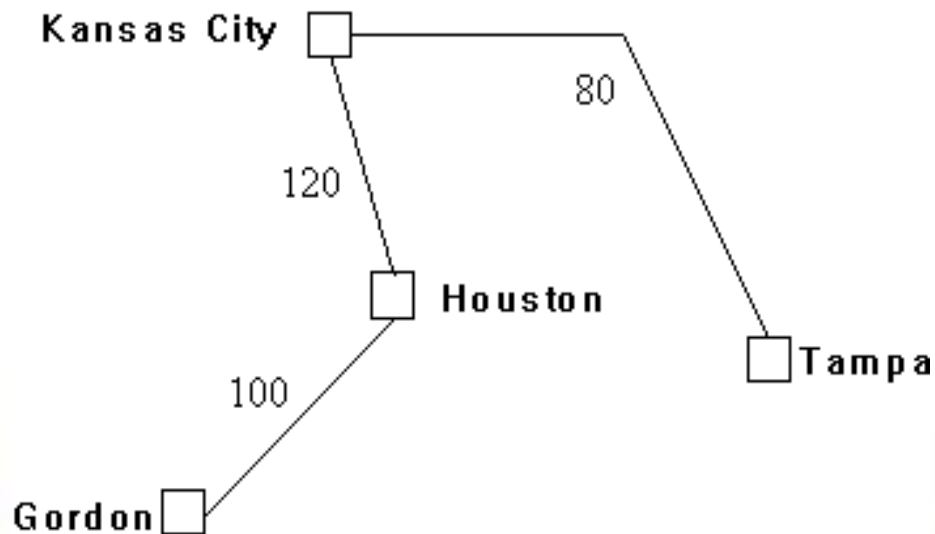
The MEA technique is a strategy to control search in problem-solving. Given a current state and a goal state, an action is chosen which will reduce the difference between the two. The action is performed on the current state to produce a new state, and the process is recursively applied to this new state and the goal state. Note that, in order for MEA to be effective, the goal-seeking system must have a means of associating to any kind of detectable difference those actions that are relevant to reducing that difference. It must also have means for detecting the progress it is making (the changes in the differences between the actual and the desired state), as some attempted sequences of actions may fail and, hence, some alternate sequences may be tried. When knowledge is available concerning the importance of differences, the most important difference is selected first to further improve the average performance of MEA over other brute-force search strategies. However, even without the ordering of differences according to importance, MEA improves over other search heuristics (again in the average case) by focusing the problem solving on the actual differences between the current state and that of the goal.

Some AI systems using MEA

The MEA technique as a problem-solving strategy was first introduced in 1963 by Allen Newell and Herbert Simon in their computer problem-solving program General Problem Solver (GPS). In that implementation, the correspondence between differences and actions, also called operators, is provided a priori as knowledge in the system. (In GPS this knowledge was in the form of table of connections.) When the action and side-effects of applying an operator are penetrable, the search may select the relevant operators by inspection of the operators and do without a table of connections. This latter case, of which the canonical example is STRIPS, an automated planning computer program, allows task-independent correlation of differences to the operators which reduce them.

PROGRAM 7 : Write a program to solve traveling salesman problem.

The following is the simplified map used for the prototype:



Production Rules:-

$\text{route}(\text{Town1}, \text{Town2}, \text{Distance}) \rightarrow \text{road}(\text{Town1}, \text{Town2}, \text{Distance}).$
 $\text{route}(\text{Town1}, \text{Town2}, \text{Distance}) \rightarrow \text{road}(\text{Town1}, \text{X}, \text{Dist1}),$
 $\text{route}(\text{X}, \text{Town2}, \text{Dist2}),$
 $\text{Distance} = \text{Dist1} + \text{Dist2},$

domains

town = symbol
 distance = integer

predicates

nondeterm $\text{road}(\text{town}, \text{town}, \text{distance})$
 nondeterm $\text{route}(\text{town}, \text{town}, \text{distance})$

clauses

$\text{road}(\text{"tampa"}, \text{"houston"}, 200).$
 $\text{road}(\text{"gordon"}, \text{"tampa"}, 300).$
 $\text{road}(\text{"houston"}, \text{"gordon"}, 100).$
 $\text{road}(\text{"houston"}, \text{"kansas_city"}, 120).$
 $\text{road}(\text{"gordon"}, \text{"kansas_city"}, 130).$
 $\text{route}(\text{Town1}, \text{Town2}, \text{Distance}) :-$
 $\text{road}(\text{Town1}, \text{Town2}, \text{Distance}).$
 $\text{route}(\text{Town1}, \text{Town2}, \text{Distance}) :-$
 $\text{road}(\text{Town1}, \text{X}, \text{Dist1}),$
 $\text{route}(\text{X}, \text{Town2}, \text{Dist2}),$
 $\text{Distance} = \text{Dist1} + \text{Dist2},$
 $!.$

goal

$\text{route}(\text{"tampa"}, \text{"kansas_city"}, \text{X}),$
 $\text{write}(\text{"Distance from Tampa to Kansas City is "}, \text{X}), \text{nl}.$

