

**COURSE CODE** : CSE2003

DATA STRUCTURES AND ALGORITHMS

**PROJECT: THE KNIGHTS TOUR**

**FACULTY : Dr. M.POONGODI**

**SLOT** : F1

**By**

Robin Thomas J -17BEE1141

Aashiq A-17BEE1185

Ganesh Ram-17BEE1114

***About***

The way the Knight’s tour is solved is using backtracking and recursion, applying the Warnsdorff's rule.

***Input***

The program will ask you for the dimensions of the board and the initial positions in x and y, in order to begin to compute a solution. The positions are integer values in [0,N-1]x[0,N-1].

***Constraints:***

Some constraints apply to have a solution. If you want a solution, the N from the input must be greater than 5. Also, for a graphical tour, N must be less than 32.

***Output***

The program will return a numpy matrix with the tour, and a list with the positions of the knight, which is used to create the graphical tour.

***What is backtracking?***

Backtracking is an algorithmic paradigm that tries different solutions until finds a solution that “works”. Problems which are typically solved using backtracking technique have following property in common. These problems can only be solved by trying every possible configuration and each configuration is tried only once. A Naive solution for these problems is to try all configurations and output a configuration that follows given problem constraints. Backtracking works in incremental way and is an optimization over the Naive solution where all possible configurations are generated and tried.

***Algorithm for backtracking:***

If all squares are visited

print the solution

Else

a) Add one of the next moves to solution vector and recursively

check if this move leads to a solution. (A Knight can make maximum

eight moves. We choose one of the 8 moves in this step).

b) If the move chosen in the above step doesn't lead to a solution

then remove this move from the solution vector and try other

alternative moves.

c) If none of the alternatives work then return false (Returning false

will remove the previously added item in recursion and if false is

returned by the initial call of recursion then "no solution exists" )

***Functions Used:***

1. inRangeAndEmpty : Tells us if the given x and y coordinates are within the n\*n board and hasn’t been visited yet
2. getAccessibility : Tells us if the knight can move to the specified square making use of inRangeAndEmpty.
3. getNextMoves : Gets the next moves of the knight, checking which squares have the least accessibility.
4. graphicTour : This function utilizes the Pygame module to depict a graphic numbered tour of the knight (numpy matrix) if N is less than 32.
5. ifSolution : This function is used to check if there is a possible solution or not for a given value of N.

Method Used:

Warnsdorff’s Algorithm For Knight’s Tours

We investigate Warnsdorff’s simple heuristic for finding knight’s tours on square chess- boards and consider various tiebreaking methods. We then analyse one particular algorithm that is consistent with Warnsdorff’s rule and runs in linear time .This algorithm requires asymptotically less memory than prior known algorithms: *O*(*n*) vs. *O*(*n\*logn*), where *n* = *m*2 is the number of squares. We present a full proof of algorithm’s correctness for the case *m* \_ 7 mod8, and present empirical evidence that suggests its correctness in general.

Warnsdorf's rule is a [heuristic](https://en.wikipedia.org/wiki/Heuristic) for finding a knight's tour. The knight is moved so that it always proceeds to the square from which the knight will have the *fewest* onward moves. When calculating the number of onward moves for each candidate square, we do not count moves that revisit any square already visited. It is, of course, possible to have two or more choices for which the number of onward moves is equal; there are various methods for breaking such ties, including one devised by Pohl and another by Squirrel and Cull.

**Warnsdorff’s Rule:**

* We can start from any initial position of the knight on the board.
* We always move to an adjacent, unvisited square with minimal degree (minimum number of unvisited adjacent).
* A position Q is accessible from a position P if P can move to Q by a single Knight’s move, and Q has not yet been visited.
* The accessibility of a position P is the number of positions accessible from P.

