# **Practical 1**

Practical 1(a)

|  |
| --- |
| Write a program to implement depth first search algorithm. |
| graph={'A':set(['B','C']),  'B':set(['A','D','E',]),  'C':set(['A','F']),  'D':set(['B']),  'E':set(['B','F']),  'F':set(['C','E']) }  def dfs(graph,start):  visited,stack=set(),[start]  while stack:  vertex=stack.pop()  if vertex not in visited:  visited.add(vertex)  stack.extend(graph[vertex]-visited)  return visited  print(dfs(graph,'A'))  def dfs\_paths(graph,start,goal):  stack=[(start,[start])]  while stack:  (vertex,path)=stack.pop()  for next in graph[vertex] - set(path):  if next==goal:  yield path+[next]  else:  stack.append((next,path+[next]))  h=list(dfs\_paths(graph,'A','E'))  print(h) |
|  |

Practical 1(b)

|  |
| --- |
| Write a program to implement breadth first search algorithm. |
| graph={'A':set(['B','C']),  'B':set(['A','D','E']),  'C':set(['A','F']),  'D':set(['B']),  'E':set(['B','F']),  'F':set(['C','E'])}  def bfs(graph,start):  visited,queue=set(),[start]  while queue:  vertex=queue.pop(0)  if vertex not in visited:  visited.add(vertex)  queue.extend(graph[vertex]-visited)  return visited  print(bfs(graph,'C'))  def bfs\_paths(graph,start,goal):  queue=[(start,[start])]  while queue:  print(queue)  (vertex,path)=queue.pop(0)  for next in graph[vertex]-set(path):  if next==goal:  yield path+[next]  else:  queue.append((next,path+[next]))  print(list(bfs\_paths(graph,'A','F')))  def shortest\_path(graph,start,goal):  try:  return next(bfs\_paths(graph,start,goal))  except StopIteration:  return none  print(shortest\_path(graph,'A','F')) |
|  |

## Practical 2

Practical 2(a)

|  |
| --- |
| Write a program to simulate 4-Queen / N-Queen problem. |
| from math import \*  import sys  x={}  n=int(4)  def clear\_future\_blocks(k):  for i in range(k,n+1):  x[i]=None  def place(k,i):  if(i in x.values()):  return False  j=1  while(j<k):  if abs(x[j]-i)==abs(j-k):  return False  j+=1  return True  def NQueens(k):  for i in range(1,n+1):  clear\_future\_blocks(k)  if place(k,i):  x[k]=i  if(k==n):  for j in x:  print(x[j],end=' ')  for k in range(1,n+1):  if(k==x[j]):  print('Q',end=' ')  else:  print('.',end=' ')  print()  print('-------')  else:  NQueens(k+1)  NQueens(1) |
|  |

Practical 2(b)

|  |
| --- |
| Write a program to solve tower of Hanoi problem. |
| def moveTower(height,fromPole, toPole, withPole):  if height >= 2:  moveTower(height-1,fromPole,withPole,toPole)  moveDisk(fromPole,toPole)  moveTower(height-1,withPole,toPole,fromPole)  def moveDisk(fp,tp):  print("moving disk from",fp,"to",tp)  moveTower(4,"A","B","C") |
|  |

# Practical 5

Practical 5(a)

