THE KNIGHT'S TOUR

PROJECT REPORT

Submitted for CAL
In
B.Tech Data Structures and Algorithms

(CSE2003)

By

Ishan Bhattacharya	16BEE1164
William C Francis	16BEE1146
Aditya Bhatnagar	16BEC1184
Arun Prakash	16BLC1115

Slot: B1

Name of faculty:**Dr. S V NAGARAJ**

(SCSE)

April, 2017

CERTIFICATE

This is to certify that the Project work entitled "***The Knight's Tour***" that is being submitted by "***Aditya Bhatnagar, Ishan Bhattacharya, William C Francis &Arun Prakash***" for CAL in B.TechData Structures and Algorithms (CSE2003) is a record of bonafide work done under my supervision. The contents of this Project work, in full or in parts, have neither been taken from any other source nor have been submitted for any other CAL course.

***Place*: Chennai**

***Signature of Students*:** **Aditya Bhatnagar**
Ishan Bhattacharya
William C Francis
Arun Prakash

***Signature of Faculty*: Dr. S V Nagaraj**

ACKNOWLEDGEMENTS

We would like to express our special thanks of gratitude to our teacher, Dr. S V Nagaraj, as well as our institution, which gave us the golden opportunity to do this wonderful project on the topic ***The Knight's Tour***, which has helped us learn and understand new things.

We would also like to thank the Dean of the School of Computer Science Engineering for giving us the chance to carry out our vision.

Aditya Bhatnagar

Ishan Bhattacharya

William C Francis

Arun Prakash

CONTENT

ABSTRACT

The aim of this project is to solve the knight's tour problem (both, open and closed loop) using various algorithm techniques, and hence find the most efficient algorithm.

We have understood and implemented 3 basic algorithms to solve this problem. We have worked upon these algorithms and codes using PYTHON 3.4. On implementation, we have analysed the time complexities of each code and compared them to one another.

The 3 algorithms implemented are:

- Brute Force
- Backtracking
- Warnsdorf heuristics

INTRODUCTION

The Knight's Tour is a very renowned problem whose objective is to find the possible legal paths that a single knight can take on a chessboard to visit every square exactly once.

This problem has two variations:

1. One where the knight ends on any position of the chessboard and
2. Where the knight ends one legal knight move away from the starting position.

Our aim is to understand and develop algorithms using different problem solving techniques to solve the Knight's Tour problem and implement it using Python.

ALGORITHMS

- **Warnsdorf Heuristics:**

  Warnsdorf's rule is a 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.

  Time Complexity: O (k^n)

1. Run a menu driven program. Containing options
   - Open loop
   - Closed loop
   - Exit

2. For open loop:
   - define a function initiateboard()
   - set a list with all possible step moves for a knight
   - set global variables board, board_size, knights_moves and set values 0 to all cells of the board.
   - Define a function isAvailable() to check if any slot is free or not. If value of cell is 0, it returns true.
   - Define getPossibleMoves() to check all next possible moves for a particular cell.
   - Define a getNumMoves() to keep count of number of moves.
   - The algorithm will go to that cell which returns the least number of next possible moves.
   - Define a function solve() to recursively solve the knights tour until a suitable solution.
   - Print solution

3. For closed loop:
   - Repeat algorithm for open loop.
   - When solution is obtained, check if initial move is within the moveset of the last move.
   - If the checking returns true, print solution. Else backtrack and repeat.
4. For exit():
   - Run sys.exit()
5. Exit the program.

- **Backtracking:**

  Works in an incremental way to attack problems. When we add an item, we check if adding the current item violates the problem constraint, if it does then we remove the item and try other alternatives. If none of the alternatives work out then we go to the previous stage and remove the item added in the previous stage. If we reach the initial stage back then we say that no solution exists. If adding an item doesn't violate constraints then we recursively add items one by one.

