# Artificial Intelligence

# Lab 03

## Question 01:

- **Lab 03 Task 01 – The Tic Tac Toe Problem:**

```
#!/usr/bin/env python
# coding: utf-8

# In[2]:

# Tic Tac Toe
# code taken from https://inventwithpython.com/chapter10.html

import random

def drawBoard(board):
    # This function prints out the board that it was passed.

    # "board" is a list of 10 strings representing the board (ignore index 0)
    print('   |   |')
    print(' ' + board[7] + ' | ' + board[8] + ' | ' + board[9])
    print('   |   |')
    print('-----------')
    print('   |   |')
    print(' ' + board[4] + ' | ' + board[5] + ' | ' + board[6])
    print('   |   |')
    print('-----------')
    print('   |   |')
    print(' ' + board[1] + ' | ' + board[2] + ' | ' + board[3])
    print('   |   |')

def inputcomputer2Letter():
    # Lets the player type which letter they want to be.
    # Returns a list with the player's letter as the first item, and the computer's letter as the second.
    letter = 'X'
    # the first element in the tuple is the player's letter, the second is the computer's letter.
    if letter == 'X':
        return ['X', 'O']
    else:
        return ['O', 'X']

def whoGoesFirst():
    # Randomly choose the player who goes first.
```

```
    if random.randint(0, 1) == 0:
        return 'computer1'
    else:
        return 'computer2'

def playAgain():
    # This function returns True if the player wants to play again, otherwise it returns False.
    print('Do you want to play again? (yes or no)')
    return input().lower().startswith('y')

def makeMove(board, letter, move):
    # This function simply marks the planned move (location of the borad) with the player's letter.
    board[move] = letter

def isWinner(bo, le):
    # Given a board and a player's letter, this function returns True if that player has won.
    # We use bo instead of board and le instead of letter so we don't have to type as much.
    return ((bo[7] == le and bo[8] == le and bo[9] == le) or # across the top
    (bo[4] == le and bo[5] == le and bo[6] == le) or # across the middle
    (bo[1] == le and bo[2] == le and bo[3] == le) or # across the bottom
    (bo[7] == le and bo[4] == le and bo[1] == le) or # down the left side
    (bo[8] == le and bo[5] == le and bo[2] == le) or # down the middle
    (bo[9] == le and bo[6] == le and bo[3] == le) or # down the right side
    (bo[7] == le and bo[5] == le and bo[3] == le) or # diagonal
    (bo[9] == le and bo[5] == le and bo[1] == le)) # diagonal

def getBoardCopy(board):
    # Make a duplicate of the board list and return it the duplicate.
    dupeBoard = []

    for i in board:
        dupeBoard.append(i)

    return dupeBoard

def isSpaceFree(board, move):
    # Return true if the passed move is free on the passed board.
    return board[move] == ' '

