Chhatrapati Shahu Maharaj Shikshan Sanstha's

CHH. SHAHU COLLEGE OF ENGINEERING

Kanchanwadi, Paithan Road, Aurangabad.

INDEX

Date : / /2023

List of Experiments in : AI

S.N.	Title	Page No.	Date	Remarks
01	Study of PROLOG. Write programs using PROLOG	1-8		
		8-		
02	Write a Program to Solve 8queen Problem.	9-12		
03	Solve any problem using depth first search	13-18		
04.	Solve any problem using best first search.	19-24		
05	Solve 8-puzzle problem using best first search.	25-29		
06	Solve Robot (traversal) problem using means End Analysis.	31-33		
07	Solve traveling Salesman problem.	34-35		

This is to Certify that, the experiments mentioned above were executed within the four walls of Institute by Vaibhav Kalyanrao Tawale [CS4256]

Lecture-in-Charge

Head of Department

Principal

Lab Assignment NO:- 01

Aim: Study of prolog.

Theory:

a) **Prolog:** Prolog as the name itself suggests is the short form of LOGical PROgramming. It is a logical and declarative programming language.

Logic Programming is one of the Computer Programmig paradigm, in which the program statements express the facts and rules about different problems within a system of Formal logic.

Here, the rules are written in the Form of logical clauses, where head and body are present. For example, H is head and B1, B2, B3 are the elements of the body. Now if we state that "H is true, when B1, B2, B2 all are true", this is a rule. On the other hand, facts are like the rules, but without any body. So, an example of fact is "H is true".

Prolog, have declarative and also imperative properties. This may also include procedural statements like "To solve the problem H, perform B1, B2. and B3",

b) Facts, Rules & Queries:-

Facts:- The fact is predicate that is t
For, example, if we say, "Tom is th
son of Jack", then this is a fact.
Syntax : relation (object1, object2...).
Rules:- Rules are extinctions of facts th
contain conditional clauses. To satisfy
a rule these conditions should be m
For example, if we define a rule as

grandFather(x,y) :- father(x,z), parent(z,

This implies that for x to be the gran
of y,z should be a parent of y and
should be father of z.

Queries or Questions:- And to run a pr
program, we need some questions,
and those questions can be answered
by the given facts and rules.

Examples:
Facts: cat(tom).
loves_to_eat(kunal, pasta).
of_color(chair, black).
lazy(Sainath).

Rules: 1) happy(lili) :- dance(lili).
2) hungry(tom) :- search_for_food(tom).
3) friends(Jack, bili) :- loves(cricket(jack), loves(cricket(bili)).
4) goToplay(ryan) :- isClosed(school), free(ryan).

Queries:
1) Does kunal love to eat pasta?
loves_to_eat_pasta(kunal, pasta).
2) Is Lili happy?
happy(Lili).

So according to these queries, Logic programming language can find the answer and return them.

C] Write a program to implement simple facts and queries

Program:-
Facts:
studies(charlie, cs135). % charlie studies cs135
studies(olivia, cs135). % Olivia studies cs135
studies(jack, cs131). % Jack studies 131
studies(arthur, cs134). % Arthur studies cs134

teaches(kirke, cs135). % kirke teaches cs135
teaches(collins, cs131). % collins teaches cs131

teaches (Collins, CS171). % Collins teaches CS171
teaches(juniper, CS134). % juniper teaches CS134

Rules:

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

% X is professor of Y if X teaches C and
studies C.

Queries :-

? - studies (charlie, What).
% What CS135

? Professor (Kirke, students).
% students = charlie;
% students = olivia.
% Who are the students of professor

D] Write a program to implement simp
arithmetic

arithmetic operator types shown belo

- Addition (+) operator
- Substraction (-) operator
- Multiplication (*) operator
- Divsion (%) operator
- Power (**) operator

- Integer division (//) operator
- Modulus (mod) operator
- Square root (sqrt) operator
- maximum (max) operator

Examples:

Addition (+) operator:

?- ADD is 45+12

ADD = 57

yes

Substraction (-) operator:

?- SUB is 45-12

SUB = 33

yes

Multiplication (*) operator:

?- MUL is 45*12.

MUL = 540

yes

Division operator (/)

?- DIV is 45/12.

DIV = 3.75

yes

Powers (**) operator:

?- POW is 12**2.

POW = 144.0

yes

Date

Integer division (//) operator:

?- IntDiv is 45//12.

IntDiv = 3

yes

Modulus (mod) operator

?- Mod is 45 mod 12

Mod = 9

yes

Square root (sqrt) operator:

?- R is 144,

S is sqrt(R).

R = 144

S = 12.0

yes

E] List Representation

A list can be either empty or non-
in first case, the list is simply wri
as a Prolog atom []. In the second
the list consists of two things
head :- first item called head
tail :- The remaining part of the list
tail.

So the following list representations
valid -

- [a, b, c] = [X | [b, c]]
- [a, b, c] = [a, b | [c]]

- [a,b,c] = [a,b,c|[]]

Operations on Lists.

1] Membership operation

list_member(X,[X|_]).
list_member(X,[_|TAIL]) :- list_member(X,TAIL).

2] Concatenation:-

list_concat([],L,L).
list_concat([X|L1],L2,[X|K3]) :-
list_concat(L1,L2,L3).

3] Insert into List.

list_delete(X,[X],[]).
list_delete(X,[X|L1],L1).
list_delete(X,[Y|L2],[Y|L1]) :-
list_delete(X,L2,L1).
list_insert(X,L,R) :- list_delete(X,R,L).

4] order operation

list_order([X,Y|Tail]) :- X =< Y, list_order([Y|Tail]).
list_order([X]).

Conclusion:-

I have studied prolog and List, Arithmetic operations, Facts, Rules and queries in Prolog.

Study of Prolog:

Program:

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

%Rule: X is professor of Y if X teaches can studies C.

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

% What does charlie study?
% studies(charlie, What).

% who are the student of professor kirke ?
%professor(kirke, Student).

parent(pam,bob).
parent(tom,bob).
parent(bob,ann).
parent(bob,pat).
parent(pat,jim).
parent(tom,liz).

% who is parent of bob ?.
%parent(Who,bob).
%parent(Who,pat).
```

Output:

```
% v:/CSMSS all/7th sem all notes/Ai notes/prolog1.pl compiled 0.00 sec, 14 clauses
?- studies(charlie, What).
What = csc135.

?- professor(kirke, Student).
Student = charlie .

?- parent(Who,bob).
Who = pam .

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

Date :

Experiment NO:- 02

Aim: Write a program to solve 8-queens problem.

Theory:-

The eight queens problem is the problem of placing eight queens on an 8x8 chessboard such that none of them attack one another.

No two are in the same row, column, or diagonal. More generally, the 'n' queens problem places 'n' queens on an nxn chessboard. There are different solutions for the problem Backtracking

Algorithm:

1] Initialization:

Initially, the chessboard is empty, and you start placing queens on it one by one.

2] Placing a Queen:-

You begin by placing the first queen in the first row and mark that position.

Then, you proceed to place the second queen in the next row while considering

the constraints to avoid threats.