  Time Complexity: O (8^ (n^2-1))

1. Start
2. Read size of board from the user.
3. Initialise all the squares of the board to 0.
4. Set value of initial position to 1 and initialise a counter to 1.Initial position by default will be taken as (0,0).
5. Find the next possible position which satisfies conditions of validity and move to that position.
6. If counter is less than n^2 and all squares are not 1 then return to previous step and choose the next alternative move.
7. For every correct move increment counter by 1 and for every backtracked move decrement it by 1.
8. On reaching a position set value of position as value of counter.
9. When counter reaches n^2 print all the positions in order of values.
10. End

- **Brute Force:**

  Brute force is a reliable but very expensive method of solving programs in terms of time and space. Basically, in this algorithm we will try every possibility without considering optimal solutions. This is not an efficient code but will ultimately get the solution. The algorithm will search for any possibility and if it is found it will go forward to the next position.

  Time Complexity: O (n!)

1. **Open**
   - Create a class 'Knight'sTour' and define the required functions inside it
   - Define size of the chess board
   - Create an NxN chess board with null values
   - Start the knight's tour beginning from the defined position, by default (0,0)
   - All the possible moves from the current position are found out using 'generate_legal_moves'
   - Store the positions in that board that are left to be visited in 'to_visit'
   - If every grid is filled then print the board and path and exits the program
   - or else, it calls the function 'sort_lonely_neighbours' and passes the squares to visit
   - The lonely neighboring squares of the current position are found
   - Sort the lonely neighbors and return back to the function
   - The tour continues by finding the next possible visit from the new position
   - The paths that are visited by the knight are stored in a list called 'path'
   - The program exits when it has filled every square on the board
   - If the program could not fill every square, then it pops the path
   - reset board and try another path
   - If all the paths are tried and still the board is not filled, then it displays "No path found"
   - Exit program

2. **Closed**
   - Create a class 'Knight'sTour' and define the required functions inside it
   - Define size of the chess board (must be >5 and even)
   - Create an NxN chess board with null values
   - Start the knight's tour beginning from the defined position, by default (0,0)
   - All the possible moves from the current position are found out using 'generate_legal_moves'
   - Store the positions in that board that are left to be visited in 'to_visit'
   - If every grid is filled then
   - Check if the last but one move by the knight is made one knight move away from the starting
   - If yes, then print the board and path and exits the program

- or else, it calls the function ‘sort_lonely_neighbours’ and passes the squares to visit
- The lonely neighboring squares of the current position are found
- Sort the lonely neighbors and return back to the function
- The tour continues by finding the next possible visit from the new position
- The paths that are visited by the knight are stored in a list called ‘path’
- The program exits when it has formed a closed knights tour path
- If the solution is not closed tour, then go back to beginning
- reset board and try another path
- If all the paths are tried and still the board is not filled or if a closed tour is not possible then
- Display “No path found”
- Exit program

APPLICATIONS OF ALGORITHMS

- The problem-solving techniques described above has given us a clear insight on their working. These algorithms can be modified accordingly and applied for various problem solving techniques.

- ***Warnsdorf heuristic:***
  The Warnsdorf heuristic provided a revolutionary way to solve the Knight's Tour problem. Such heuristics can make solving complex questions much easier by applying a simple logic.

- ***Brute Force:***
  Though brute-force is simple to implement, and will always find a solution if it exists, its cost is proportional to the number of candidate solutions. Therefore, brute-force search is typically used when the problem size is limited. It is used in cryptography and sorting and searching for small range of data.

- ***Backtracking:***
  Backtracking is a type of self-correcting algorithm and ca be effectively used to solve puzzles such as eight queens problem, sudoku etc.It can also be used for combinatorial optimization problems such as the knapsack problem.

