**TASK 1:**

**Implement A\* search and find the optimal path for finding the goal.**

**CODE:**

import queue as Q

def search(graph, start, end):

    whileiterations = 0

    foriteration = 0

    if start not in graph:

        raise TypeError(str(start) + ' not found in graph !')

    if end not in graph:

        raise TypeError(str(end) + ' not found in graph !')

    queue = Q.PriorityQueue()

    queue.put((0, [start]))

    while not queue.empty():

        whileiterations = whileiterations+1

        node = queue.get()

        current = node[1][len(node[1]) - 1]

        if end in node[1]:

            print("Path found: " + str(node[1]) + ", Cost = " + str(node[0]))

            break

        cost = node[0]

        for neighbor in graph[current]:

            foriteration = foriteration+1

            temp = node[1][:]

            temp.append(neighbor)

            queue.put((cost + graph[current][neighbor], temp))

    print("Total while loop executed "+str(whileiterations)+" times")

    print("Total for loop executed "+str(foriteration)+" times")

def main():

    graph = {

    'Karachi': {'Hyderabad': 175, 'Nawabshah': 218, 'Khairpur': 240},

    'Hyderabad': {'Sadiqbad': 271, 'Karachi': 375},

    'Nawabshah': {'Karachi': 318, 'Rahimyarkhan': 411},

    'Khairpur': {'Karachi': 190, 'Sadiqbad': 231, 'Toba Tek Singh': 199, 'Bahawalpur': 60},

    'Sadiqbad': {'Hyderabad': 171, 'Khairpur': 251},

    'Rahimyarkhan': {'Nawabshah': 211, 'Multan': 270},

    'Bahawalpur': {'Khairpur': 180, 'Jhang': 97, 'DG Khan': 246},

    'Multan': {'Rahimyarkhan': 270, 'Faisalabad': 175},

    'DG Khan': {'Faisalabad': 120, 'Bahawalpur': 346, 'Jhang': 138},

    'Jhang': {'Bahawalpur': 97, 'DG Khan': 138, 'Sahiwal': 101},

    'Toba Tek Singh': {'Khairpur': 99, 'Sahiwal': 211},

    'Faisalabad': {'Multan': 75, 'DG Khan': 120},

    'Sahiwal': {'Toba Tek Singh': 211, 'Jhang': 101, 'Lahore': 90},

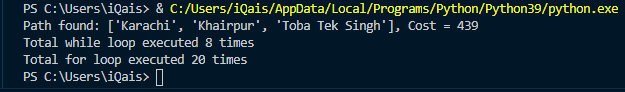
    'Lahore': {'Sahiwal': 90}

    }

    search(graph, 'Karachi', 'Toba Tek Singh')

main()

**OUTPUT:**



**TASK 2:**

**Apply A\* search on pacman game**

**CODE:**

import util

class SearchProblem:

    """

    This class outlines the structure of a search problem, but doesn't implement

    any of the methods (in object-oriented terminology: an abstract class).

    You do not need to change anything in this class, ever.

    """

    def getStartState(self):

        """

        Returns the start state for the search problem.

        """

        util.raiseNotDefined()

    def isGoalState(self, state):

        """

          state: Search state

        Returns True if and only if the state is a valid goal state.

        """

        util.raiseNotDefined()

    def getSuccessors(self, state):

        """

          state: Search state

        For a given state, this should return a list of triples, (successor,

        action, stepCost), where 'successor' is a successor to the current

        state, 'action' is the action required to get there, and 'stepCost' is

        the incremental cost of expanding to that successor.

        """

        util.raiseNotDefined()

    def getCostOfActions(self, actions):

        """

         actions: A list of actions to take

        This method returns the total cost of a particular sequence of actions.

        The sequence must be composed of legal moves.

        """

        util.raiseNotDefined()

def tinyMazeSearch(problem):

    """

    Returns a sequence of moves that solves tinyMaze.  For any other maze, the

    sequence of moves will be incorrect, so only use this for tinyMaze.

    """

    from game import Directions

    s = Directions.SOUTH

    w = Directions.WEST

    return  [s, s, w, s, w, w, s, w]

'''-----------  A\* begins  -----------'''

def aStarSearch(problem, heuristic=nullHeuristic):

    """Search the node that has the lowest combined cost and heuristic first."""

    '''This function pushes non-visited nodes onto the priority queue.

    Nodes are popped one by one, and the following steps are performed:

    1. The node is marked as visited.

    2. If it is a goal node, the loop stops, and the solution is obtained by backtracking using stored parents.

    3. If it is not a goal node, it is expanded.

    4. If the successor node is not visited, its cost is calculated using the heuristic function.

    5. If the cost of the successor node was calculated earlier while expanding a different node,

       and if the new calculated cost is less than old cost, then the cost and parent are updated,

       and it is pushed onto the priority queue with new cost as priority.'''

    # initializations

    # "visited" contains nodes which have been popped from the queue,

    # and the direction from which they were obtained

    visited = {}

    # "solution" contains the sequence of directions for Pacman to get to the goal state

    solution = []

    # "queue" contains triplets of: (node in the fringe list, direction, cost)

    queue = util.PriorityQueue()

    # "parents" contains nodes and their parents

    parents = {}

    # "cost" contains nodes and their corresponding costs

    cost = {}

    # start state is obtained and added to the queue

    start = problem.getStartState()

    queue.push((start, 'Undefined', 0), 0)

    # the direction from which we arrived in the start state is undefined

    visited[start] = 'Undefined'

    # cost of start state is 0

    cost[start] = 0

    # return if start state itself is the goal

    if problem.isGoalState(start):

        return solution

    # loop while queue is not empty and goal is not reached

    goal = False;

    while(queue.isEmpty() != True and goal != True):

        # pop from top of queue

        node = queue.pop()

        # store element and its direction

        visited[node[0]] = node[1]

        # check if element is goal

        if problem.isGoalState(node[0]):

            node\_sol = node[0]

            goal = True

            break

        # expand node

        for elem in problem.getSuccessors(node[0]):

            # if successor is not visited, calculate its new cost

            if elem[0] not in visited.keys():

                priority = node[2] + elem[2] + heuristic(elem[0], problem)

                # if cost of successor was calculated earlier while expanding a different node,

                # if new cost is more than old cost, continue

                if elem[0] in cost.keys():

                    if cost[elem[0]] <= priority:

                        continue

                # if new cost is less than old cost, push to queue and change cost and parent

                queue.push((elem[0], elem[1], node[2] + elem[2]), priority)

                cost[elem[0]] = priority

                # store successor and its parent

                parents[elem[0]] = node[0]

    # finding and storing the path

    while(node\_sol in parents.keys()):

        # find parent

        node\_sol\_prev = parents[node\_sol]

        # prepend direction to solution

        solution.insert(0, visited[node\_sol])

        # go to previous node

        node\_sol = node\_sol\_prev

    return solution

**OUTPUT:**

