### 

[**Fibonacci Series**](#_812uc43gbkhv) **1**

[**Family Tree**](#_dklm5k9lqwve) **1**

[**Angry Raja**](#_d7ncthv8j37v) **2**

[**Hanoi**](#_osh5rieqmurh) **2**

[**Bluebird is mammal**](#_de92e5iv21aw) **3**

[**John is Happy**](#_7uc5kq3idt3i) **3**

[**Truth Table**](#_beshjdqdc2mf) **3**

[**N Queens**](#_vryv6dlr6oiv) **3**

[**Hill Climbing 4 Queens**](#_7f2qhhi8i3ak) **4**

[**8 Puzzle using uninformed search**](#_oflu8jcjdxeh) **8**

### 

### 

### **Fibonacci Series**

fib(0,0).

fib(1,1).

fib(X,N):-

N>1,

N1 is N-1,

N2 is N-2,

fib(X1,N1),fib(X2,N2),

X is X1+X2.

fib\_seq(S,N):- N>1,fib\_seq\_(N,SR,1,[1,0]),reverse(SR,S).

fib\_seq\_(N,Seq,N,Seq).

fib\_seq\_(N,Seq,N0,[B,A|Fs]):- N>N0,N1 is N0+1,C is A+B, fib\_seq\_(N,Seq,N1,[C,B,A|Fs]).

**Output:** fib\_seq(S,23)

Will display all the number

### **Family Tree**

parent(lalit, naresh).

parent(lalit, vicky).

parent(lalit, laveena).

parent(meena, naresh).

parent(meena, laveena).

parent(meena, vicky).

female(meena).

female(laveena).

female(raveena).

male(lalit).

male(naresh).

male(vicky).

mother(X,Y):- parent(X,Y),female(X).

father(X,Y):- parent(X,Y),male(X).

sister(x,y):-parent(z,x),parent(z,y),female(x).

brother(x,y):-parent(z,x),parent(z,y),male(x).

**Output:** ?- father(lalit,naresh)

True

### **Angry Raja**

hungry(rimi).

barks(rimi):-hungry(rimi).

angry(raja):-barks(rimi).

bool(true).

bool(false).

tablebody(A,B,E):-

bool(A),

bool(B),

write(A),

write(' '),

write(B),

write(' '),

write(E),nl,fail.

**Output-**  angry(raja)- true

### **Hanoi**

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).

**Output:** move(4,source,target,auxiliary).

### **Bluebird is mammal**

mammal(X):-horse(X).

horse(bluebird).

### **John is Happy**

lucky(\_).

study(\_).

passexam(X):-lucky(X),study(X).

happy(X):-passexam(X);lucky(X).

john(X):-lucky(X), not study(X).

**Output:** happy(john).

true

### **Truth Table**

and(A, B) :- A, B.

evaluate(E, true) :- E, !.

evaluate(\_, false).

bool(true).

bool(false).

tableBody(A,B,E) :-

bool(A),

bool(B),

write(A),

write(' \t '),

write(B),

write(' \t '),

evaluate(E, Result),

write(Result),nl, fail.

**Output:** tableBody(A,B,and(A,B)).

### **N Queens**

print ("Enter the number of queens")

N = int(input())

board = [[0]\*N for \_ in range(N)]

def is\_attack(i, j):

for k in range(0,N):

if board[i][k]==1 or board[k][j]==1:

return True

for k in range(0,N):

for l in range(0,N):

if (k+l==i+j) or (k-l==i-j):

if board[k][l]==1:

return True

return False

def N\_queen(n):

if n==0:

return True

for i in range(0,N):

for j in range(0,N):

if (not(is\_attack(i,j))) and (board[i][j]!=1):

board[i][j] = 1

if N\_queen(n-1)==True:

return True

board[i][j] = 0

return False

N\_queen(N)

for i in board:

print (i)

### **Hill Climbing 4 Queens**

import random

import numpy as np

N\_QUEEN = 4

def in\_conflict(column, row, other\_column, other\_row):

if column == other\_column:

return True # Same column

if row == other\_row:

return True # Same row

if abs(column - other\_column) == abs(row - other\_row):

return True # Diagonal

return False

def in\_conflict\_with\_another\_queen(row, column, board):

for other\_column, other\_row in enumerate(board):

if in\_conflict(column, row, other\_column, other\_row):

if row != other\_row or column != other\_column:

return True

return False

def count\_conflicts(board):

cnt = 0

for queen in range(0, len(board)):

for other\_queen in range(queen + 1, len(board)):

if in\_conflict(queen, board[queen], other\_queen, board[other\_queen]):

cnt += 1

return cnt

def evaluate\_state(board):

return (len(board) - 1) \* len(board) / 2 - count\_conflicts(board)

def print\_board(board):

for row in range(len(board)):

line = ''

for column in range(len(board)):

if board[column] == row:

line += 'Q' if in\_conflict\_with\_another\_queen(row, column, board) else 'q'

else:

line += '\_'

print(line)

print("")