EXPERIMENTAL RESULTS

- **Warnsdorf Heuristics**

(OPEN)

```
=========
>>>
  USING WARNDSDORFF EURISTICS
1. OPEN Tour
2. CLOSED Tour
3. EXIT
enter choice:
 1
enter dimensions: 8
[1, 16, 55, 22, 3, 18, 49, 60]
[40, 23, 2, 17, 56, 61, 4, 19]
[15, 54, 41, 64, 21, 48, 59, 50]
[24, 39, 32, 53, 62, 57, 20, 5]
[33, 14, 63, 42, 47, 30, 51, 58]
[38, 25, 36, 31, 52, 43, 6, 9]
[13, 34, 27, 46, 11, 8, 29, 44]
[26, 37, 12, 35, 28, 45, 10, 7]
>>>
```

(CLOSED)

```
>>>
  USING WARNDSDORFF EURISTICS
1. OPEN Tour
2. CLOSED Tour
3. EXIT
enter choice:
 2
enter dimensions: 8
[36, 1, 18, 21, 54, 61, 16, 13]
[19, 22, 37, 64, 17, 14, 55, 60]
[2, 35, 20, 53, 62, 57, 12, 15]
[23, 38, 63, 40, 47, 52, 59, 56]
[34, 3, 24, 45, 58, 41, 30, 11]
[25, 6, 39, 48, 31, 46, 51, 42]
[4, 33, 8, 27, 44, 49, 10, 29]
[7, 26, 5, 32, 9, 28, 43, 50]
>>>
```

- **Backtracking**

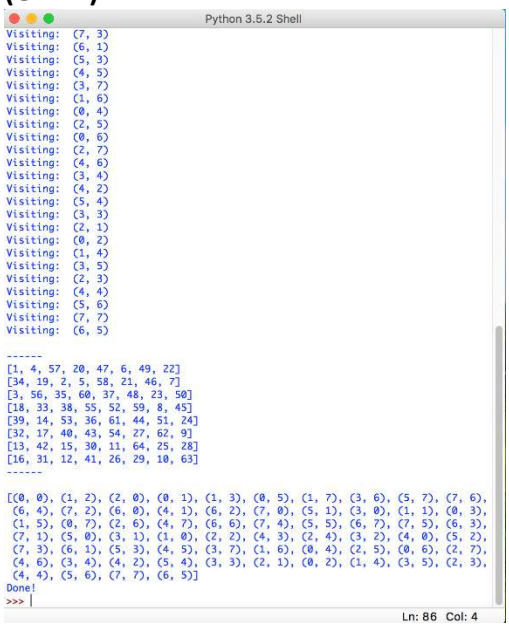
Python 3.5.2 Shell

File Edit Shell Debug Options Window

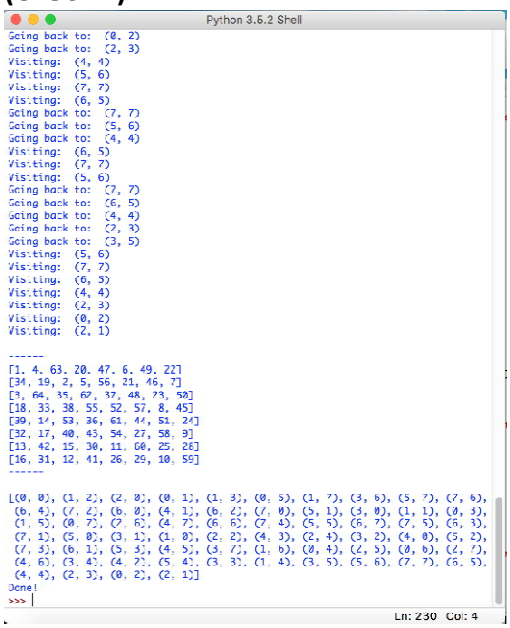
```
Python 3.5.2 (v3.5.2:4def2a2901a
Type "copyright", "credits" or "
>>>
 RESTART: C:\Users\Ishan\Desktop
Enter size of board : 8
+--+--+--+--+--+--+--+--+
| 1|38|55|34| 3|36|19|22|
+--+--+--+--+--+--+--+--+
|54|47| 2|37|20|23| 4|17|
+--+--+--+--+--+--+--+--+
|39|56|33|46|35|18|21|10|
+--+--+--+--+--+--+--+--+
|48|53|40|57|24|11|16| 5|
+--+--+--+--+--+--+--+--+
|59|32|45|52|41|26| 9|12|
+--+--+--+--+--+--+--+--+
|44|49|58|25|62|15| 6|27|
+--+--+--+--+--+--+--+--+
|31|60|51|42|29| 8|13|64|
+--+--+--+--+--+--+--+--+
|50|43|30|61|14|63|28| 7|
+--+--+--+--+--+--+--+--+
>>>
```