Distance from Tampa to Kansas City is 320
 X=320

Practical beyond syllabus

PROGRAM 1 : Program to add two numbers.

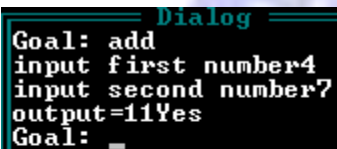
predicates

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 number4
input second number7
output=11Yes
Goal: _
```

PROGRAM 2 :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:-

```
Dialog
Goal: small(X)
X=rat
X=cat
2 Solutions
Goal: large(lion)
Yes
Goal: color(cat,brown)
No
Goal: color(rabbit,white)
Yes
Goal:
```

PROGRAM 3 : 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).
```

```
Dialog
Accept(Y/N)?Yes
Goal: go
Name:neha
Street:rampur
City:hissar
State:haryana
Zip:123566

address('neha','rampur',
'hissar','haryana','1235
66')

Accept(Y/N)?Yes
Goal: _
```

Output:-

PROGRAM 4 : Program of fun to show concept of cut operator .

predicates

fun(integer,integer)

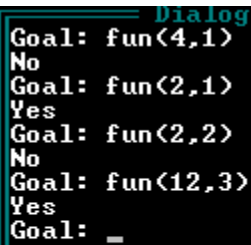
clauses

fun(Y,1):-Y<3,!,

fun(Y,2):-Y>3,Y<=10,!,

fun(Y,3):-Y>10,!,

Output:-



```
Dialog
Goal: fun(4,1)
No
Goal: fun(2,1)
Yes
Goal: fun(2,2)
No
Goal: fun(12,3)
Yes
Goal: _
```

PROGRAM 5 : Program to count number of elements in a list .

domains

x=integer

list=integer*

predicates

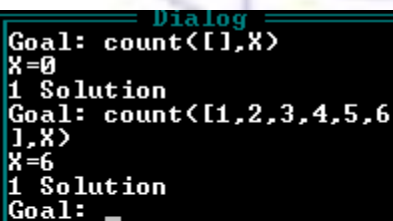
count(list,x)

clauses

count([],0).

count([_|T],N):-count(T,N1),N=N1+1.

Output:-



```
Dialog
Goal: count([],X)
X=0
1 Solution
Goal: count([1,2,3,4,5,6],X)
X=6
1 Solution
Goal: _
```

PROGRAM 6 : 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([1,2,3,4],X)  
X=[4,3,2,1]  
1 Solution  
Goal:
```

PROGRAM 7 : 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:-

```

Dialog
Goal: append(1,[2,3,4,5]
,X)
X=[2,3,4,5,1]
1 Solution
Goal:

```

PROGRAM 8: 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:-

```

Dialog
Goal: replace(1,2,[1,4,5]
,5,61,X)
X=[2,4,5,5,6]
1 Solution
Goal: replace(4,3,[2,3,4]
,5,61,X)
X=[2,3,3,5,6]
1 Solution
Goal:

```

PROGRAM 9 : 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:-


```

Dialog
Goal: del(3,[2,3,4],X)
X=[2,4]
1 Solution
Goal: del(1,[1,2,3,4,5],
X)
X=[2,3,4,5]
1 Solution
Goal:

```

PROGRAM 10 : Program to show concept of list.

domains

name=symbol*

predicates

itnames(name)

clauses

itnames([ram,kapil,shweta]).

itnames([ram,shweta,kapil]).

Output:-

```

Dialog
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
Goal:

```

PROGRAM 11 : 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:-

```
Dialog
Goal: child(X,h)
No Solution
Goal: female(b)
Yes
Goal: male(a)
Yes
Goal: grandparent(a,X)
No Solution
Goal: grandparent(a,d)
No
Goal:
```

PROGRAM 12 : 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.

```
Dialog
Goal: go
10
No
Goal: go
10
No
Goal: _
```

Output:-