|  |
| --- |
| Write a program to solve water jug problem. |
| # 3 water jugs capacity -> (x,y,z) where x>y>z  # initial state (12,0,0)  # final state (6,5,1)  capacity = (12,8,5)  # Maximum capacities of 3 jugs -> x,y,z  x = capacity[0]  y = capacity[1]  z = capacity[2]  # to mark visited states memory is a dictionarycontaining key value pair.  memory = {}  # store solution path  ans = []  def get\_all\_states(state):  # Let the 3 jugs be called a,b,c  a = state[0]  b = state[1]  c = state[2]  if(a==6 and b==6):  ans.append(state)  return True  # if current state is already visited earlier  if((a,b,c) in memory):  return False  memory[(a,b,c)] = 1  #empty jug a  if(a>0):  #empty a into b  if(a+b<=y):  if( get\_all\_states((0,a+b,c)) ):  ans.append(state)  return True  else:  if( get\_all\_states((a-(y-b), y, c)) ):  ans.append(state)  return True  #empty a into c  if(a+c<=z):  if( get\_all\_states((0,b,a+c)) ):  ans.append(state)  return True  else:  if( get\_all\_states((a-(z-c), b, z)) ):  ans.append(state)  return True  #empty jug b  if(b>0):  #empty b into a  if(a+b<=x):  if( get\_all\_states((a+b, 0, c)) ):  ans.append(state)  return True  else:  if( get\_all\_states((x, b-(x-a), c)) ):  ans.append(state)  return True  #empty b into c  if(b+c<=z):  if( get\_all\_states((a, 0, b+c)) ):  ans.append(state)  return True  else:  if( get\_all\_states((a, b-(z-c), z)) ):  ans.append(state)  return True  #empty jug c  if(c>0):  #empty c into a  if(a+c<=x):  if( get\_all\_states((a+c, b, 0)) ):  ans.append(state)  return True  else:  if( get\_all\_states((x, b, c-(x-a))) ):  ans.append(state)  return True  #empty c into b  if(b+c<=y):  if( get\_all\_states((a, b+c, 0)) ):  ans.append(state)  return True  else:  if( get\_all\_states((a, y, c-(y-b))) ):  ans.append(state)  return True  return False  initial\_state = (12,0,0)  print("Starting work...\n")  get\_all\_states(initial\_state)  ans.reverse()  for i in ans:  print(i) |
|  |

Practical 5(b)

|  |
| --- |
| Design the simulation of tic – tac – toe game using min-max algorithm. |
| import os  import time  board = [' ',' ',' ',' ',' ',' ',' ',' ',' ',' ']  player = 1  ########win Flags##########  Win = 1  Draw = -1  Running = 0  Stop = 1  ###########################  Game = Running  Mark = 'X'  def DrawBoard():  print(" %c | %c | %c " % (board[1],board[2],board[3]))  print("\_\_\_|\_\_\_|\_\_\_")  print(" %c | %c | %c " % (board[4],board[5],board[6]))  print("\_\_\_|\_\_\_|\_\_\_")  print(" %c | %c | %c " % (board[7],board[8],board[9]))  print(" | | ")  def CheckPosition(x):  if(board[x] == ' '):  return True  else:  return False  def CheckWin():  global Game  if(board[1] == board[2] and board[2] == board[3] and board[1] != ' '):  Game = Win  elif(board[4] == board[5] and board[5] == board[6] and board[4] != ' '):  Game = Win  elif(board[7] == board[8] and board[8] == board[9] and board[7] != ' '):  Game = Win  elif(board[1] == board[4] and board[4] == board[7] and board[1] != ' '):  Game = Win  elif(board[2] == board[5] and board[5] == board[8] and board[2] != ' '):  Game = Win  elif(board[3] == board[6] and board[6] == board[9] and board[3] != ' '):  Game=Win  elif(board[1] == board[5] and board[5] == board[9] and board[5] != ' '):  Game = Win  elif(board[3] == board[5] and board[5] == board[7] and board[5] != ' '):  Game=Win  elif(board[1]!=' ' and board[2]!=' ' and board[3]!=' ' and board[4]!=' ' and board[5]!=' ' and board[6]!=' ' and board[7]!=' ' and board[8]!=' ' and board[9]!=' '):  Game=Draw  else:  Game=Running  print("Tic-Tac-Toe Game")  print("Player 1 [X] --- Player 2 [O]\n")  print()  print()  print("Please Wait...")  time.sleep(1)  while(Game == Running):  os.system('cls')  DrawBoard()  if(player % 2 != 0):  print("Player 1's chance")  Mark = 'X'  else:  print("Player 2's chance")  Mark = 'O'  choice = int(input("Enter the position between [1-9] where you want to mark :"))  if(CheckPosition(choice)):  board[choice] = Mark  player+=1  CheckWin()  os.system('cls')  DrawBoard()  if(Game==Draw):  print("Game Draw")  elif(Game==Win):  player-=1  if(player%2!=0):  print("Player 1 Won")  else:  print("Player 2 Won") |
|  |

