

STUDENT: NGO DUC LONG

BJ19BDS002

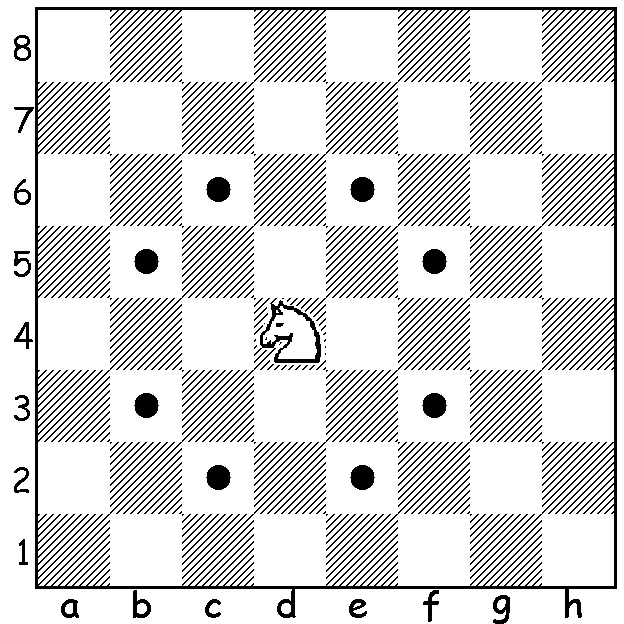
KNIGHT’s TRAVAILS

1. INTRODUCTION:
2. Problem:

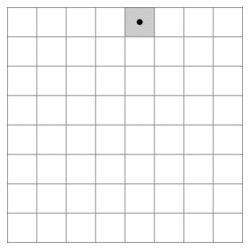
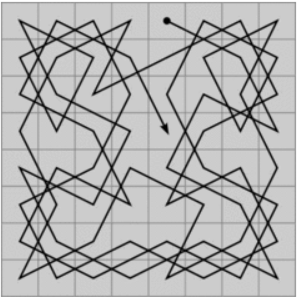
* Given a standard 8x8 chessboard, write Python code that accepts two squares identified by algebraic chess notation. The first square is the starting position, and the second square is the ending position. Find the shortest sequence of valid moves to take a Knight piece from the starting position to the ending position. Each move must be a legal move by a Knight. The output should be the number of steps needed to get a knight from starting point to end point and all the point the knight stop along the way.
* If we expand the problem, we can develop a code where we can set the size of a chess map and set the possible way of moving and still we can find a way of moving to get to the target with the minimum distance traveled.

1. Based knowledge to solve the problem:

* The standard chess size is 8x8
* The **knight move** is unusual among **chess** pieces. It **moves** to a square that is two squares away horizontally and one square vertically, or two squares vertically and one square horizontally.
* At one position in chess map, a knight has at most 8 possible moves



* Given a random starting point, the knight can go to any position in the chess map.

1. SOLUTION FOR THE PROBLEM:
2. Approach:

* This problem can be seen as shortest path in unweighted graph. Hence, we use Breath First Search (BFS) to list out all possible move of the knight.
* With all possible move(reachable, not already visited and is inside the board), create a class called cell to store information about the position, the number of move taken and all previous move(which is stored in a linked list).Then we push object into queue.
* Pop each element out of the queue to check with the target. As soon as, the possible moves match the target, the algorithm stops and return answer. This answer is also the shortest step needed to be taken.

\*Breath First Search(BFS) is a vertex based technique for finding a shortest path in graph. It uses a [queue data structure](https://www.geeksforgeeks.org/queue-data-structure/) which follows first in first out. In BFS, one vertex is selected at a time when it is visited and marked then its adjacent are visited and stored in the queue.

b. Code and Explaination:

* Define class Node:
* Class node will have data and the pointer (nextNode) to point to the next node
* Class node will also have function like getData, getNextNode, setNextNode for the ease of further implementation of a linked list
* Define class LinkedList:
* This is just a singly linked list where there is just one pointer per node pointing toward the next node. The last node will pointing to None.
* The latest node will be append at the end of the list
* The LinkedList class has functions namely copy, addNode, printNode.