```

```
def getPlayerMove(board):
    # Given a board and the computer's letter, determine where to move and return that move.
    if computer1Letter == 'X':
        computer2Letter = 'O'
    else:
        computer2Letter = 'O'

    # Here is our algorithm for our Tic Tac Toe AI:
    # First, check if we can win in the next move
    for i in range(1, 10):
        copy = getBoardCopy(board)
        if isSpaceFree(copy, i):
            makeMove(copy, computer1Letter, i)
            if isWinner(copy, computer1Letter):
                return i

    # Check if the player could win on his next move, and block them.
    for i in range(1, 10):
        copy = getBoardCopy(board)
        if isSpaceFree(copy, i):
            makeMove(copy, computer2Letter, i)
            if isWinner(copy, computer2Letter):
                return i

    # Try to take one of the corners, if they are free.
    move = chooseRandomMoveFromList(board, [1, 3, 7, 9])
    if move != None:
        return move

    # Try to take the center, if it is free.
    if isSpaceFree(board, 5):
        return 5

    # Move on one of the sides.
    return chooseRandomMoveFromList(board, [2, 4, 6, 8])

def chooseRandomMoveFromList(board, movesList):
    # Returns a valid move from the passed list on the passed board.
    # Returns None if there is no valid move.
    possibleMoves = []
    for i in movesList:
        if isSpaceFree(board, i):
            possibleMoves.append(i)

    if len(possibleMoves) != 0:
        return random.choice(possibleMoves)
    else:
```

```
        return None

def getComputerMove(board, computer1Letter):
    # Given a board and the computer's letter, determine where to move and return that move.
    if computer1Letter == 'X':
        computer2Letter = 'O'
    else:
        computer2Letter = 'X'

    # Here is our algorithm for our Tic Tac Toe AI:
    # First, check if we can win in the next move
    for i in range(1, 10):
        copy = getBoardCopy(board)
        if isSpaceFree(copy, i):
            makeMove(copy, computer1Letter, i)
            if isWinner(copy, computer1Letter):
                return i

    # Check if the player could win on his next move, and block them.
    for i in range(1, 10):
        copy = getBoardCopy(board)
        if isSpaceFree(copy, i):
            makeMove(copy, computer2Letter, i)
            if isWinner(copy, computer2Letter):
                return i

    # Try to take one of the corners, if they are free.
    move = chooseRandomMoveFromList(board, [1, 3, 7, 9])
    if move != None:
        return move

    # Try to take the center, if it is free.
    if isSpaceFree(board, 5):
        return 5

    # Move on one of the sides.
    return chooseRandomMoveFromList(board, [2, 4, 6, 8])

def isBoardFull(board):
    # Return True if every space on the board has been taken. Otherwise return False.
    for i in range(1, 10):
        if isSpaceFree(board, i):
            return False
    return True

print('Welcome to Tic Tac Toe!')
```

```
while True:
    # Reset the board
    theBoard = [' '] * 10
    computer2Letter, computer1Letter = inputcomputer2Letter()
    turn = whoGoesFirst()
    print('The ' + turn + ' will go first')
    gameIsPlaying = True

    while gameIsPlaying:
        if turn == 'computer2':
            # Player's turn.
            drawBoard(theBoard)
            move = getPlayerMove(theBoard)
            makeMove(theBoard, computer2Letter, move)

            if isWinner(theBoard, computer2Letter):
                drawBoard(theBoard)
                print('computer2 have won the game!')
                gameIsPlaying = False
            else:
                if isBoardFull(theBoard):
                    drawBoard(theBoard)
                    print('The game is a tie!')
                    break
                else:
                    turn = 'computer1'

        else:
            # Computer's turn.
            move = getComputerMove(theBoard, computer1Letter)
            makeMove(theBoard, computer1Letter, move)

            if isWinner(theBoard, computer1Letter):
                drawBoard(theBoard)
                print('The computer1 has beaten computer2!')
                gameIsPlaying = False
            else:
                if isBoardFull(theBoard):
                    drawBoard(theBoard)
                    print('The game is a tie!')
                    break
                else:
                    turn = 'computer2'

    if not playAgain():
        break
```