# Practical 6

Practical 6(a)

|  |
| --- |
| Write a program to solve Missionaries and Cannibals problem. |
| import math  class State():  def \_\_init\_\_(self, cannibalLeft, missionaryLeft, boat, cannibalRight, missionaryRight):  self.cannibalLeft = cannibalLeft  self.missionaryLeft = missionaryLeft  self.boat = boat  self.cannibalRight = cannibalRight  self.missionaryRight = missionaryRight  self.parent = None  def is\_goal(self):  if self.cannibalLeft == 0 and self.missionaryLeft == 0:  return True  else:  return False  def is\_valid(self):  if self.missionaryLeft >= 0 and self.missionaryRight >= 0 and self.cannibalLeft >= 0 and self.cannibalRight >= 0 and (self.missionaryLeft == 0 or self.missionaryLeft >= self.cannibalLeft) and (self.missionaryRight == 0 or self.missionaryRight >= self.cannibalRight):  return True  else:  return False  def \_\_eq\_\_(self, other):  return self.cannibalLeft == other.cannibalLeft and self.missionaryLeft == other.missionaryLeft and self.boat == other.boat and self.cannibalRight == other.cannibalRight and self.missionaryRight == other.missionaryRight  def \_\_hash\_\_(self):  return hash((self.cannibalLeft, self.missionaryLeft, self.boat, self.cannibalRight, self.missionaryRight))  @staticmethod  def successors(cur\_state):  children = []  if cur\_state.boat == 'left':  new\_state = State(cur\_state.cannibalLeft, cur\_state.missionaryLeft - 2, 'right', cur\_state.cannibalRight, cur\_state.missionaryRight + 2)  if new\_state.is\_valid():  new\_state.parent = cur\_state  children.append(new\_state)  new\_state = State(cur\_state.cannibalLeft - 2, cur\_state.missionaryLeft, 'right', cur\_state.cannibalRight + 2, cur\_state.missionaryRight)  if new\_state.is\_valid():  new\_state.parent = cur\_state  children.append(new\_state)  new\_state = State(cur\_state.cannibalLeft - 1, cur\_state.missionaryLeft - 1, 'right', cur\_state.cannibalRight + 1, cur\_state.missionaryRight + 1)  if new\_state.is\_valid():  new\_state.parent = cur\_state  children.append(new\_state)  new\_state = State(cur\_state.cannibalLeft, cur\_state.missionaryLeft - 1, 'right', cur\_state.cannibalRight, cur\_state.missionaryRight + 1)  if new\_state.is\_valid():  new\_state.parent = cur\_state  children.append(new\_state)  new\_state = State(cur\_state.cannibalLeft - 1, cur\_state.missionaryLeft, 'right', cur\_state.cannibalRight + 1, cur\_state.missionaryRight)  if new\_state.is\_valid():  new\_state.parent = cur\_state  children.append(new\_state)  else:  new\_state = State(cur\_state.cannibalLeft, cur\_state.missionaryLeft + 2, 'left', cur\_state.cannibalRight, cur\_state.missionaryRight - 2)  if new\_state.is\_valid():  new\_state.parent = cur\_state  children.append(new\_state)  new\_state = State(cur\_state.cannibalLeft + 2, cur\_state.missionaryLeft, 'left', cur\_state.cannibalRight - 2, cur\_state.missionaryRight)  if new\_state.is\_valid():  new\_state.parent = cur\_state  children.append(new\_state)  new\_state = State(cur\_state.cannibalLeft + 1, cur\_state.missionaryLeft + 1, 'left', cur\_state.cannibalRight - 1, cur\_state.missionaryRight - 1)  if new\_state.is\_valid():  new\_state.parent = cur\_state  children.append(new\_state)  new\_state = State(cur\_state.cannibalLeft, cur\_state.missionaryLeft + 1, 'left', cur\_state.cannibalRight, cur\_state.missionaryRight - 1)  if new\_state.is\_valid():  new\_state.parent = cur\_state  children.append(new\_state)  new\_state = State(cur\_state.cannibalLeft + 1, cur\_state.missionaryLeft, 'left', cur\_state.cannibalRight - 1, cur\_state.missionaryRight)  if new\_state.is\_valid():  new\_state.parent = cur\_state  children.append(new\_state)  return children  def breadth\_first\_search():  initial\_state = State(3, 3, 'left', 0, 0)  if initial\_state.is\_goal():  return initial\_state  frontier = list()  explored = set()  frontier.append(initial\_state)  while frontier:  state = frontier.pop(0)  if state.is\_goal():  return state  explored.add(state)  children = State.successors(state)  for child in children:  if child not in explored and child not in frontier:  frontier.append(child)  return None  def print\_solution(solution):  path = []  path.append(solution)  parent = solution.parent  while parent:  path.append(parent)  parent = parent.parent  for t in range(len(path)):  state = path[len(path) - t - 1]  print("(" + str(state.cannibalLeft) + "," + str(state.missionaryLeft) + "," + state.boat + "," + str(state.cannibalRight) + "," + str(state.missionaryRight) + ")")  def main():  solution = breadth\_first\_search()  print("missionaries and cannibals solution:")  print("(cannibal Left ,Missionary Left,boat,cannibal Right,Missionary Right)")  if solution:  print\_solution(solution)  else:  print("No solution found.")  if \_\_name\_\_ == "\_\_main\_\_":  main() |
|  |