def init\_board(nqueens):

board = []

for column in range(nqueens):

board.append(random.randint(0, nqueens - 1))

print('Initial State:')

print\_board(board)

return board

def Hill\_Climbing(board):

i = 0

optimum = (len(board) - 1) \* len(board) / 2

evaluation = [evaluate\_state(board)]

while evaluate\_state(board) != optimum:

i += 1

max\_score\_of\_each\_column = []

row\_resulting\_in\_max\_score = []

for col in range(len(board)):

col\_scores = []

for row in range(len(board)):

new\_board = board.copy()

new\_board[col] = row

col\_scores.append(evaluate\_state(new\_board))

if max(col\_scores) > evaluate\_state(board):

max\_score\_of\_each\_column.append(max(col\_scores))

row\_resulting\_in\_max\_score.append(np.argmax(col\_scores))

else:

max\_score\_of\_each\_column.append(False)

row\_resulting\_in\_max\_score.append(False)

if max(max\_score\_of\_each\_column):

maximizing\_col = np.argmax(max\_score\_of\_each\_column)

maximizing\_row = row\_resulting\_in\_max\_score[maximizing\_col]

board[maximizing\_col] = maximizing\_row

evaluation.append(evaluate\_state(board))

if evaluate\_state(board) == optimum:

print('\nSolved Puzzle!')

print('\nFinal State:')

print\_board(board)

if \_\_name\_\_ == "\_\_main\_\_":

Hill\_Climbing(init\_board(N\_QUEEN))

### **8 Puzzle using uninformed search**

| import sys  import numpy as np      class Node:  def \_\_init\_\_(self, state, parent, action):  self.state = state  self.parent = parent  self.action = action      class StackFrontier:  def \_\_init\_\_(self):  self.frontier = []    def add(self, node):  self.frontier.append(node)    def contains\_state(self, state):  return any((node.state[0] == state[0]).all() for node in self.frontier)    def empty(self):  return len(self.frontier) == 0    def remove(self):  if self.empty():  raise Exception("Empty Frontier")  else:  node = self.frontier[-1]  self.frontier = self.frontier[:-1]  return node      class QueueFrontier(StackFrontier):  def remove(self):  if self.empty():  raise Exception("Empty Frontier")  else:  node = self.frontier[0]  self.frontier = self.frontier[1:]  return node      class Puzzle:  def \_\_init\_\_(self, start, startIndex, goal, goalIndex):  self.start = [start, startIndex]  self.goal = [goal, goalIndex]  self.solution = None    def neighbors(self, state):  mat, (row, col) = state  results = []    if row > 0:  mat1 = np.copy(mat)  mat1[row][col] = mat1[row - 1][col]  mat1[row - 1][col] = 0  results.append(('up', [mat1, (row - 1, col)]))  if col > 0:  mat1 = np.copy(mat)  mat1[row][col] = mat1[row][col - 1]  mat1[row][col - 1] = 0  results.append(('left', [mat1, (row, col - 1)]))  if row < 2:  mat1 = np.copy(mat)  mat1[row][col] = mat1[row + 1][col]  mat1[row + 1][col] = 0  results.append(('down', [mat1, (row + 1, col)]))  if col < 2:  mat1 = np.copy(mat)  mat1[row][col] = mat1[row][col + 1]  mat1[row][col + 1] = 0  results.append(('right', [mat1, (row, col + 1)]))    return results    def print(self):  solution = self.solution if self.solution is not None else None  print("Start State:\n", self.start[0], "\n")  print("Goal State:\n", self.goal[0], "\n")  print("\nStates Explored: ", self.num\_explored, "\n")  print("Solution:\n ")  for action, cell in zip(solution[0], solution[1]):  print("action: ", action, "\n", cell[0], "\n")  print("Goal Reached!!")    def does\_not\_contain\_state(self, state):  for st in self.explored:  if (st[0] == state[0]).all():  return False  return True    def solve(self):  self.num\_explored = 0    start = Node(state=self.start, parent=None, action=None)  frontier = QueueFrontier()  frontier.add(start)    self.explored = []    while True:  if frontier.empty():  raise Exception("No solution")    node = frontier.remove()  self.num\_explored += 1    if (node.state[0] == self.goal[0]).all():  actions = []  cells = []  while node.parent is not None:  actions.append(node.action)  cells.append(node.state)  node = node.parent  actions.reverse()  cells.reverse()  self.solution = (actions, cells)  return    self.explored.append(node.state)    for action, state in self.neighbors(node.state):  if not frontier.contains\_state(state) and self.does\_not\_contain\_state(state):  child = Node(state=state, parent=node, action=action)  frontier.add(child)      start = np.array([[1, 2, 3], [8, 0, 4], [7, 6, 5]])  goal = np.array([[2, 8, 1], [0, 4, 3], [7, 6, 5]])  startIndex = (1, 1)  goalIndex = (1, 0)  p = Puzzle(start, startIndex, goal, goalIndex)  p.solve()  p.print() |
| --- |