class Node:

def \_\_init\_\_(self,data,nextNode=None):

self.data = data

self.nextNode = nextNode

def getData(self):

return self.data

def getNextNode(self):

return self.nextNode

def setNextNode(self,val):

self.nextNode = val

class LinkedList:

def \_\_init\_\_(self,head = None):

self.head = head

def copy(self):

curr = self.head

b=LinkedList()

while curr:

b.addNode(curr.data)

curr = curr.getNextNode()

return b

def addNode(self,data):

newNode = Node(data)

if self.head==None:

self.head = newNode

else:

curr = self.head

while curr.nextNode!=None:

curr=curr.nextNode

curr.nextNode=newNode

return True

def printNode(self):

curr = self.head

while curr:

print(curr.data)

curr = curr.getNextNode()

* Define class cell:
* Get the starting position of the knight as the input and store it in x and y
* Dist is the distance the knight need to get to the target. Every time the knight moves, the dist increase by one to keep track of number of step taken to reach to the target
* Linkedlist is a variable to store the linkedlist. Each cell has its own linkedlist to store the all the previous position

class cell:

def \_\_init\_\_(self, x = 0, y = 0, dist = 0,linkedlist=None):

self.x = x

self.y = y

self.dist = dist

self.linkedlist=linkedlist

* Functions to solve the problem:
* The isInside function is just to check whether the possible move is in the map or not. The function will return True if the possible move is valid and False if the possible move is outside of the map
* In the second function, we store all the possible move in a queue and run a for-loop to access each and every element in a queue. For each element, we check whether it is the target or not. If yes, return the number of step and all previous move. If no, then we find all valid possible move from that position and append it into the queue. While appending, remember to append the object cell(which contain not only the position but also the number of step taken and all previous position). We will stop when we the possible move matches the target.

\*) In the function, I let the chess size as a variable, so we can expand the problem to not only in a 8x8 chess map but in a NxN map.

# checks whether given position is inside the board

def isInside(x, y, N):

if (x >= 1 and x <= N and y >= 1 and y <= N):

return True

return False

# Method returns minimum step to reach the target position

def minStepToReachTarget(knightpos,targetpos, N):

#all possible movments for the knight

dx = [2, 2, -2, -2, 1, 1, -1, -1]

dy = [1, -1, 1, -1, 2, -2, 2, -2]

queue = []

# push starting position of knight with 0 distance

a=LinkedList()

queue.append(cell(knightpos[0], knightpos[1], 0,a))

# make all cell unvisited

visited = [[False for i in range(N + 1)] for j in range(N + 1)]

# visit starting state

visited[knightpos[0]][knightpos[1]] = True

# loop untill we have one element in queue

while(len(queue) > 0):

t = queue[0]

queue.pop(0)

# if current cell is equal to target cell, return its distance and print out the path

if(t.x == targetpos[0] and t.y == targetpos[1]):

t.linkedlist.printNode()

return t.dist

# iterate for all reachable states and check whether the position is in the map or not

for i in range(8):

x = t.x + dx[i]

y = t.y + dy[i]

if (isInside(x, y, N) and not visited[x][y]):

visited[x][y] = True

a=t.linkedlist.copy()

a.addNode((x,y))

queue.append(cell(x, y, t.dist + 1,a))

# Driver Code

if \_\_name\_\_=='\_\_main\_\_':

N = 8

knightpos = [1, 1]

targetpos = [6, 4]

print (minStepToReachTarget(knightpos,targetpos, N))

1. DATA STRUCTURE AND ALGORITHM USED TO SOLVE THE PROBLEM:
2. Breath First Search (BFS):

* In this problem, we can consider all possible moves are vertices of a graph and they are connected to all possible next and previous moves, which can also be considered to be vertices.
* We use Breath First Search instead of Depth First Seach or recursive function because:
* BFS can even give you the shortest distances from one vertex *u* to all other vertices in the graph: for each vertex, just remember the edge that was used to discover it.
* DFS or recursive introduces an overhead, and also might cause your code to hit the stack size limit for large maps(graphs)

1. Linked List:

* Linked list is used in storing the previous position.
* Linked list is used to replace list because:
* Linked List can grow and shrink during run time.
* Insertion and Deletion Operations are Easier
* Efficient Memory Utilization , no need to pre-allocate memory
* Faster Access time can be expanded in constant time without memory overhead