3]. Constraints checking:
- After placing each queen, you che
it threatens any other queens alre
on the board. (horizontally, vertical
and diagonally).
- If the newly placed queen vio
constraints, you backtrack and try
different position for the previou
queen.

4] Backtracking:
- When a constraints is violated, you
back to the previous queen and
a different position for that q
- You continue this process until y
find a valid position for the cu
queen.

5] Exploring All posibilities:
- You repeat this process for e
queen, trying different positions
backtracking if necessary.
- You explore all possible combinati
systematically moving forward a
backward, until you find a solut
exhaust all possibilities.

6) Completing the Solution:
- If you successfully place all eight queens on the board without violating any constraints, you have found a solution to the 8-Queens problem.

7) No solution found:
- If during the process, you cannot find a valid position for a queen in a certain row without violating constraints, you backtrack further to the previous queen and explore a different position for that queen.
- If you reach a point where there are no more positions to explore for the first queen, you conclude that no solution exists for the given configuration

#Solution After implimenting Backtracking Algorithm

	0	1	2	3	4	5	6	7	8
1		Q							
2								Q	
3						Q			
4									Q
5			Q						
6					Q				
7							Q		
8				Q					

Conclusion:-

In this practical i have placed 8 queens in 8x8 matrix with using Backtracking aproach.

Eight Queen Problem

Program:-

```
:- use_module(library(clpfd)).

n_queens(N, Qs) :-
        length(Qs, N),
        Qs ins 1..N,
        safe_queens(Qs).

safe_queens([]).
safe_queens([Q|Qs]) :- no_threat(Q, Qs, 1), safe_queens(Qs).

no_threat(_, [], _).
no_threat(Q1, [Q2|Qs], D) :-
        Q1 #\= Q2,
        abs(Q1 - Q2) #\= D,
        D1 #= D + 1,
        no_threat(Q1, Qs, D1).

% Define a predicate to solve the N-Queens problem and print the solution.
solve_n_queens(N) :-
        n_queens(N, Qs),
        label(Qs),
        format('Solution for ~w-Queens: ~w~n', [N, Qs]).

% Example: solve the 8-Queens problem
:- solve_n_queens(8).
```

OUTPUT:-

```
% v:/CSMSS all/7th sem all notes/Ai notes/eight_queens.pl compiled 0.22 sec, 7 clauses
?- solve_n_queens(8).
Solution for 8-Queens: [1,5,8,6,3,7,2,4]
```

Date :

Experiment NO:-03

Aim:- Write a program to solve Water-Jug Problem using Depth-first search.

Theory:-

A search problem consists of:

- **A State Space:-** Set of all possible states where you can be.
- **A Start State:-** The state from where the search begins.
- **A Goal State:-** A Function that looks at the current state returns whether or not it is the goal state.
- **Solution** to a search problem is a sequence of actions, called the plan that transforms the start state to the goal state.

Depth first search

Depth-first search (DFS) is an algorithm for traversing or searching tree or graph data structures. The algorithm starts at the root node (selecting some arbitrary node as the root node in the case of a graph) and explores as far as possible along each branch before backtracking. It uses last-in-first-out [LIFO] strategy and hence it is implemented using a stack.

Da

Properties of depth-first (tree) search:

- Space complexity is O(bm) where b is branching factor and m is the maxi depth of the tree.
- time complexity is O(bm)
  not complete unless the state space finite and contains no loops) - we may stuck going down an infinite bran that doesn't lead to a solution.
- even if the state space is infinite contains no loops, the first solution f by depth-first search may not be shortest

The two Water jug Puzzle:-

You are on the side of the river are given a m liter jug and a n lite jug where $0 < m < n$ Both the jugs initially empty.

The jugs don't have markings to measuring smaller quantities. You have to use the jugs to measure d liters of water where $d < n$

The operations you can perform are:

1. Empty a jug.
2. Fill a jug
3. Pour water from one jug to another

Date :

Algorithm

Solution 1 (Always pour from m liter jug into n liter jug)

1. Fill the m litre jug and empty it into n liter jug.
2. Whenever the m liter jug becomes empty fill it.
3. Whenever the n liter jug becomes full empty it.
4. Repeat steps 1, 2, 3 till either n liter jug or the m liter jug contains d liters of water.

Solution 2 (Always pour from n liter jug into m liter jug)

1. Fill the n liter jug and empty it into m liter jug.
2. Whenever the n liter jug becomes empty fill it.
3. Whenever the m liter jug becomes full empty it.
4. Repeat steps 1, 2 and 3 till either n liter jug or the m liter jug contains d liters of water.

Suppose there are a 3 liter jug and a 5 liter jug to measure 4 liters water
So $m=3$, $n=5$ and $d=4$. The associated Diophantine equation will be $3m+5n=4$.

We use pair (x,y) to represent amount of water inside the 3-liter jug and 5-liter jug respectively in each pouring step.

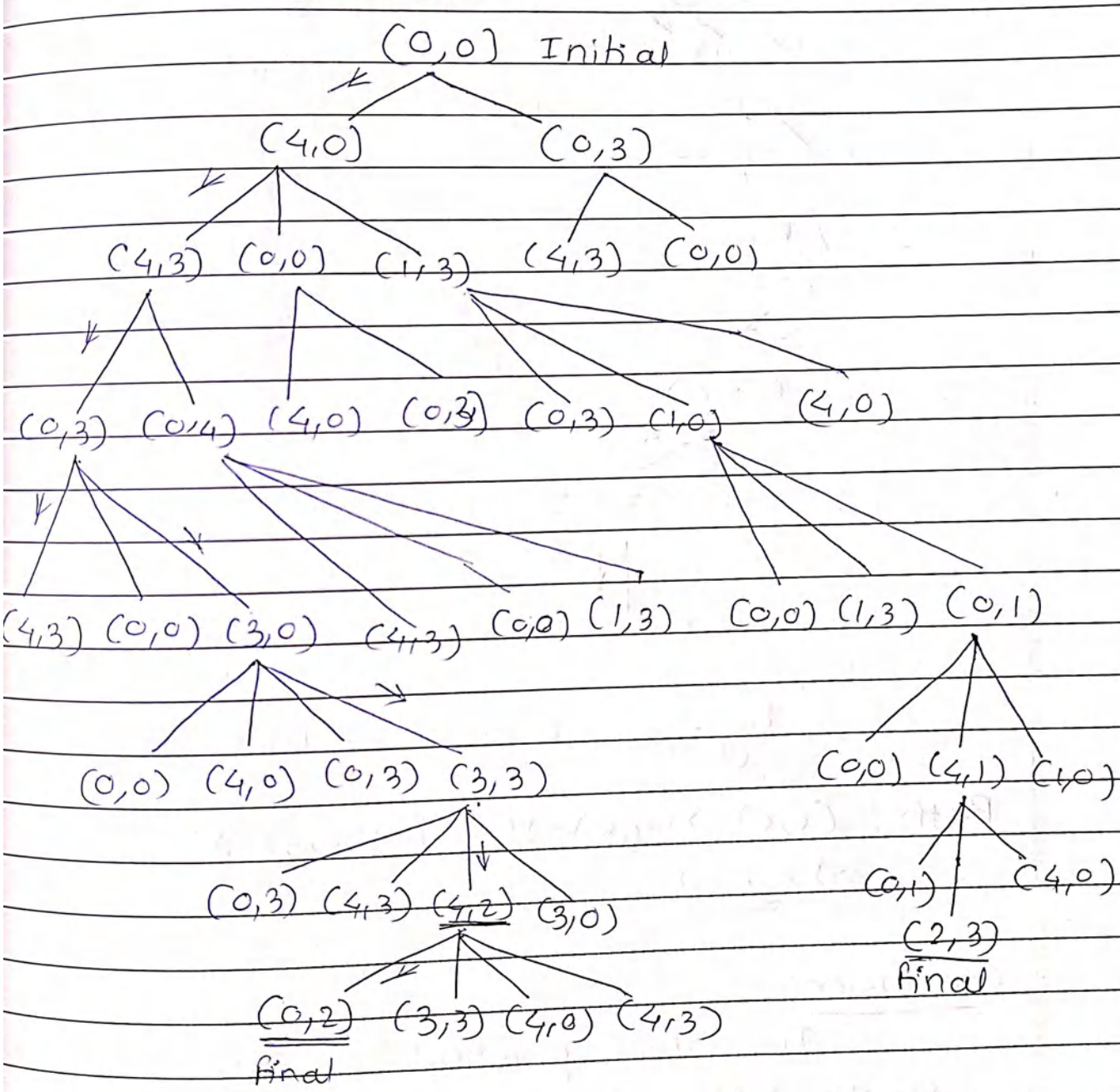
Using solution 1, successive pouring steps are