**Algorithm:**

1. Set P to be a random initial position on the board
2. Mark the board at P with the move number “1”
3. Do following for each move number from 2 to the number of squares on the board:
   * let S be the set of positions accessible from P.
   * Set P to be the position in S with minimum accessibility
   * Mark the board at P with the current move number
4. Return the marked board — each square will be marked with the move number on which it is visited

**Various Other Methods For Solving The Knight’s tour**

From Human Solutions to Program Solutions

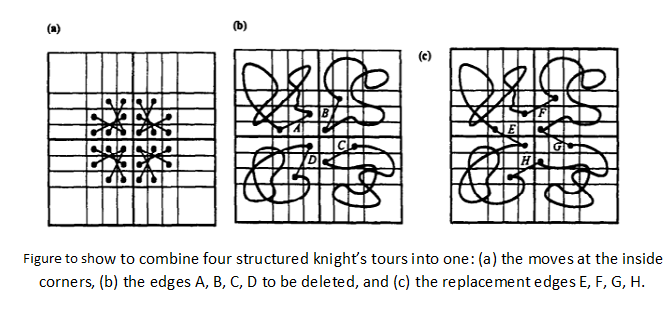
A new approach to knight’s tour is to visualize it as a graph to give a more human-like solution. The major objective is to imagine the tour as a binary tree. This way we can visit one node and remember the other provided that the tour starts from any of the bordering squares. This way the process becomes easier as if the obtained solution is not optimal the other square value from the memory (here stack). Other approach can be to imagine this tour as a planar graph where the neighbors will be the possible moves. Using BFS or DFS will help reduce the complexity faced in brute force and backtracking.

Application of divide and conquer

A knight’s tour is a series of moves made by a knight visiting every square of an n x n chessboard exactly once. The knight’s tour problem is the problem of constructing such a tour, given n. A knight’s tour is called closed if the last square visited is also reachable from the first square by a knight’s move, and open otherwise. Define the knight’s graph for an n x n chessboard to be the graph G = (V,E), where V={(i,j) ( 1 d i,j Gn}, and E={((i,j),(k,Q) ( {Ii-kl,Ij- t[} = {1,2}}. That is, there is a vertex for every square of the board and an edge between two vertices exactly when there is a knight’s move from one to the other. Then, more formally, an open knight’s tour is defined to be a Hamiltonian path, and a closed knight’s tour is defined to be a Hamiltonian cycle on a knight’s graph. A knight’s graph has n2 vertices and 4n2 - 12n + 8 edges.

This section is devoted to describing a new, particularly simple, linear-time divide-and-conquer algorithm for knight’s tours of various types.

For all even n > 6 there exists u structured knight’s tour on an n x n and an n x (n + 2) board. Such a tour can be constructed in time 0(n2).



Graph Theory

The algorithm is not a specific one. Initially the author tried a manual process of solving the 8\*8 chess board, considering different starting points. The author also made use of the Warnsdorff Algorithm and tried solving the problem for other combinations of the board. Warnsdorff’s algorithm involves looking at the valency of each of the next possible vertices (knight’s moves) and choosing that square which has the least degree so that any square likely to be isolated will be used up before isolation occurs. This is all the more logical because no vertex has less than degree 2. The only other stipulation is that if the choices of move all have the same degree then the algorithm will look further down each of the possible avenues of moves applying the least degree rule until a successful successor is found. This algorithm has proved to be very slow for large boards and fails at 76 x 76 squares.

Depth First Search(DFS) Using Stacks

Another search algorithm used to solve the knight’s tour problem is called depth first search (DFS). Whereas the breadth first search algorithm discussed in the previous section builds a search tree one level at a time, a depth first search creates a search tree by exploring one branch of the tree as deeply as possible. In this section we will look at two algorithms that implement a depth first search. The first algorithm we will look at directly solves the knight’s tour problem by explicitly forbidding a node to be visited more than once. The second implementation is more general but allows nodes to be visited more than once as the tree is constructed.

The depth first exploration of the graph is exactly what we need in order to find a path that has exactly 63 edges. We will see that when the depth first search algorithm finds a dead end (a place in the graph where there are no more moves possible) it backs up the tree to the next deepest vertex that allows it to make a legal move.