Practical 6(b)

|  |
| --- |
| Design an application to simulate number puzzle problem. |
| from \_\_future\_\_ import print\_function  from simpleai.search import astar, SearchProblem  from simpleai.search.viewers import WebViewer  GOAL = '''1-2-3  4-5-6  7-8-e'''    INITIAL = '''4-1-2  7-3-e  8-5-6'''  def list\_to\_string(list\_):  return '\n'.join(['-'.join(row) for row in list\_])  def string\_to\_list(string\_):  return [row.split('-') for row in string\_.split('\n')]  def find\_location(rows, element\_to\_find):  '''Find the location of a piece in the puzzle. Returns a tuple: row, column'''  for ir, row in enumerate(rows):  for ic, element in enumerate(row):  if element == element\_to\_find:  return ir, ic  # we create a cache for the goal position of each piece, so we don't have to  # recalculate them every time  goal\_positions = {}  rows\_goal = string\_to\_list(GOAL)  for number in '12345678e':  goal\_positions[number] = find\_location(rows\_goal, number)  class EigthPuzzleProblem(SearchProblem):  def actions(self, state):  '''Returns a list of the pieces we can move to the empty space.'''  rows = string\_to\_list(state)  row\_e, col\_e = find\_location(rows, 'e')  actions = []  if row\_e > 0:  actions.append(rows[row\_e - 1][col\_e])  if row\_e < 2:  actions.append(rows[row\_e + 1][col\_e])  if col\_e > 0:  actions.append(rows[row\_e][col\_e - 1])  if col\_e < 2:  actions.append(rows[row\_e][col\_e + 1])  return actions  def result(self, state, action):  '''Return the resulting state after moving a piece to the empty space. (the "action" parameter contains the piece to move)'''  rows = string\_to\_list(state)  row\_e, col\_e = find\_location(rows, 'e')  row\_n, col\_n = find\_location(rows, action)  rows[row\_e][col\_e], rows[row\_n][col\_n] = rows[row\_n][col\_n],rows[row\_e][col\_e]  return list\_to\_string(rows)  def is\_goal(self, state):  '''Returns true if a state is the goal state.'''  return state == GOAL  def cost(self, state1, action, state2):  '''Returns the cost of performing an action. No useful on this problem, ibut needed.'''  return 1  def heuristic(self, state):  '''Returns an \*estimation\* of the distance from a state to the goal. We are using the manhattan distance.'''  rows = string\_to\_list(state)  distance = 0  for number in '12345678e':  row\_n, col\_n = find\_location(rows, number)  row\_n\_goal, col\_n\_goal = goal\_positions[number]  distance += abs(row\_n - row\_n\_goal) + abs(col\_n - col\_n\_goal)  return distance  result = astar(EigthPuzzleProblem(INITIAL))  for action, state in result.path():  print('Move number', action)  print(state) |
|  |