(0,0) → (3,0) → (0,3) → (3,3) → (1,5) → (1,0) → (3,1) → (0,4)

Using solution 2, successive pouring steps are

(0,0) → (0,5) → (3,2) → (0,2) → (2,0) → (2,

Depth - first Search:-



Jug 1 :- 4 l
Jug 2 :- 3 l
Goal :- (-, 2) OR (2, -)

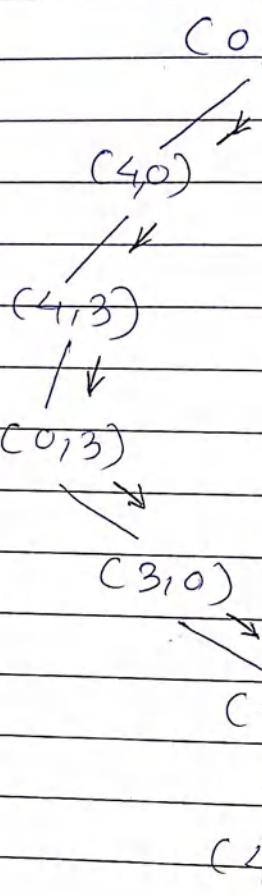


Fig. Search tree of DFS Path

Path :- (0,0) → (4,0) → (4,3) → (0,3) → (3,0) → (3,3) → (4,2)

Conclusion :-

In this practical I have studied Depth-first search and implementation of DFS on water jug problem.

Water Jug problem using DFS

Program:-

```
% Solve the Water Jug Problem using DFS

% solve_dfs(State, History, Moves, FinalState):
% If the State is the FinalState, return an empty list of Moves.
solve_dfs(State, History, [], State) :- % Removed FinalState
    true. % Always succeeds

% solve_dfs(State, History, Moves, FinalState):
% Move to the next state, update the history, and continue searching.
solve_dfs(State, History, [Move | Moves], FinalState) :-
    move(State, Move),
    update(State, Move, State1),
    legal(State1),
    not(member(State1, History)),
    solve_dfs(State1, [State1 | History], Moves, FinalState).

% Query to find a solution to the Water Jug Problem.
solve_water_jug_problem(FinalState, Moves) :- % Pass FinalState as an argument
    initial_state(jugs(0, 0)),
    solve_dfs(jugs(0, 0), [jugs(0, 0)], Moves, FinalState). % Use FinalState as the final goal

% Define the capacity of the jugs as constants.
capacity(1, 4).
capacity(2, 3).

% Define initial states.
initial_state(jugs(0, 0)).

% Define the legal states.
legal(jugs(V1, V2)) :- V1 >= 0, V2 >= 0.

% Define the available moves.
move(jugs(V1, V2), fill(1)) :- V1 < 4.
move(jugs(V1, V2), fill(2)) :- V2 < 3.
move(jugs(V1, V2), empty(1)) :- V1 > 0.
move(jugs(V1, V2), empty(2)) :- V2 > 0.
move(jugs(V1, V2), transfer(1, 2)) :- V1 > 0, V2 < 3.
move(jugs(V1, V2), transfer(2, 1)) :- V2 > 0, V1 < 4.

% Define how to update the state after a move.
update(jugs(V1, V2), fill(1), jugs(4, V2)).
update(jugs(V1, V2), fill(2), jugs(V1, 3)).
update(jugs(V1, V2), empty(1), jugs(0, V2)).
update(jugs(V1, V2), empty(2), jugs(V1, 0)).
update(jugs(V1, V2), transfer(1, 2), jugs(NewV1, NewV2)) :-
    Liquid is V1 + V2,
    (Liquid =< 3, NewV1 = 0, NewV2 = Liquid;
     Liquid > 3, NewV1 = Liquid - 3, NewV2 = 3).
update(jugs(V1, V2), transfer(2, 1), jugs(NewV1, NewV2)) :-
    Liquid is V1 + V2,
    (Liquid =< 4, NewV1 = Liquid, NewV2 = 0;
     Liquid > 4, NewV1 = 4, NewV2 = Liquid - 4).
```

% Adjust the liquid between the jugs.
adjust(Liquid, Excess, Liquid, 0) :- Excess =< 0.
adjust(Liquid, Excess, V, Excess) :- Excess > 0, V is Liquid - Excess.

OUTPUT:-

```
% w /CSMSS all/7th sem all notes/Ai notes/DFS1.pl compiled 0.02 sec, 21 clauses
?- solve_water_jug_problem(jugs(2, 0), Moves).
Moves = [fill(1), fill(2), empty(1), transfer(2, 1), fill(2), transfer(2, 1), empty(1), transfer
```

Date :

Experiment No:- 04

Aim: Write a program to solve Water Jug problem using Breadth First search.

Theory:

BFS algorithm:

Breadth-first search is a graph traversal algorithm that starts traversing the graph from the root node and explores all the neighboring nodes. Then, it selects the nearest node and explores all the unexplored nodes. While using BFS for traversal, any node in the graph can be considered as the root node.

Breadth-first search algorithm follow a simple, level-based approach to solve a problem.

algorithm:

1] In the various levels of the data, you can mark any node as the starting or initial node to begin traversing. The

2] BFS will visit the nearest and unvisited nodes and marks them. These values are also added to the queue. The q queue works on the FIFO model.

3]. In a similar manner, the remaining nearest and un-visited nodes in the graph are analyzed marked and added to the queue. These items are deleted from the queue as received and printed as the result.

Why do we need BFS Algorithm?:

- BFS is useful for analyzing the nodes in a graph and constructing the shortest path of traversing through these.
- BFS can traverse through a graph in the smallest number of iterations.
- The architecture of the BFS algorithm is simple and robust.
- BFS iterations are seamless, and there is no possibility of this algorithm getting caught up in the infinite loop problem.