**Code:**

import pygame, sys

from pygame.locals import \*

import numpy as np

def inRangeAndEmpty(posx,posy,board,N):

return (posx < N and posx >= 0 and posy < N and posy >= 0 and board[posx][posy] == 0)

def getAccessibility(x,y,moves,board,N):

accessibility = 0

for i in range(8):

if inRangeAndEmpty(x+moves[i][0],y+moves[i][1],board,N):

accessibility += 1

return accessibility

def getNextMoves(move,moves,board,N):

positionx = move[0]

positiony = move[1]

accessibility = 8

for i in range(8):

newx = positionx + moves[i][0]

newy = positiony + moves[i][1]

newacc = getAccessibility(newx,newy,moves,board,N)

if inRangeAndEmpty(newx,newy,board,N) and newacc < accessibility:

move[0] = newx

move[1] = newy

accessibility = newacc

return

def graphicTour(N,L\_coor):

horse = pygame.image.load("knight.png")

# Initialize window size and title:

pygame.init()

window = pygame.display.set\_mode((32\*N,32\*N))

pygame.display.set\_caption("Knight's Tour")

background = pygame.image.load("chess.png")

index = 0

# Text:

font = pygame.font.SysFont("Ubuntu",16)

text = []

surface = []

while True:

# Fill background:

window.blit(background,(0,0))

if index < N\*N:

window.blit(horse,(L\_coor[index][0]\*32,L\_coor[index][1]\*32))

text.append(font.render(str(index+1),True,(255,255,255)))

surface.append(text[index].get\_rect())

surface[index].center = (L\_coor[index][0]\*32+16,L\_coor[index][1]\*32+16)

index += 1

else:

window.blit(horse,(L\_coor[index-1][0]\*32,L\_coor[index-1][1]\*32))

for x in range(10000000):

pass

# Check events on window:

for event in pygame.event.get():

if event.type == QUIT:

pygame.quit()

sys.exit()

elif event.type == pygame.KEYDOWN:

if event.key == 27:

pygame.quit()

sys.exit()

for i in range(index):

window.blit(text[i],surface[i])

# Update window:

pygame.display.update()

def ifSolution(Board,N):

for i in range(N):

for j in range(N):

if Board[i][j] == 0:

return False

return True

# Initialize the variables:

N = int(input("Enter N, size of the board (NxN): "))

positionx = int(input("Enter initial x position: "))%N

positiony = int(input("Enter initial y position: "))%N

x = positionx

y = positiony

moveNumber = 2

move = [positionx,positiony]

moves = [[2,1],[2,-1],[1,2],[1,-2],[-1,2],[-1,-2],[-2,1],[-2,-1]]

Board = np.zeros([N,N])

Board[positionx][positiony] = 1

L = []

# We look for the solution and apply Wansdorff:

for i in range(N\*N):

move[0] = positionx

move[1] = positiony

getNextMoves(move,moves,Board,N)

positionx = move[0]

positiony = move[1]

Board[positionx][positiony] = moveNumber

moveNumber += 1

Board[positionx][positiony] -= 1

# We check if we find a solution:

sol = ifSolution(Board,N)

if sol:

# We add the positions to the list of coordinates L:

k = 1

while k <= N\*N:

for i in range(N):

for j in range(N):

if Board[i][j] == k:

L.append([i,j])

k += 1

print(Board)

else:

moves = [[2,1],[-2,1],[2,-1],[-2,-1],[1,2],[-1,2],[1,-2],[-1,-2]]

Board = np.zeros([N,N])

positionx = x

positiony = y

Board[positionx][positiony] = 1

L = []

moveNumber = 2

move = [positionx,positiony]

# We look for the solution and apply Wansdorff Algorithm:

for i in range(N\*N):

move[0] = positionx

move[1] = positiony

getNextMoves(move,moves,Board,N)

positionx = move[0]

positiony = move[1]

Board[positionx][positiony] = moveNumber

moveNumber += 1

Board[positionx][positiony] -= 1

# We check if we find a solution:

sol = ifSolution(Board,N)

if sol:

# We add the positions to the list of coordinates L:

k = 1

while k <= N\*N:

for i in range(N):

for j in range(N):

if Board[i][j] == k:

L.append([i,j])

k += 1

print(Board)

if len(L) == 0:

print("Didn't find a solution.")

print("Knights' positions: ", L)

if N <= 32 and sol:

graphicTour(N,L)