- **Lab 03 Task 02 – Vacuum Cleaner Agent:**

```
import random

class Environment(object):
    def __init__(self):
        # instantiate locations and conditions
        # 0 indicates Clean and 1 indicates Dirty
        self.Location = {'A':random.randint(0, 1) , 'B':random.randint(0, 1),'
C':random.randint(0, 1) }

        # randomize conditions in locations A and B

class SimpleReflexVacuumAgent(Environment):
    def __init__(self, Environment):
        print (Environment.Location)
        # Instantiate performance measurement
        Score = 0
        # place vacuum at random location
        vacuumLocation = random.randint(0, 1)
        # if vacuum at ALocation
        if vacuumLocation == 0:
            print ("Vacuum is randomly placed at Location A.")
            # and Location A is Dirty.
            if Environment.Location['A'] == 1:
                print ("Location A is Dirty.")
                # suck the dirt  and mark it clean
                Environment.Location['A'] = 0;
                Score += 1
                print ("Location A has been Cleaned.")
                # move to B
                print ("Moving to Location B...")
                Score -= 1
                # if B is Dirty
                if Environment.Location['B'] == 1:
                    print ("Location B is Dirty.")
                    # suck and mark clean
                    Environment.Location['B'] = 0;
                    Score += 1
                    print ("Location B has been Cleaned.")
                if Environment.Location['C'] == 1:
                    print ("Location c is Dirty.")
                    # suck and mark clean
                    Environment.Location['C'] = 0;
                    Score += 1
                    print ("Location C has been Cleaned.")

            elif Environment.Location['B'] == 1:
                # move to B
```

```
            Score -= 1
            print ("Moving to Location B...")
            # if B is Dirty
            if Environment.Location['B'] == 1:
                print ("Location B is Dirty.")
                # suck and mark clean
                Environment.Location['B'] = 0;
                Score += 1
                print ("Location B has been Cleaned.")
            # move to C
            Score -= 1
            print ("Moving to Location C...")
            # if C is Dirty
            if Environment.Location['C'] == 1:
                print ("Location C is Dirty.")
                # suck and mark clean
                Environment.Location['C'] = 0;
                Score += 1
                print ("Location C has been Cleaned.")
        elif Environment.Location['C'] == 1:
            # move to C
            Score -= 1
            print ("Moving to Location C...")
            # if C is Dirty
            if Environment.Location['C'] == 1:
                print ("Location C is Dirty.")
                # suck and mark clean
                Environment.Location['C'] = 0;
                Score += 1
                print ("Location C has been Cleaned.")

    elif vacuumLocation == 1:
        print ("Vacuum randomly placed at Location A.")
        # and A is Dirty
        if Environment.Location['A'] == 1:
            print ("Location A is Dirty.")
            # suck and mark clean
            Environment.Location['A'] = 0;
            Score += 1
            print ("Location A has been Cleaned.")
            # move to A
            Score -= 1
            print ("Moving to Location B...")
            # if b is Dirty
            if Environment.Location['B'] == 1:
                print ("Location B is Dirty.")
                # suck and mark clean
                Environment.Location['B'] = 0;
```

```
                    Score += 1
                    print ("Location B has been Cleaned.")
                #if C is Dirty
                if Environment.Location['C'] == 1:
                    print ("Location C is Dirty.")
                    # suck and mark clean
                    Environment.Location['C'] = 0;
                    Score += 1
                    print ("Location C has been Cleaned.")
            elif Environment.Location['B'] == 1:
                print ("Location C is Dirty.")
                # suck and mark clean
                Environment.Location['B'] = 0;
                Score += 1
                print ("Location B has been Cleaned.")
                Score -= 1
            elif Environment.Location['C'] == 1:
                # move to C
                print ("Moving to Location C...")
                Score -= 1
                # if A is Dirty
                if Environment.Location['C'] == 1:
                    print ("Location C is Dirty.")
                    # suck and mark clean
                    Environment.Location['C'] = 0;
                    Score += 1
                    print ("Location C has been Cleaned.")
        # done cleaning
        print (Environment.Location)


theEnvironment = Environment()
theVacuum = SimpleReflexVacuumAgent(theEnvironment)
```

- **Lab 04 Task 01 – 8 Puzzle Problem:**

```python
from copy import deepcopy
import numpy as np
import time

# takes the input of current states and evaluvates the best path to goal state
def bestsolution(state):
    bestsol = np.array([], int).reshape(-1, 9)
    count = len(state) - 1
    while count != -1:
        bestsol = np.insert(bestsol, 0, state[count]['puzzle'], 0)
        count = (state[count]['parent'])
    return bestsol.reshape(-1, 3, 3)


# this function checks for the uniqueness of the iteration(it) state, weather it has been previously traversed or not.
def all(checkarray):
    set=[]
    for it in set:
        for checkarray in it:
            return 1
        else:
            return 0

# calculate Manhattan distance cost between each digit of puzzle(start state) and the goal state
def manhattan(puzzle, goal):
    a = abs(puzzle // 3 - goal // 3)
    b = abs(puzzle % 3 - goal % 3)
    mhcost = a + b
    return sum(mhcost[1:])



# will calcuates the number of misplaced tiles in the current state as compared to the goal state
def misplaced_tiles(puzzle,goal):
    mscost = np.sum(puzzle != goal) - 1
    return mscost if mscost > 0 else 0


#3[on_true] if [expression] else [on_false]

# will indentify the coordinates of each of goal or initial state values