Problem statement:

You have two jug A and jug B, with known capacity. Your goal is to measure a specific amount of water using these jugs. You can perform following operations

- Fill jug completely
- Pour water from one jug to another
- Empty jug completely.

BFS Algorithm For Water Jug Problem:

1] Create a queue to store states of the two jugs and an empty set to keep track of visited states

2] Add the initial state (0,0) to the queue and mark it as visited.

3] Start a loop that continues until the queue is not empty:

a). Dequeue a state (x,y) from the front of the queue.

b) check if (x,y) is equal to the target. If so, you have found a solution. Terminate the algorithm.

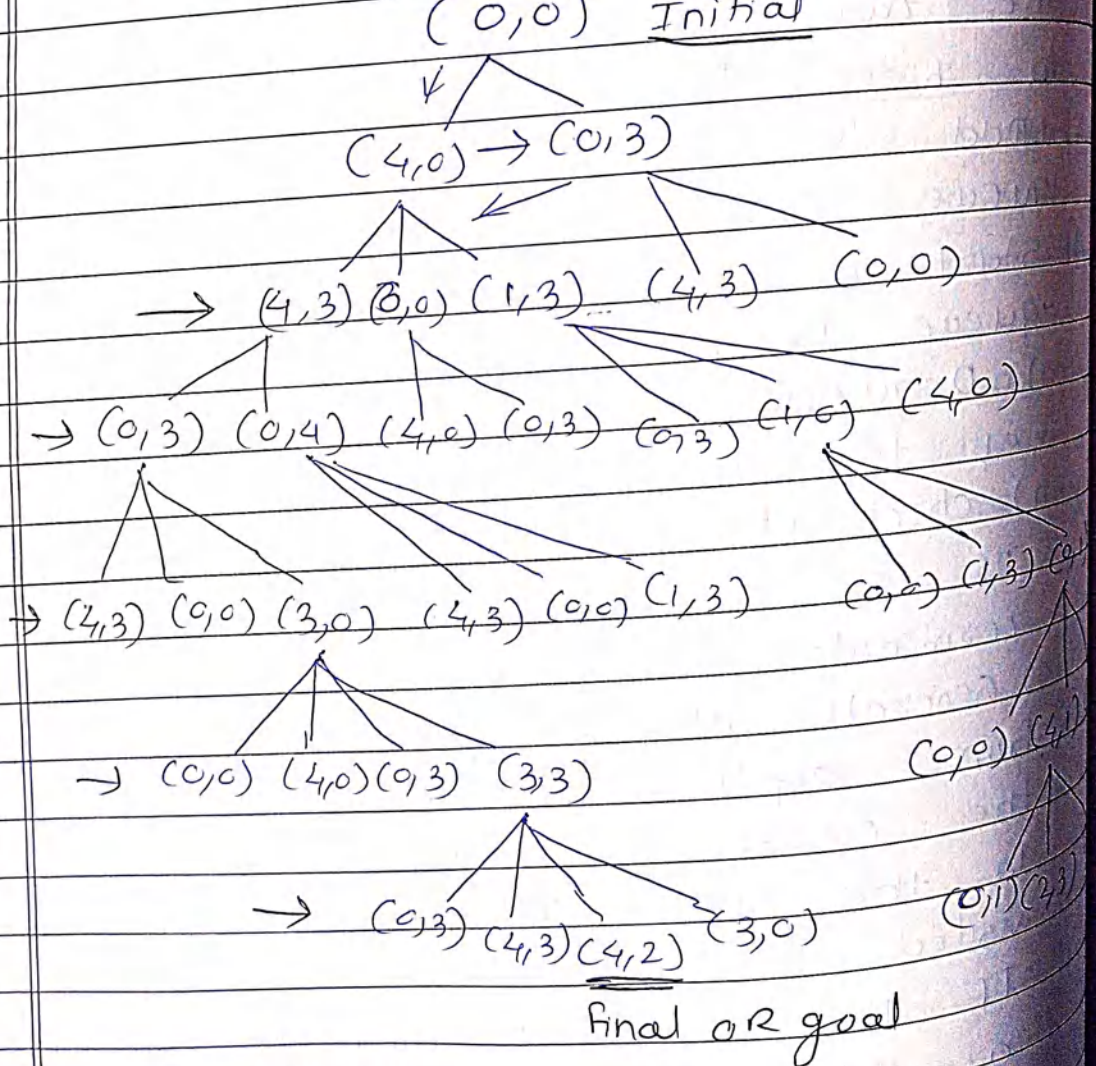
c) Generate all the possible states resulting from applying the valid operations to the current state. Add these states to the queue if they haven't been visited before.

4] If the loop completes without finding a solution, there is no way to measure the target amount using the given jugs.

5) The sequence of operations leading to the target state can be reconstructed

by tracing back from the target
to the initial state using info
stored during the BFS traversal

BFS (Breadth first search)



Jug 1 :- 4 l
Jug 2 :- 3 l
Goal :- (—, 2) OR (2, —)

 Date :

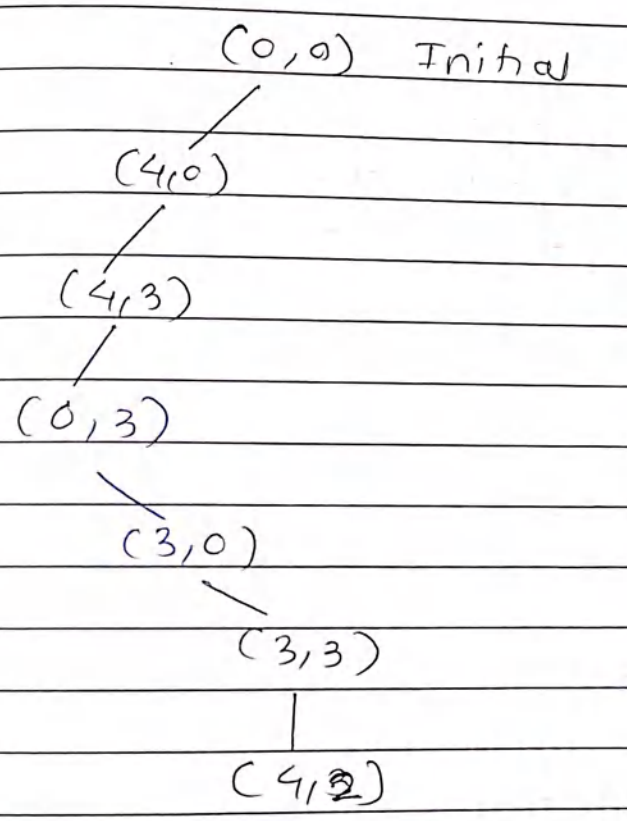


fig. search tree of BFS Path

Path:- (0,0) → (4,0) → (4,3) → (0,3) → (3,0) → (3,3) → (4,2)

Conclusion:-

In this practical I have studied Breadth first search and Implementation of BFS on Water jug problem.

