TERMWORK-5

* PROBLEM STATEMENT :-

Design an algorithm for TO SOLVE EIGHT QUEENS PROBLEM and develop a PROLOG program for the same.

* THEORY :- The eight queens problem is the problem of placing eight queens on an 8x8 chessboard such that none of them attack (no two are in the same row, column, or diagonal). More generally, the n queens problem places n queens on an n x n chessboard. There are different solution for the problem. Backtracking

START

1. Begin from the leftmost column
2. if all the queens are placed,
   return true / print configuration
3. Check for all rows in the current column
   a) if queens placed safely, mark row & colum; and recursively check if we approach in the current configuration, do we obtain a solution or not.
   b) if placing yields a solution, return true
   c) if placing does not yield a solution, unmark and try other rows.
4. if all rows tried and solution not obtained, return false & backtrack

END

* Algorithm:-

```
N-Queens (k, n)
{
  For i ← 1 to n
     do if Place (k, i) then
  {
     x[k] ← i ;
   if (k == n) then
     write (x [1 .... n]);
    else
     N - Queens (k+1, n);
  }
}
```

PROGRAM :-

```
global N
N=8
def printSolution(board):
  for i in range(N):
    for j in range(N):
      print(board[i][j], end=' ')
  print()
  for i in range(col):
    if board[row][i]==1:
      return False
  for i,j in zip(range(row,-1,-1),
  range(col,-1,-1)):
    if board[i][j]==1:
      return False
  return True
def solveNQUtil(board, col):
  if col>=N:
    return True
  for i in range(N):
    if isSafe(board,i,col):
      board[i][col]=1
      if solveNQUtil(board,col+1)==True:
        return True
      board[i][col]=0
def solveNQ():
  board=[[0 for in range(N)] for in range(N)]
  print(board)
```

```
if solveNQUtil(board, 0) == False:
    print("Solution does not exist")
    return False
printSolution(board)
return True
solveNQ()
```

OUTPUT:-

```
- - - q - - - -
- - - - - - q -
- - q - - - - -
- - - - - - - q
- q - - - - - -
- - - - q - - -
q - - - - - - -
- - - - - q - -
```

TERMWORK-6

PROBLEM STATEMENT :-

Design an algorithm to implement Depth-first-search and develop a PROLOG program for the same.

THEORY :-

Depth-first-search is a graph traversal algorithm that explores the graph by visiting nodes in depth-first-search manner. The algo starts at a given node & explores as far as possible along each branch before backtracking.

DFS can be implemented using recursive rules & predicates

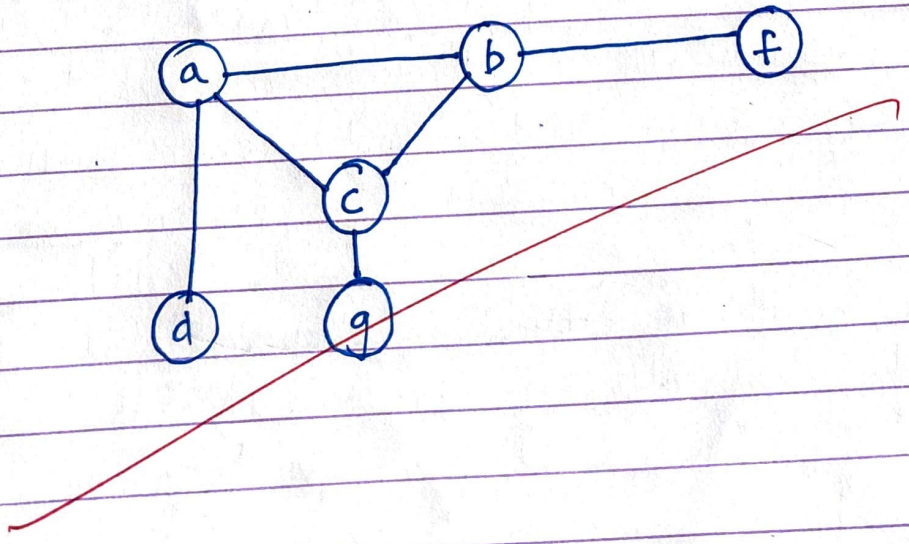
1. Start at a given node in the graph.
2. Visit the current node in the graph
3. Explore unvisited neighbours of the current node & recursively applies to the DFS algorithm to that neighbour.
4. Maintain a path of visited nodes
5. Until the termination condition is met

Program :-

```
child (a,b).
child (a,c).
child (a,d).
child (b,c).
child (b,f).
child (c,g).

paths (A, G [A|Z]) :- childnode (A, G, Z]
childnode (A, G, [G]) :- childnode (A, G).
childnode (A, G, [X|L]) :- child (A, Y), childnode (X, G).
```