- **Brute Force**

(OPEN)



(CLOSED)



LIMITATIONS OF THE ALGORITHMS

- As we can see from the techniques used the Warnsdorf heuristic gives us the fastest solution so far. However extensive research has shown that even this rule fails at very high order of n.

- The brute force algorithm is also effective but only for n<10. Since it does not apply any heuristic and checks every possibility one after the other it takes a very long time to solve the problem and is hence impractical.

- The backtracking algorithm is also effective. It is faster than the brute force method but is still not as good as the Warnsdorf rule.

- For closed knight's tour the order of n must be >5 and even.

CODES

- **Warnsdorf heuristic:**

```
import sys
board = []
board_size = -1
import sys
def initiateBoard (board_dimensions):
    global board
    global board_size
    global knights_moves
board_size = board_dimensions
    for i in range(0, board_size):
board.append(board_size*[0]) #untouched board

knights_moves = ((-2,-1),(1,2),(2,-1),(-2,1),(2,1),(-1,2),(1,-
2),(-1,-2))

def isAvailable(x, y):
    if x <len(board) and y <len(board[0]) and \
       x >= 0 and y >= 0 and board[x][y]==0:
        return True
    else:
        return False

def getPossibleMoves(x, y):
possible_moves = []
     for move in knights_moves:
cx,cy = x+move[0], y+move[1]
        if isAvailable(cx,cy):
possible_moves.append((move[0],move[1]))
     return possible_moves

def getNumMoves(x, y):
     return len(getPossibleMoves(x, y))
def drawBoard():
    for i in range(len(board)):
        print(board[i])
    return


def getNextMove (numMoves):
smallestIndex = 0

    if numMoves==[]:
        for i in range(len(board)):
            for j in range(len(board)):
                if board[i][j]==1:
mmin=(i,j)
                if board[i][j]==xxx*xxx:
mmax=(i,j)
        dif1=mmin[0]-mmax[0]
```

```
        dif2=mmin[1]-mmax[1]
        #print(dif1,dif2)
        if dif1<0:
            dif1*=-1
        if dif2<0:
            dif2*=-1
        #if (dif1==2 and dif2==1) or (dif1==1 and dif2==2):
drawBoard()
sys.exit()
    else:
        smallest = numMoves[0]
        for i in range(len(numMoves)):
            if numMoves[i] < smallest:
                smallest = numMoves[i]
smallestIndex = i
    return smallestIndex

def solve (x,y,num_move):
    assert board[x][y] == 0
    board[x][y] = num_move
possible_moves = getPossibleMoves(x,y)
numOfMoves = []
    for move in possible_moves:
numOfMoves.append(getNumMoves(x+move[0],y+move[1]))
nextMove = possible_moves[getNextMove(numOfMoves)]
    ############################################################
    if len(nextMove)==0:pass
    else:
        solve(x+nextMove[0],y+nextMove[1],num_move+1)
while True:
    print('''   USING WARNDSDORFF EURISTICS
1. OPEN Tour
2. CLOSED Tour
3. EXIT''')
    print('enter choice: ')
ch=int(input(' '))
    if ch==1:
        xxx=int(input('enter dimensions: '))
        if xxx>=5:
initiateBoard (xxx)
            solve(0,0,1)
            continue
        else:
            print('No Solution Exists')
    if ch==2:
        xxx=int(input('enter dimensions: '))
        if xxx>=5 and xxx%2==0:
initiateBoard (xxx)
            solve(0,1,1)
            continue
        else:
            print('No Solution Exists')
elifch==3:
sys.exit()
    else:
        print('invalid input. try again.')
```