Water Jug problem using BFS

Program:-

```
go :-
    start(Start),
    solve(Start, Solution),
    reverse(Solution, L),
    print(L, _).

solve(Start, Solution) :-
    breadthfirst([[Start]], Solution).

% breadthfirst([Path1, Path2, ...], Solution):
% Solution is an extension to a goal of one of the paths

breadthfirst([[Node | Path] | _], [Node | Path]) :-
    goal(Node).

breadthfirst([Path | Paths], Solution) :-
    extend(Path, NewPaths),
    append(Paths, NewPaths, Paths1),
    breadthfirst(Paths1, Solution).

extend([Node | Path], NewPaths) :-
    findall([NewNode, Node | Path],
        (next_state(Node, NewNode), \+ member(NewNode, [Node | Path])),
        NewPaths),
    !.

extend(_, []).

% States are represented by the compound term (4-gallon jug, 3-gallon jug);
% In the initial state, both jugs are empty:

start((0, 0)).

% The goal state is to measure 2 gallons of water:
goal((2, _)).
goal((_, 2)).

% Fill up the 4-gallon jug if it is not already filled:
next_state((X, Y), (4, Y)) :- X < 4.

% Fill up the 3-gallon jug if it is not already filled:
next_state((X, Y), (X, 3)) :- Y < 3.

% If there is water in the 3-gallon jug (Y > 0) and there is room in the 4-gallon jug (X < 4), THEN use it
to fill up
% the 4-gallon jug until it is full (4-gallon jug = 4 in the new state) and leave the rest in the 3-gallon
jug:
next_state((X, Y), (4, Z)) :-
    Y > 0, X < 4,
    Aux is X + Y,
    Aux >= 4,
    Z is Y - (4 - X).
```

```
% If there is water in the 4-gallon jug (X > 0) and there is room in the 3-gallon jug (Y < 3), THEN use it
to fill up
% the 3-gallon jug until it is full (3-gallon jug = 3 in the new state) and leave the rest in the 4-gallon
jug:
next_state((X, Y), (Z, 3)) :-
        X > 0, Y < 3,
        Aux is X + Y,
        Aux >= 3,
        Z is X - (3 - Y).

% There is something in the 3-gallon jug (Y > 0) and together with the amount in the 4-gallon jug it fits
in the
% 4-gallon jug (Aux is X + Y, Aux =< 4), THEN fill it all (Y is 0 in the new state) into the 4-gallon jug (Z is
Y + X):
next_state((X, Y), (Z, 0)) :-
        Y > 0,
        Aux is X + Y,
        Aux =< 4,
        Z is Y + X.

% There is something in the 4-gallon jug (X > 0) and together with the amount in the 3-gallon jug it fits
in the
% 3-gallon jug (Aux is X + Y, Aux =< 3), THEN fill it all (X is 0 in the new state) into the 3-gallon jug (Z is
Y + X):
next_state((X, Y), (0, Z)) :-
        X > 0,
        Aux is X + Y,
        Aux =< 3,
        Z is Y + X.

% Empty the 4-gallon jug IF it is not already empty (X > 0):
next_state((X, Y), (0, Y)) :-
        X > 0.

% Empty the 3-gallon jug IF it is not already empty (Y > 0):
next_state((X, Y), (X, 0)) :-
        Y > 0.

action((_, Y), (4, Y), fill1).
action((X, _), (X, 3), fill2).
action((_, Y), (4, Z), put(2, 1)) :- Y \= Z.
action((X, _), (Z, 3), put(1, 2)) :- X \= Z.
action((X, _), (Z, 0), put(2, 1)) :- X \= Z.
action((_, Y), (0, Z), put(2, 1)) :- Y \= Z.
action((_, Y), (0, Y), empty1).
action((X, _), (X, 0), empty2).

print([], _).

print([H | T], 0) :-
        write(start), tab(4), write(H), nl,
        print(T, H).

print([H | T], Prev) :-
        action(Prev, H, X),
        write(X), tab(4), write(H), nl,
        print(T, H).
```

Output:-

```
% v:/CSMSS all/7th sem all notes/Ai notes/BFS.pl compiled 0.02 sec, 28 clauses
?- go.
start    0,0
fill2    0,3
put(2,1)    3,0
fill2    3,3
put(2,1)    4,2
true
```

Experiment No:- 05

Aim: Solve 8-puzzle problem using BFS

Theory:

It has set off a 3x3 board having 9 block spaces out of which 8 blocks having tiles bearing number from 1 to 8. One space is left blank. The tile adjacent to blank space can move into it. We have to arrange the tile in a sequence for getting the goal state.

Initial State

1	2	3
8		4
7	6	5

Goal State

2	8	1
	4	3
7	6	5

The 8-puzzle problem belongs to the category of "Sliding block puzzle" type of problem. The puzzle is a square tray in which eight square tiles are placed. The remaining ninth square is uncovered. Each tile in the tray has a number on it.

A tile that is adjacent to blank space can be slide into that space. The game consists of a starting position and a specified goal position. The goal is to transform the starting position into the goal

position by sliding the tiles around.

The control mechanism for an 8-pu solvers must keep track of the orders which operations are performed, so th the operations can be undone one a starting configuration to a goal config such as two situations given below.

Figure (Starting State) (Goal State)

The state of 8-puzzle is the differe permutation of tiles within the frame. operations are the permissible moves up left, right, Here at each step of the a function f(x) will be defined which the combination of g(x) and h(x). i-e. $f(x) = g(x) + h(x)$

Where.

g(x): how many steps in the problem have already done or the current s from the initial state.

h(x): Number of ways through which can reach at the goal state fro the current state or

h(x): is the heuristic estimates that Compares the current state with th state note down how many state displaced from the initial or the

state. After calculating the f value at each step finally take the smallest f(x) value at every step and choose that as the next current state to get the goal.

Let us take an example.

figure (Initial state). Goal state.

1	2	3
4	5	6
7	8	

1	2	3
4	8	5
	7	6

1] Step 1

A

1	2	3
4	5	6
7	8	

F(x) = 0+4 = 4

1	2	3	B
4	5	6	
7		8	F(x) = 1+5 = 6

1	2	3	
4	5		C
7	8	6	F(x) = 1+3 = 4

F(x) is the step required to reach at the goal state from the initial state.

Step 2

C

1	2	3
4	5	
7	8	6

D

1	2	3
4	5	6
7	8	

F(x) = 2+4

E

1	2	
4	5	3
7	8	6

F(x) = 2+4

F

1	2	3
4		5
7	8	6

F(x) = 2+2

Step 3:

F

1	2	3
4		5
7	8	6

G

1		3
4	2	5
7	8	6

F(x) = 3

H

1	2	3
	4	5
7	8	6

F(x) = 3

J

1	2	3
4	5	
7	8	6

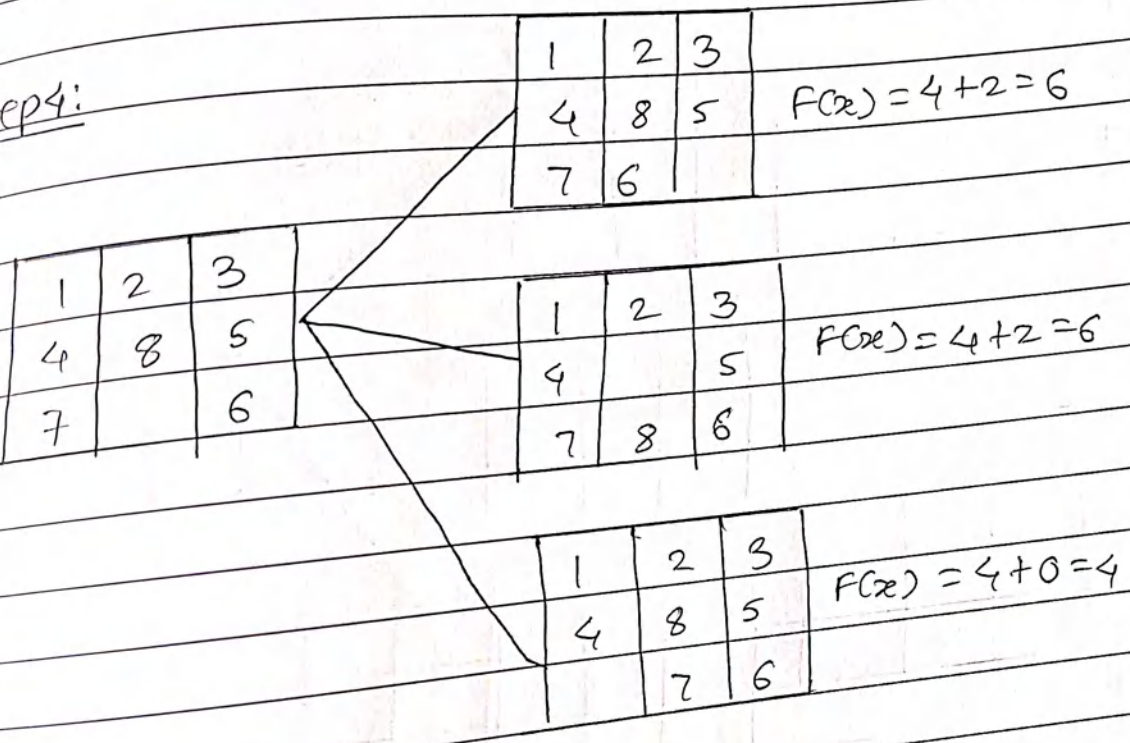
F(x) = 3+3 = 6

I

1	2	3
4	8	5
7		6

F(x) = 3

Step4:



Conclusion:

In this practicle we have implemented 8-puzzle problem using best first search using prolog.

8-Puzzle Problem:

program :

```
% Simple Prolog Planner for the 8 Puzzle Problem

% This predicate initializes the problem states. The first argument
% of solve/3 is the initial state, the 2nd the goal state, and the
% third the plan that will be produced.

test(Plan):-
    write('Initial state:'), nl,
    Init = [at(tile4,1), at(tile3,2), at(tile8,3), at(empty,4), at(tile2,5), at(tile6,6), at(tile5,7),
at(tile1,8), at(tile7,9)],
    write_sol(Init),
    Goal = [at(tile1,1), at(tile2,2), at(tile3,3), at(tile4,4), at(empty,5), at(tile5,6), at(tile6,7),
at(tile7,8), at(tile8,9)],
    nl, write('Goal state:'), nl,
    write_sol(Goal), nl, nl,
    solve(Init, Goal, Plan).

solve(State, Goal, Plan):-
    solve(State, Goal, [], Plan).

% Determines whether Current and Destination tiles are a valid move.
is_movable(X1, Y1) :- (1 is X1 - Y1) ; (-1 is X1 - Y1) ; (3 is X1 - Y1) ; (-3 is X1 - Y1).

% This predicate produces the plan. Once the Goal list is a subset
% of the current State, the plan is complete and it is written to
% the screen using write_sol/1.

solve(State, Goal, Plan, Plan):-
    is_subset(Goal, State), nl,
    write('Solution Plan:'), nl,
    write_sol(Plan).

solve(State, Goal, Sofar, Plan):-
    act(Action, Preconditions, Delete, Add),
    is_subset(Preconditions, State),
    \+ member(Action, Sofar),
    delete_list(Delete, State, Remainder),
    append(Add, Remainder, NewState),
    solve(NewState, Goal, [Action|Sofar], Plan).

% The problem has three operators.
% 1st arg = name
% 2nd arg = preconditions
% 3rd arg = delete list
% 4th arg = add list.

% Tile can move to a new position only if the destination tile is empty & Manhattan distance =
1
```

```
act(move(X, Y, Z),
    [at(X, Y), at(empty, Z), is_movable(Y, Z)],
    [at(X, Y), at(empty, Z)],
    [at(X, Z), at(empty, Y)]).

% Utility predicates.

% Check if the first list is a subset of the second.
is_subset([H|T], Set):-
    member(H, Set),
    is_subset(T, Set).
is_subset([], _).

% Remove all elements of the first list from the second to create the third.
delete_list([H|T], Curstate, Newstate):-
    remove(H, Curstate, Remainder),
    delete_list(T, Remainder, Newstate).
delete_list([], Curstate, Curstate).

remove(X, [X|T], T).
remove(X, [H|T], [H|R]):-
    remove(X, T, R).

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

Output:

```
% w:/CSMSS all/7th sem all notes/Ai notes/puzzle pl compiled 0.00 sec, 14 clauses
?- test(Plan).
Initial state:
at(tile7,9)
at(tile1,8)
at(tile5,7)
at(tile6,6)
at(tile2,5)
at(empty,4)
at(tile8,3)
at(tile3,2)
at(tile4,1)