Graph :-



OUTPUT :-

```
? = path(a,g,L)
L = [a,c,g]

? = path(a,e,L)
L = [a,b,e]
```

Tracing :

```
path(a,g,[a|Z]) :- childnode(a,g,Z)
childnode(a,g,[g]) :- child(a,g)
childnode(a,g,[X|L]) :- child(a,X), childnode(X,g,L).
child(a,c), childnode(c,g,L)
childnode(c,g,[g]) :- child(c,g)
  L = [a,c,g]
```

Conclusion :-

In this turnwork we learnt and implemented DFS Algo in PROLOG a executed the graph with the resulting path in goal node.

write graph

09/10

29/8/25

TERMWORK-7

* PROBLEM STATEMENT:-

Design an algorithm for TO IMPLEMENT BREDTH FIRST SEARCH and develop a PROLOG program for the same.

* THEORY :- The only catch here is, that, unlike trees, graphs may come to the same node again. To avoid processing a node more than once, we divide the vertices into two categories :-
  - Visited
  - Non-visited.

* Algorithm :-

Step 1 :- Consider the graph you want to navigate
Step 2 :- Select any vertex in your graph (say v1), from which you want to traverse the graph.
Step 3 :- Utilize the following two data structures for traversing the graph.
Visited array (size of the graph).
Queue data structure
Step 4 :- Add the starting vertex to the visited array, and afterward, you add v1's adjacent vertices to the queue data structure
Step 5 :- Now using the FIFO concept, remove the first element from the queue, put it into the visited array, and then add the vertices of the removed element to the queue
Step 6 :- Repeat Step 5 until the queue is not empty & no vertex is left to be visited.

PSEUDOCODE :-

```
Breadth_First_Search (Graph, x):
Let Q be the queue
Q.enque (x)
mark X node as visited.

While (Q is not empty)
  Y=Q.dequeue()

Process all the neighbours of Y, For all the neighbours Z of Y.
If Z is not visited :
    Q.enqueued (Z)
  Mark Z as visited.
```

PROGRAM

```
#include <stdio.h>
int n,i,j,visited[10], queue[10], front=-1, rear=-1;
int adj[10][10];

void bfs(int v)
{
    for (i=1; i<n; i++)
    if (adj[v][i] && !visited[i])
      queue[++rear]=i;
  if (front<=rear)
  {
     visited[queue[front]]=1;
     bfs(queue[front++]);
  }
}

void main()
 {
   int v;
   printf("Enter the number of vertices:");
   scanf("%d",&n);
   for(i=1; i<=n; i++)
 {

   queue[i]=0;
   visited[i]=0;
}
 print("Enter graph data in matrix form:");
     for(i=1; i<=n; i++)
```

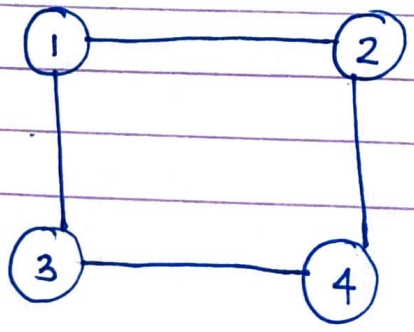
```
        for (j=1; j<=n; j++)
            scanf ("%d", &adj[i][j]);

    printf ("Enter the starting vertex :");
        scanf ("%d", &v);
        bfs(v);
    printf ("The node which are reachable are :");
            for (i=1; i<=n; i++)
                if (visited[i])
                printf ("%d\t", i);
            else
                print~~f ("%d\t",i);~~ ("BFS is not possible");
            return 0;
}
```

OUTPUT :-

```
Enter the number of vertices : 4
Enter graph data in matrix form:
    0 1 1 0
    1 0 0 1
    1 0 0 1
    0 1 1 0
Enter the starting vertex : 2
The node which are reachable are
    1 2 3 4.
```

Graphs:-



TERMWORK-8

PROBLEM STATEMENT :-

Design an algorithm for TO SOLVE MONKEY BANANA PROBLEM and develop a PROLOG program for the same.

THEORY :-

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

We can solve this by the following tricks :-

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

PROGRAM :-

```
move (state (middle, onbox, middle, hasnot),
    grasp,
    state (middle, onbox, middle, has)).
move (state (P, onfloor, P, H),
    climb,
    state (P, onbox, P, H)).
move (state (P1, onfloor, P1, H),
    drag (P1, P2),
    state (P2, onfloor, P2, H)).
move (state (P1, onfloor, B, H),
    walk (P1, P2),
    state (P2, onfloor, B, H)).
canget (state (_, _, _, has)).
canget (State 1) :-
        move (State 1, _, Stage 2),
            canget (State 2).
```