Expt 5:

The eight queens problem is the problem of placing eight queens on 8x8 chess board 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 NXN chess board. There are different solution for the problem, backtracking.

START

1. Begin from the left most 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 and column and recursively check if we approach in the current configuration, do we obtain a solution or not.
   b) if placing yields a solution returns true
   c) if placing does not yield a solution and mark and try other rows.
4. If all rows tried and solution not obtained return falls and BackTrack

END

Program:

```
def solve(board, row=1):
   if row > 8:
      print("Solution:")
      print(board)
      return
   for col in range(1, 9):
      if safe(board, row, col):
         place_queen(board, row, col)
         solve(board, row + 1)
         remove_queen(board, row, col)

def safe(board, row, col):
   for i in range(1, row):
      if board[i] == col or abs(board[i] - col) ==
abs(i - row):
         return False
   return True

def place_queen(board, row, col):
   board[row] = col

def remove_queen(board, row, col):
   board[row] = 0

board = [0] * 9
solve(board)
```

Expt 6:

Depth first search is a graph travel algorithm that explore the graph by visiting notes in depth first manner. They algorithm starts at a given node and explores as far as possible along each branch before back tracking.

```
DFS(S)
OPEN ←(5, null): [ ]
CLOSED←empty list
while OPEN is not empty
nodePair←head OPEN
(N._)  ←nodePair
if GOAL TEST(N) =TRUE
return RECONSTRUCTPATH(nodePair, CLOSED)
else CLOSED ← nodePair: CLOSED
neighbours ← MOVEGEN(N)
newNodes ←REMOVESEEN (neighbours, OPEN,
CLOSED)
newPairs← MAKEPAIRS(newNodes, N)
OPEN←newPairs ++ (tail OPEN)
 return empty list
```

Program

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

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).
```

Input & output

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

Expt 7:

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: i)visited ii)non visited

BFS(S)
OPEN(S, null): [ ]
CLOSED ←empty list
while OPEN is not empty
nodePair←head OPEN
(N._)← nodePair
if GOAL TEST(N) = TRUE)
return RECONSTRUCTPATH(nodePair, CLOSED)
else CLOSED←nodePair: CLOSED
neighbours ←MOVEGEN(N)
newNodes← REMOVESEEN (neighbours, OPEN, CLOSED)
newPairs ←MAKEPAIRS(newNodes, N)
OPEN ← (tail OPEN) ++ newPairs

Program:

```
 graph = {
 '5' : ['3','7'],
 '3' : ['2', '4'],
 '7' : ['8'],
 '2' : [],
 '4' : ['8'],
 '8' : []
}

visited = [] .
queue = []

def bfs(visited, graph, node):
 visited.append(node)
 queue.append(node)

 while queue:
  m = queue.pop(0)
  print (m, end = " ")

  for neighbour in graph[m]:
   if neighbour not in visited:
    visited.append(neighbour)
    queue.append(neighbour)


print("Following is the Breadth-First Search")
bfs(visited, graph, '5')
```

Expt 8:

Here hungry monkey is in a room and his near the door, the monkey is on the floor, bananas have been hung from the centre of the ceiling of the room, this is block present in the room near the window. The monkey wants the banana, but cannot reach it.

We can solve this by following tricks

When the block is at the middle and monkeys on the top of the block the monkey does not have the bananas, then using the graph action it will change from has not state to have state.

From the floor it can move to the top of the block by performing the action climb.

The push and drag operation moves the block from one place to another

Monkey can move from one to another using walk or move classes.

Program

```
move(state(middle,onbox,middle,hasnot),
  grasp,
  state(middle,onbox,middle,has)).
move(state(P,onfloor,P,H),
  climb,
  state(P,onbox,P,H)).
move(state(P1,onfloor,P1,H),
  drag(P1,P2),
  state(P2,onfloor,P2,H)).
move(state(P1,onfloor,B,H),
  walk(P1,P2),
  state(P2,onfloor,B,H)).
canget(state(_,_,_,has)).
canget(State1) :-
  move(State1,_,State2),
  canget(State2).
```

Input & output

```
?- canget(state(atdoor, onfloor, atwindow,
hasnot)).
true .
```