Goal state:
at(tile8,9)
at(tile7,8)
at(tile6,7)
at(tile5,6)
at(empty,5)
at(tile4,4)
at(tile3,3)
at(tile2,2)
at(tile1,1)
```

Experiment No: 06

Aim: Solve Robot (Traversal) problem using means End Analysis.

Theory:

- Means-Ends Analysis is problem-solving techinque used in Artificial intelligence for limiting search in AI programs.
- It is a mixture of Backward and Forword search technique.
- The MEA technique was first introduced in 1961 by Allen Newell, and Herbert A. Simon in their problem-solving computer program, which was named as General problem Solver (GPS)
- The MEA analysis process centered on the evaluation of the difference between the current state and goal state.

How means-ends analysis Works:

The means-ends analysis process can be applied recursively for a problem. It is a strategy to control search in problem-solving. Following are the main steps which describes the working of MEA technique for solving a problem.

1. First, evalute the difference between Initial state and final state.
2. Select the various operators which can be applied for each difference.
3. Apply the operator at each difference, which reduces the difference between the current state and goal state.

Algorithm For Means-Ends Analysis:

Let's we take current state as CURRENT and goal state as GOAL, then following are the steps for the MEA algorithm

* Step 1: Compare CURRENT to GOAL, if there are no differences between both then return success and Exit.

* Step 2: Else, select the most significant difference and reduce it by doing the following steps until the success or failure occurs.

a). Select a new operator O which is applicable for the current difference, and if there is no such operator, then signal failure

b) Attempt to apply operator O to CURRENT Make a description of two states.

i) O-Start, a state in which O? s

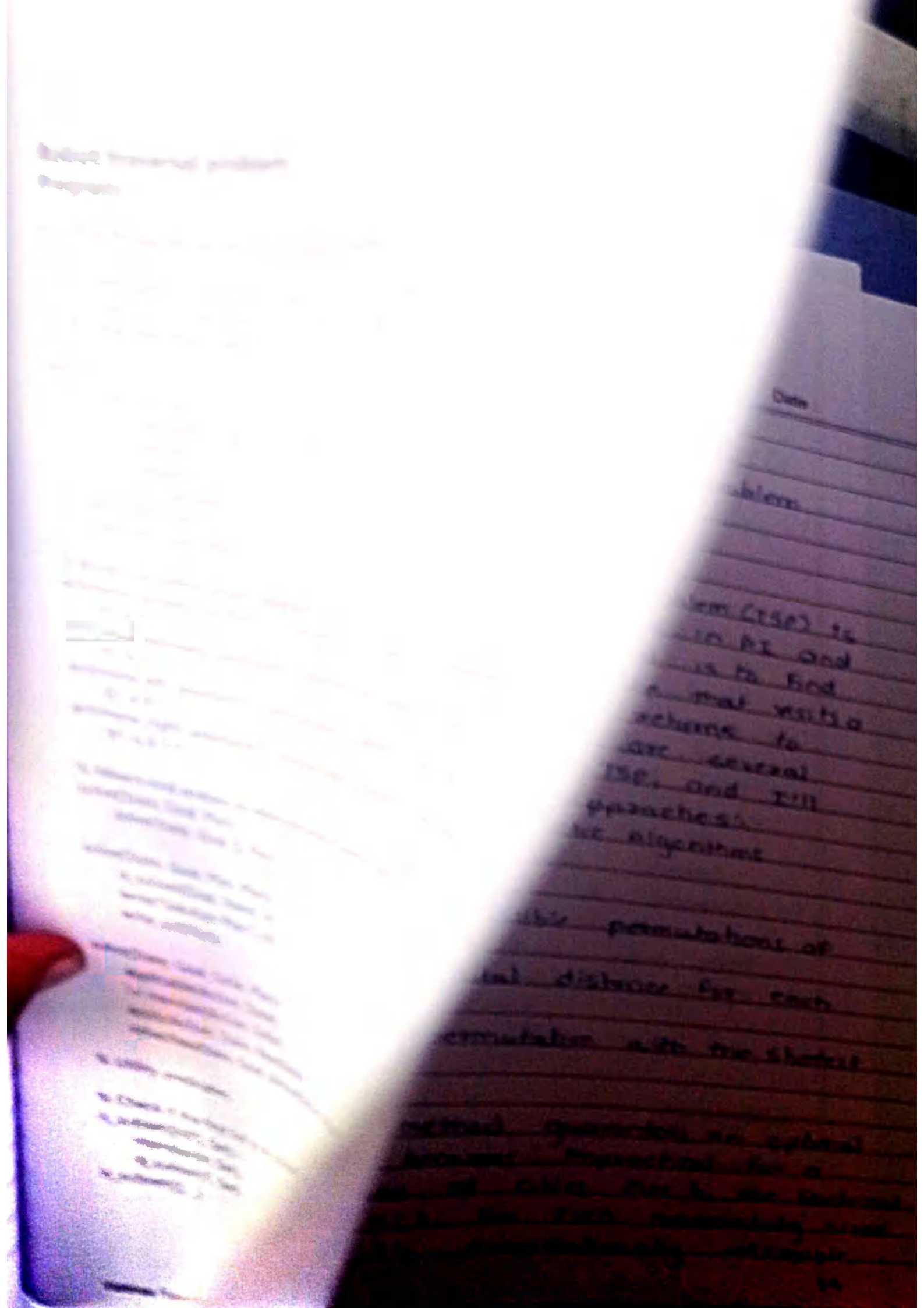
preconditions are satisfied.

ii) O-Result, the state that would result if O were applied In O-start

c) IF
(First-part <--- MEA (CURRENT, O-START)
And
(LAST-part <--- MEA (O-Result, GOAL) are successful, then signal success and return the result of combining FIRST-PART, O, and LAST-PART.

Conclusion:

I have implemented Robot traversal problem using means End Analysis.



Robot traversal problem:

Program:

```
%% Simple Prolog Planner for Robot Traversal Problem

% This predicate initializes the problem states. The first argument
% of solve/3 is the initial state, the 2nd the goal state, and the
% third the plan that will be produced.

test(Plan):-
    write('Initial state:'), nl,
    Init = [position(0, 0), direction(north)],
    write_sol(Init),
    Goal = [position(2, 3)],
    nl, write('Goal state:'), nl,
    write_sol(Goal), nl, nl,
    solve(Init, Goal, Plan).

% Robot can move forward, backward, turn left, or turn right.
act(move_forward, [position(X, Y), direction(north)], [], [position(X, Y1), direction(north)]) :-
    Y1 is Y + 1.
act(move_backward, [position(X, Y), direction(south)], [], [position(X, Y1), direction(south)]) :-
    Y1 is Y - 1.
act(move_left, [position(X, Y), direction(west)], [], [position(X1, Y), direction(west)]) :-
    X1 is X - 1.
act(move_right, [position(X, Y), direction(east)], [], [position(X1, Y), direction(east)]) :-
    X1 is X + 1.

% Means-end analysis to determine actions needed to achieve the goal.
solve(State, Goal, Plan):-
    solve(State, Goal, [], Plan).

solve(State, Goal, Plan, Plan):-
    is_subset(Goal, State), nl,
    write('Solution Plan:'), nl,
    write_sol(Plan).

solve(State, Goal, Sofar, Plan):-
    applicable(Action, State),
    \+ member(Action, Sofar),
    apply(Action, State, NewState),
    solve(NewState, Goal, [Action|Sofar], Plan).

% Utility predicates.

% Check if the first list is a subset of the second.
is_subset([H|T], Set):-
    member(H, Set),
    is_subset(T, Set).
is_subset([], _).
```

```
% Remove all elements of the first list from the second to create the third.
delete_list([H|T], Curstate, Newstate):-
    remove(H, Curstate, Remainder),
    delete_list(T, Remainder, Newstate).
delete_list([], Curstate, Curstate).

remove(X, [X|T], T).
remove(X, [H|T], [H|R]):-
    remove(X, T, R).

write_sol([]).
write_sol([H|T]):-
    write_sol(T),
    write(H), nl.

% Determine applicable actions based on the current state.
applicable(Action, State):-
    act(Action, Preconditions, _, _),
    is_subset(Preconditions, State).