```

```
def coordinates(puzzle):
    pos = np.array(range(9))
    for p, q in enumerate(puzzle):
        pos[q] = p
    return pos


# start of 8 puzzle evaluvation, using Manhattan heuristics
def evaluvate(puzzle, goal):
    steps = np.array([('up', [0, 1, 2], -
3),('down', [6, 7, 8],  3),('left', [0, 3, 6], -
1),('right', [2, 5, 8],  1)],
                dtype =  [('move',  str, 1),('position', list),('head',
 int)])

    dtstate = [('puzzle',  list),('parent', int),('gn',  int),('hn',  i
nt)]

     # initializing the parent, gn and hn, where hn is manhattan distan
ce function call
    costg = coordinates(goal)
    parent = -1
    gn = 0
    hn = manhattan(coordinates(puzzle), costg)
    state = np.array([(puzzle, parent, gn, hn)], dtstate)

# We make use of priority queues with position as keys and fn as value.
    dtpriority = [('position', int),('fn', int)]
    priority = np.array( [(0, hn)], dtpriority)


    while 1:
        priority = np.sort(priority, kind='mergesort', order=['fn', 'po
sition'])
        position, fn = priority[0]

        priority = np.delete(priority, 0, 0)
        # sort priority queue using merge sort,the first element is pic
ked for exploring remove from queue what we are exploring

        puzzle, parent, gn, hn = state[position]
        puzzle = np.array(puzzle)
        # Identify the blank square in input
        blank = int(np.where(puzzle == 0)[0])
        gn = gn + 1
        c = 1
        start_time = time.time()
        for s in steps:
```

```
            c = c + 1
            if blank not in s['position']:
                # generate new state as copy of current
                openstates = deepcopy(puzzle)
                openstates[blank], openstates[blank + s['head']] = openstates[blank + s['head']], openstates[blank]
                # The all function is called, if the node has been previously explored or not
                if ~(np.all(list(state['puzzle']) == openstates, 1)).any():
                    end_time = time.time()
                    if (( end_time - start_time ) > 2):
                        print(" The 8 puzzle is unsolvable ! \n")
                        exit
                    # calls the manhattan function to calcuate the cost

                    hn = manhattan(coordinates(openstates), costg)
                     # generate and add new state in the list

                    q = np.array([(openstates, position, gn, hn)], dtstate)
                    state = np.append(state, q, 0)
                    # f(n) is the sum of cost to reach node and the cost to rech fromt he node to the goal state
                    fn = gn + hn


                    q = np.array([(len(state) - 1, fn)], dtpriority)

                    priority = np.append(priority, q, 0)
                      # Checking if the node in openstates are matching the goal state.
                    if np.array_equal(openstates, goal):

                        print(' The 8 puzzle is solvable ! \n')
                        return state, len(priority)


    return state, len(priority)

# start of 8 puzzle evaluvation, using Misplaced tiles heuristics
def evaluvate_misplaced(puzzle, goal):
    steps = np.array([('up', [0, 1, 2], -3),('down', [6, 7, 8],  3),('left', [0, 3, 6], -1),('right', [2, 5, 8],  1)],
                dtype =  [('move',  str, 1),('position', list),('head', int)])