- **Backtracking**

```
import sys
Size = 0
# the board size, passed at runtime
board = []
# the board will be constructed as a list of lists

def main():
    global Size
tsize=int(input("Enter size of board : "))
    Size = tsize
# Default: Fill the normal 8x8 chess board
    for i in range(0, Size):
board.append(Size*[0])
# Fill the board with zeroes
    Fill(0,0,1)
# Start the recursion with a 1 in the upper left
    print("No solution found")
# if the recursion returns, it failed

def Validate(ty,tx):
# check if coordinates are within the board
    return ty>=0 and tx>=0 and ty<Size and tx<Size and
board[ty][tx] == 0                                    # and the
square is empty

def Fill(y,x,counter):
# The recursive function that fills the board
    assert board[y][x] == 0
    board[y][x] = counter
# Fill the square
    if counter == Size*Size:
# Was this the last empty square?
PrintBoard()
# Yes, print the board...
sys.exit(1)
# ...and exit
    jumps = ((-2,1),(-1,2),(1,2),(2,1),(2,-1),(1,-2),(-1,-2),(-
2,-1))
    for jump in jumps:
# otherwise, try all the empty neighbours in turn
ty,tx = y+jump[0], x+jump[1]
        if Validate(ty,tx):
            Fill(ty,tx,counter+1)
# *** RECURSION! ***
    board[y][x] = 0
# if we get here, all the neighbours failed,
```

```
# so reset the square and return

def PrintBoard():
# print the board using nice ASCII art ('+' and '-')
    scale = len(str(Size*Size))
    print(Size*("+" + scale*"-") + "+")
    for line in board:
        for elem in line:
sys.stdout.write("|%*d" % (scale,elem))
        print("|\n"+Size*("+" + scale*"-") + "+")

if __name__ == "__main__":
    main()
```

- **Brute Force:**

(open)