# Practical 7

|  |
| --- |
| Write a program to shuffle Deck of cards. |
| #first let's import random procedures since we will be shuffling  import random  #next, let's start building list holders so we can place our cards in there:  cardfaces = []  suits = ["Hearts", "Diamonds", "Clubs", "Spades"]  royals = ["J", "Q", "K", "A"]  deck = []  #now, let's start using loops to add our content:  for i in range(2,11):  cardfaces.append(str(i)) #this adds numbers 2-10 and converts them to string data  for j in range(4):  cardfaces.append(royals[j]) #this will add the royal faces to the cardbase  for k in range(4):  for l in range(13):  card = (cardfaces[l] + " of " + suits[k])  #this makes each card, cycling through suits, but first through faces  deck.append(card)  #this adds the information to the "full deck" we want to make  #now let's shuffle our deck!  random.shuffle(deck)  #now let's see the cards!  for m in range(52):  print(deck[m]) |
|  |

# Practical 8

# Solve constraint satisfaction problem

Practical 8(b)

|  |
| --- |
| Map coloring |
| Open any editor and write this code:  /\* prolog tutorial 2.1 Map Colorings \*/  adjacent(1,2). adjacent(2,1).  adjacent(1,3). adjacent(3,1).  adjacent(1,4). adjacent(4,1).  adjacent(1,5). adjacent(5,1).  adjacent(2,3). adjacent(3,2).  adjacent(2,4). adjacent(4,2).  adjacent(3,4). adjacent(4,3).  adjacent(4,5). adjacent(5,4).  /\*------------------------------------\*/  color(1,red,a). color(1,red,b).  color(2,blue,a). color(2,blue,b).  color(3,green,a). color(3,green,b).  color(4,yellow,a). color(4,blue,b).  color(5,blue,a). color(5,green,b).  /\*------------------------------------\*/  conflict(Coloring) :-  adjacent(X,Y),  color(X,Color,Coloring),  color(Y,Color,Coloring).  /\*-------------------------------------\*/  conflict(R1,R2,Coloring) :-  adjacent(R1,R2),  color(R1,Color,Coloring),  color(R2,Color,Coloring).  And save it with name of mapcoloring.pl  Then go to prolog and type:  ?-[c:\path\to\mapcoloring].  ?-conflict(a)  ?-color(1,2,a).  ?-color(1,X,Y). |
|  |

Practical 8(c)

|  |
| --- |
| Zebra puzzle |
| Open any editor and type:  kid(ethan).  kid(ali).  kid(anya).  hero(spiderman).  hero(captain\_america).  hero(iron\_man).  age(six).  age(eight).  age(ten).  relation(K,H,A):- K=anya, H=spiderman, age(A).  relation(K,H,A):- K=ethan, hero(H), age(A), H\=captain\_america.  relation(K,H,A):- kid(K), H=spiderman, A=six.  relation(K,H,A):- kid(K), H=captain\_america, age(A).  relation(K,captain\_america,A):- kid(K),age(A).  different(X,Y,Z):-X\=Y,X\=Z,Y\=Z.  solve(K1,K2,K3,H1,H2,H3,A1,A2,A3):- relation(K1,H1,A1),relation(K2,H2,A2),relation(K3,H3,A3),different(K1,K2,K3),different(H1,H2,H3),different(A1,A2,A3).  Then go to prolog and type  ?- [zebrapuzzle].  ?- solve(K1,K2,K3,H1,H,H3,A1,A2,A3). |
|  |

Practical 8(d)

|  |
| --- |
| Magic squares |
| def generate\_square(n):  if n % 2 == 0:  raise ValueError("Siamese method only works for odd n.")  # Create empty n x n grid  mat = [[0] \* n for \_ in range(n)]  # Start from middle of first row  i, j = 0, n // 2  num = 1  while num <= n \* n:  mat[i][j] = num  num += 1  # Tentative next position: one up, one right (wrap around)  ni, nj = (i - 1) % n, (j + 1) % n  if mat[ni][nj] != 0: # If occupied, move down instead  i = (i + 1) % n  else:  i, j = ni, nj  return mat  # Example usage  n = 5  magic\_square = generate\_square(n)  magic\_sum = n \* (n \* n + 1) // 2  #print(f"{n}x{n} Magic Square (Magic Sum = {magic\_sum}):\n")  for row in magic\_square:  print(" ".join(f"{x:2}" for x in row)) |
|  |