% Apply an action to the current state to produce a new state.
apply(Action, State, NewState):-
    act(Action, _, Delete, Add),
    delete_list(Delete, State, Remainder),
    append(Add, Remainder, NewState).
```

Output:

```
% d:/CSMSS all/7th sem all notes/Ai notes/robot.pl compiled 0.02 sec, 18 clauses
?- test(Plan).
Initial state:
direction(north)
position(0,0)

Goal state:
position(2,3)
```

Date :

Experiment NO:07

Aim: Solve traveling salesman problem

Theory:

The Traveling Salesman Problem (TSP) is a classic optimization problem in AI and computer science. Its goal is to find the shortest possible route that visits a given set of cities and returns to the starting city. There are several algorithms to solve the TSP, and I'll describe two common approaches: Brute Force and Genetic Algorithms

1 Brute Force:-

- Generate all possible permutations of cities.
- Calculate the total distance for each permutation.
- Select the permutation with the shortest distance.

While this method guarantees an optimal solution, it becomes impractical for a large number of cities due to the factorial complexity (n!). For even moderately sized instances, it's computationally infeasible.

2] Genetic Algorithms:-

- Initialize a populations of potential routes
- Evaluate each route's fitness (shortestness of the path).
- Select individuals for reproduction based on their fitness.
- Create new routes through crossover (combining routes) and mutation (small changes).
- Repeat the evaluation, selection, crossover and mutations steps for several generations
- The best route found after a certain number of generations is the solution

Genetic algorithms can handle large instances of the TSP more efficiently than brute force, algorithm, although they may not always guarantee the optimal solution.

Conclusion:

In this problem i have implement traveling salesman problem in prolog

Traveling Salesman Problem with Genetic Algorithm:

Program:

```
:- use_module(library(random)).

% Define the number of cities
num_cities(5).

% Define the population size for the GA
population_size(10).

% Define the mutation rate for the GA
mutation_rate(0.1).

% Define cities
city(0, 0).
city(1, 2).
city(3, 1).
city(4, 3).
city(2, 4).

% Generate a random route
generate_random_route(Route) :-
    num_cities(NumCities),
    length(Route, NumCities),
    numlist(0, NumCitiesMinusOne, Cities),
    random_permutation(Cities, Route).

% Calculate the total distance of a route
calculate_total_distance(Route, TotalDistance) :-
    append(Route, [Route[0]], ClosedRoute), % Close the route
    calculate_total_distance_helper(ClosedRoute, TotalDistance).

calculate_total_distance_helper([City1, City2 | Rest], TotalDistance) :-
    city(City1, X1-Y1),
    city(City2, X2-Y2),
    DX is X1 - X2,
    DY is Y1 - Y2,
    Distance is sqrt(DX*DX + DY*DY),
    calculate_total_distance_helper([City2 | Rest], RestDistance),
    TotalDistance is Distance + RestDistance.
calculate_total_distance_helper([_], 0).

% Perform crossover between two parent routes to produce a child route
crossover(Parent1, Parent2, Child) :-
    length(Parent1, Length),
    random_between(1, Length, CrossoverPoint),
    append(Prefix, Suffix, Parent1),
    append(Prefix, RestParent2, Parent2),
    append(RestParent2, Suffix, Child).

% Mutate a route by swapping two cities
mutate(Route, MutatedRoute) :-
    mutation_rate(MutationRate),
    (random_float < MutationRate ->
        random_permutation(Route, MutatedRoute)
```

Traveling Salesman Problem with Brute Force:

Program:

```
% Define cities and distances between them.
city_distance(city1, city2, 10).
city_distance(city1, city3, 15).
city_distance(city1, city4, 20).
city_distance(city2, city3, 35).
city_distance(city2, city4, 25).
city_distance(city3, city4, 30).

% Entry point for the TSP solver.
tsp(StartCity, OptimalTour, MinDistance) :-
    findall(City, city_distance(StartCity, City, _), Cities),
    permutation(Cities, Permutation),
    calculate_distance([StartCity | Permutation], Distance),
    MinDistance is Distance,
    OptimalTour = [StartCity | Permutation].

% Calculate the total distance of a tour.
calculate_distance([_], 0).
calculate_distance([City1, City2 | Rest], TotalDistance) :-
    city_distance(City1, City2, Distance),
    calculate_distance([City2 | Rest], RemainingDistance),
    TotalDistance is Distance + RemainingDistance.

% Example query
% Replace 'city1' with the starting city of your choice.
% ?- tsp(city1, Tour, Distance).
```

Output:

```
% v:/CSMSS all/7th sem all notes/A1 notes/tsp.pl compiled 0.00 sec, 0 clauses
?- tsp(city1, Tour, Distance).
Tour = [city1, city2, city3, city4],
Distance = 75
```

```
    ;   MutatedRoute = Route
    ).

% Create an initial population
initialize_population(StartCity, Population) :-
    population_size(PopSize),
    findall(Route, (between(1, PopSize, _), generate_initial_route(StartCity, Route)), Population).

generate_initial_route(StartCity, Route) :-
    num_cities(NumCities),
    length(Route, NumCities),
    numlist(0, NumCitiesMinusOne, Cities),
    random_permutation(Cities, ShuffledCities),
    select(StartCity, ShuffledCities, Route).

% Evolutionary algorithm iteration
evolve_population([], []).
evolve_population([Parent1, Parent2 | Rest], [Child1, Child2 | NewPopulation]) :-
    crossover(Parent1, Parent2, Child1),
    crossover(Parent2, Parent1, Child2),
    mutate(Child1, MutatedChild1),
    mutate(Child2, MutatedChild2),
    evolve_population(Rest, NewPopulation).

% Perform the GA iterations
ga_iteration(Population, NewPopulation) :-
    evolve_population(Population, Children),
    append(Population, Children, CombinedPopulation),
    sort_population(CombinedPopulation, SortedPopulation),
    take_best(SortedPopulation, PopulationSize, NewPopulation).

% Sort the population based on fitness (total distance)
sort_population(Population, SortedPopulation) :-
    predsort(compare_fitness, Population, SortedPopulation).

compare_fitness(Order, Route1, Route2) :-
    calculate_total_distance(Route1, Fitness1),
    calculate_total_distance(Route2, Fitness2),
    compare(Order, Fitness1, Fitness2).

% Take the best N individuals from the population
take_best([], _, []).
take_best(Population, N, BestPopulation) :-
    length(Population, Length),
    MaxN is min(N, Length),
    take_best_helper(Population, MaxN, BestPopulation).

take_best_helper(_, 0, []).
take_best_helper([Individual | Rest], N, [Individual | BestRest]) :-
    N > 0,
    N1 is N - 1,
    take_best_helper(Rest, N1, BestRest).

% Main function
tsp_genetic(StartCity, OptimalTour) :-
    initialize_population(StartCity, Population),
    ga_iterations(Population, max_generations, OptimalTour).
```


```
% Perform GA iterations
ga_iterations(Population, 0, BestRoute) :-
    find_best_route(Population, BestRoute).
ga_iterations(Population, GenerationsLeft, BestRoute) :-
    ga_iteration(Population, NewPopulation),
    NextGenerationsLeft is GenerationsLeft - 1,
    ga_iterations(NewPopulation, NextGenerationsLeft, BestRoute).

% Print the best route
print_best_route(BestRoute) :-
    writeln('Best Route:'),
    writeln(BestRoute).

% Print the total distance of the best route
print_total_distance(BestRoute) :-
    calculate_total_distance(BestRoute, Fitness),
    writeln('Total Distance:'),
    writeln(Fitness).
```

Output:

```
% c:/CSNSS all/7th sem all notes/AI notes/tsp_genetic.pl compiled 0.02 sec, 30 clauses
- tsp_genetic(0, OptimalTour), print_best_route(OptimalTour), print_total_distance(OptimalTour
    Best Route:
    [0, 4, 1, 3, 2]

    Total Distance:
    10.472
```