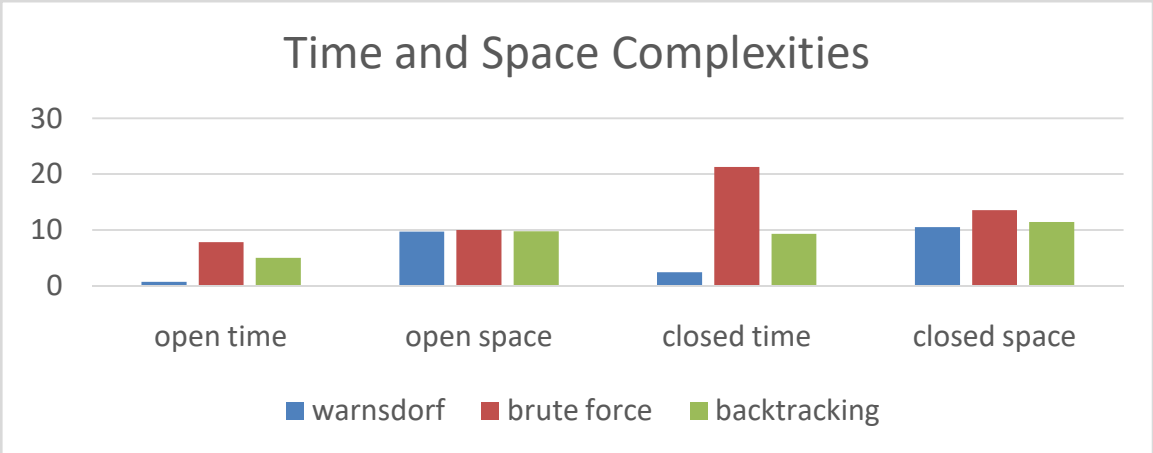
```
import sys
class KnightsTour:
    def __init__(self, width, height):
self.w = width
self.h = height
self.board = []
self.generate_board()
    def generate_board(self):
        """
        Creates a nested list to represent the game board
        """
        for i in range(self.h):
self.board.append([0]*self.w)
    def print_board(self):
        print("  ")
        print("------")
        for elem in self.board:
            print(elem)
        print("------")
        print("  ")

    def generate_legal_moves(self, cur_pos):
        """
        Generates a list of legal moves for the knight to take
next
        """
possible_pos = []
move_offsets = [(1, 2), (1, -2), (-1, 2), (-1, -2),
                        (2, 1), (2, -1), (-2, 1), (-2, -1)]

        for move in move_offsets:
```

```
new_x = cur_pos[0] + move[0]
new_y = cur_pos[1] + move[1]

            if (new_x>= self.h):
                continue
elif (new_x< 0):
                continue
elif (new_y>= self.w):
                continue
elif (new_y< 0):
                continue
            else:
possible_pos.append((new_x, new_y))

        return possible_pos

    def sort_lonely_neighbors(self, to_visit):
        """
        It is more efficient to visit the lonely neighbors first,
        since these are at the edges of the chessboard and cannot
        be reached easily if done later in the traversal
        """
neighbor_list = self.generate_legal_moves(to_visit)
empty_neighbours = []

        for neighbor in neighbor_list:
np_value = self.board[neighbor[0]][neighbor[1]]
            if np_value == 0:
empty_neighbours.append(neighbor)

        scores = []
        for empty in empty_neighbours:
            score = [empty, 0]
            moves = self.generate_legal_moves(empty)
for m in moves:
                if self.board[m[0]][m[1]] == 0:
                    score[1] += 1
scores.append(score)

scores_sort = sorted(scores, key = lambda s: s[1])
sorted_neighbours = [s[0] for s in scores_sort]
        return sorted_neighbours

    def tour(self, n, path, to_visit):
        """
        Recursive definition of knights tour. Inputs are as
follows:
        n = current depth of search tree
        path = current path taken
to_visit = node to visit
        """
```

```
self.board[to_visit[0]][to_visit[1]] = n
path.append(to_visit) #append the newest vertex to the current
point
        print("Visiting: ", to_visit)

        if n == self.w * self.h:#if every grid is filled
self.print_board()
                print(path)
                print("Done!")
sys.exit(1)
        else:
sorted_neighbours = self.sort_lonely_neighbors(to_visit)
            for neighbor in sorted_neighbours:
self.tour(n+1, path, neighbor)

            #If we exit this loop, all neighbours failed so we
reset
self.board[to_visit[0]][to_visit[1]] = 0
            try:
path.pop()
                print("Going back to: ", path[-1])
            except IndexError:
                print("No path found")
sys.exit(1)

if __name__ == '__main__':
    #Define the size of grid. We are currently solving for an 8x8
grid
kt = KnightsTour(8, 8)
kt.tour(1, [], (0,0))
kt.print_board()
```

RESULTS

- We can see from the time complexities that the Warnsdorf heuristic is the most efficient method to solve the problem of the Knight's Tour. However, in recent times a faster algorithm has been presented by Arnd Roth although it doesn't work for higher order n.



- We see that warnsdorf heuristic is the most efficient algorithm in terms of time and space complexity. But even warnsdorf's heuristics has its own limitations.

REFERENCES

- http://www.geeksforgeeks.org/warnsdorffs-algorithm-knights-tour-problem/
- https://www.cs.cmu.edu/~sganzfri/Knights_REU04
- Rob Gaebler, Tsu-wang Yang;1999;Knight's Tour
- Introduction To Algorithm; Charles E. Leiserson, Clifford Stein, Ronald Rivest, and Thomas H. Cormen