```

```
    dtstate = [('puzzle',  list),('parent', int),('gn',  int),('hn',  int)]

    costg = coordinates(goal)
    # initializing the parent, gn and hn, where hn is misplaced_tiles function call
    parent = -1
    gn = 0
    hn = misplaced_tiles(coordinates(puzzle), costg)
    state = np.array([(puzzle, parent, gn, hn)], dtstate)

   # We make use of priority queues with position as keys and fn as value.
    dtpriority = [('position', int),('fn', int)]

    priority = np.array([(0, hn)], dtpriority)

    while 1:
        priority = np.sort(priority, kind='mergesort', order=['fn', 'position'])
        position, fn = priority[0]
        # sort priority queue using merge sort,the first element is picked for exploring.
        priority = np.delete(priority, 0, 0)
        puzzle, parent, gn, hn = state[position]
        puzzle = np.array(puzzle)
         # Identify the blank square in input
        blank = int(np.where(puzzle == 0)[0])
        # Increase cost g(n) by 1
        gn = gn + 1
        c = 1
        start_time = time.time()
        for s in steps:
            c = c + 1
            if blank not in s['position']:
                 # generate new state as copy of current
                openstates = deepcopy(puzzle)
                openstates[blank], openstates[blank + s['head']] = openstates[blank + s['head']], openstates[blank]
                # The check function is called, if the node has been previously explored or not.
                if ~(np.all(list(state['puzzle']) == openstates, 1)).any():
                    end_time = time.time()
                    if (( end_time - start_time ) > 2):
                        print(" The 8 puzzle is unsolvable \n")
                        break
```

```
•                       # calls the Misplaced_tiles function to calcuate th
    e cost
•                       hn = misplaced_tiles(coordinates(openstates), costg
    )
•                       # generate and add new state in the list

•                       q = np.array([(openstates, position, gn, hn)], dtst
    ate)
•                       state = np.append(state, q, 0)
•                       # f(n) is the sum of cost to reach node and the cos
    t to rech fromt he node to the goal state
•                       fn = gn + hn

•

•                       q = np.array([(len(state) - 1, fn)], dtpriority)
•                       priority = np.append(priority, q, 0)
•                       # Checking if the node in openstates are matching t
    he goal state.
•                       if np.array_equal(openstates, goal):

•                           print(' The 8 puzzle is solvable \n')
•                           return state, len(priority)
•

•       return state, len(priority)
•


•   # ----------  Program start -----------------
•
•    # User input for initial state
•   puzzle = []
•   print(" Input vals from 0-8 for start state ")
•   for i in range(0,9):
•       x = int(input("enter vals :"))
•       puzzle.append(x)
•
•    # User input of goal state
•   goal = []
•   print(" Input vals from 0-8 for goal state ")
•   for i in range(0,9):
•       x = int(input("Enter vals :"))
•       goal.append(x)
•


•   n = int(input("1. Manhattan distance \n2. Misplaced tiles"))
•
•   if(n ==1 ):
•       state, visited = evaluvate(puzzle, goal)
•       bestpath = bestsolution(state)
```

```
    print(str(bestpath).replace('[', ' ').replace(']', ''))
    totalmoves = len(bestpath) - 1
    print('Steps to reach goal:',totalmoves)
    visit = len(state) - visited
    print('Total nodes visited: ',visit, "\n")
    print('Total generated:', len(state))

if(n == 2):
    state, visited = evaluvate_misplaced(puzzle, goal)
    bestpath = bestsolution(state)
    print(str(bestpath).replace('[', ' ').replace(']', ''))
    totalmoves = len(bestpath) - 1
    print('Steps to reach goal:',totalmoves)
    visit = len(state) - visited
    print('Total nodes visited: ',visit, "\n")
    print('Total generated:', len(state))
```

# Lab 4: Task02

C: > Users > Saad > Downloads > Untitled-1.py > ...

```
import collections
def bfs(graph,root):
    visited=set()
    queue=collections.deque([root])
    visited.add(root)
    while queue:
        vertex=queue.popleft()
        print(str(vertex))

        for neighbour in graph[vertex]:
            if neighbour not in visited:
                visited.add(neighbour)
                queue.append(neighbour)
graph = {
    'A': ['B','D'],
    'B': ['C','E'],
    'C': [],
    'D': ['G','H','E'],
    'E': ['F','C'],
    'F': [],
    'G': ['H'],
    'H': [],
}
bfs(graph,'A')
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

```
C
E
G
H
F
```

# Question 02:

- **BFS with a goal state:**
- **DFS with a goal state:**
- **DLS Algorithm:**
- **IDS Algorithm:**

```
BFS

1
In [1]:
graph = {
    'A' : ['B','C'],
  'B' : ['C', 'A','D'],
  'C' : ['A','E','B'],
  'D' : ['E','B','F'],
  'E' : ['C','D','F'],
  'F' : ['D','E'],
}


visited = [] # List to keep track of visited nodes.
queue = []     #Initialize a queue

def bfs(visited, graph, node):
    visited.append(node)
    queue.append(node)
    while queue:
        s = queue.pop(0)
        print (s, end = " ")
        for neighbour in graph[s]:
            if neighbour not in visited:
                visited.append(neighbour)
                queue.append(neighbour)

# Driver Code
bfs(visited, graph, 'B')
20.009 seconds
Output:

B C A D E F
DFS
3
In [2]:
# Using a Python dictionary to act as an adjacency list
graph = {
    'A' : ['B','C'],
  'B' : ['C', 'A','D'],
```

```
  'C' : ['A','E','B'],
  'D' : ['E','B','F'],
  'E' : ['C','D','F'],
  'F' : ['D','E'],
}
visited = set() # Set to keep track of visited nodes.

def dfs(visited, graph, node):
    if node not in visited:
        print (node)
        visited.add(node)
        for neighbour in graph[node]:
            dfs(visited, graph, neighbour)

# Driver Code
dfs(visited, graph, 'A')
40.009 seconds
Output:

A
B
C
E
D
F
In [21]:
## DLS AND IDFS

In [3]:
from collections import defaultdict

# This class represents a directed graph using adjacency
# list representation
class Graph:

    def __init__(self,vertices):

        # No. of vertices
        self.V = vertices

        # default dictionary to store graph
        self.graph = defaultdict(list)

    # function to add an edge to graph
    def addEdge(self,u,v):
        self.graph[u].append(v)

    # A function to perform a Depth-Limited search
```

```
    # from given source 'src'
    def DLS(self,src,target,maxDepth):

        if src == target : return True

        # If reached the maximum depth, stop recursing.
        if maxDepth <= 0 : return False

        # Recur for all the vertices adjacent to this vertex
        for i in self.graph[src]:
                if(self.DLS(i,target,maxDepth-1)):
                    return True
        return False

    # IDDFS to search if target is reachable from v.
    # It uses recursive DLS()
    def IDDFS(self,src, target, maxDepth):

        # Repeatedly depth-limit search till the
        # maximum depth
        for i in range(maxDepth):
            if (self.DLS(src, target, i)):
                return True
        return False



g = Graph (7);
g.addEdge('A', 'B')
g.addEdge('A', 'C')
g.addEdge('B', 'A')
g.addEdge('B', 'C')
g.addEdge('B', 'D')
g.addEdge('C', 'A')
g.addEdge('C', 'B')
g.addEdge('C', 'E')
g.addEdge('D', 'E')
g.addEdge('D', 'B')
g.addEdge('D', 'F')
g.addEdge('E', 'D')
g.addEdge('E', 'C')
g.addEdge('E', 'F')
g.addEdge('F', 'D')
g.addEdge('F', 'E')


target = 'F'; maxDepth = 3; src = 'A'
```

```
if g.IDDFS(src, target, maxDepth) == True:
    print ("IDFS: Target is reachable from source " +
        "within max depth")
else :
    print ("IDFS: Target is NOT reachable from source " +
        "within max depth")

if g.DLS(src, target, maxDepth)==True:
    print ("DLS: Target is reachable from source " +
        "within max depth")
else :
    print ("DLS: Target is NOT reachable from source " +
        "within max depth")



60.009 seconds


Output:
IDFS: Target is NOT reachable from source within max depth
DLS: Target is reachable from source within max depth
```

# Question 03:

**Commentary on 8 Puzzle Problem:**

- Copy library is imported for deep copy function along with time library to start baseline time.
- The heap functions from python library are imported for Priority Queue. A variable n is declared.
- A class for Priority Queue is initialized with a constructor to initialize a Priority Queue
- Push function is made that inserts a new key 'k'. Pop method is made to remove minimum element from
- Priority Queue. There should also be some check for empty condition so empty method is made to know
- If the Queue is empty or not. Another class node is initialized with self attributes that stores
- The parent node of the current node & helps in tracing path when the answer is found & Copy data
- From parent matrix to current matrix. Another self-attribute is for storing the matrix, another to
- Store the position at which the empty space tile exists in the matrix, next stores the number of

- misplaced tiles & the last self-attribute stores the number of moves made so far.
- A private method is defined so that the priority queue is formed based on the cost variable of the objects.
- calculate Cost() function is made to calculate the number of misplaced tiles ie. number of non-blanks.
- Tiles not in their goal position. Copy library method is utilized to copy data from parent matrix to
- current matrix. Then variables are declared to move tile by 1 position & set number of misplaced tiles
- PrintMatrix() function is defined to print the N x N matrix as well as isSafe() function to check if (x, y)
- Is a valid matrix coordinate. PrintPath() function prints path from root node to destination node.
- Finally, the main solve() function is defined to solve N*N - 1 puzzle algorithm using Branch and
- Bound. empty_tile_pos is the blank tile position in the initial state. The function first has a
- variable pq to create a priority queue to store live nodes of search tree. Another variable root to create
- The root node & finally pushed root to list of live nodes. A loop is initialized that finds a live node
- With least cost, add its children to list of live nodes and finally deletes it from the list.
- If minimum is the answer node, then the path from root to destination is printed
- Another loop generates all possible children, create a child node & add child to list of live nodes.
- Driver Code performs initial configuration where value 0 is used for empty space.