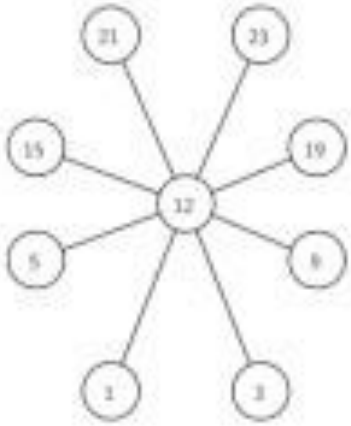
TABLE OF CONTENTS:

THE KNIGHT'S TOUR PROBLEM

Problem Statement:

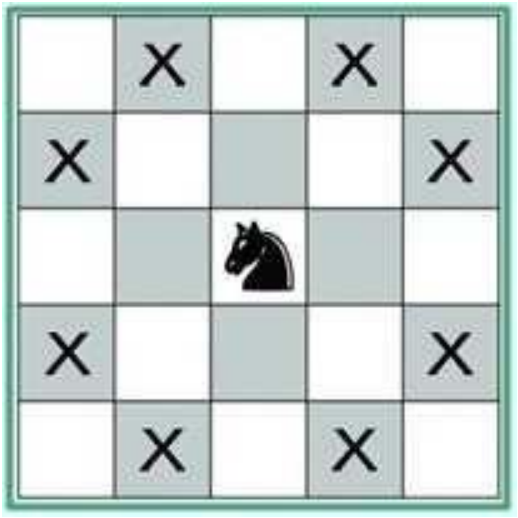
A knight's tour is a sequence of moves of a knight on a N*N chessboard such that the knight visits every square only once. If the knight ends on a square that is one knight's move from the beginning square (so that it could tour the board again immediately, following the same path), the tour is closed, otherwise it is open. Print all sequences of moves of a knight on a chessboard such that the knight visits every square only once.



Description:

The knight's tour problem is an ancient puzzle whose goal is to find out how to construct a series of legal moves made by a knight so that it visits every square of a chessboard exactly once.

According to the problem, we have to make the knight cover all the cells of the board and it can move to a cell only once. There can be two ways of finishing the knight move - the first in which the knight is one knight's move away from the cell from where it began, so it can go to the position from where it started and form a loop, this is called closed tour; the second in which the knight finishes anywhere else, this is called open tour.



Back Tracking Algorithm:

Some problems can be solved by exhaustive search. The exhaustive-search technique suggests generating all candidate solutions and then identifying the one (or the ones) with a desired property. Backtracking is a more intelligent variation of this approach. The principal idea is to construct solutions one component at a time and evaluate such partially constructed candidates as follows. If a partially constructed solution can be developed further without violating the problem's constraints, it is done by taking the first remaining legitimate option for the next component. If there is no legitimate option for the next component, no alternatives for any remaining component need to be considered. In this case, the algorithm backtracks to replace the last component of the partially constructed solution with its next option. It is convenient to implement this kind of processing by constructing a tree of choices being made, called the state-space tree. Its root represents an initial state before the search for a solution begins. The nodes of the first level in the tree represent the choices made for the first component of a solution; the nodes of the second level represent the choices for the second component, and soon. A node in a state-space tree is said to be promising if it corresponds to a partially constructed solution that may still lead to a complete solution; otherwise, it is called non-promising. Leaves represent either non-promising dead ends or complete solutions found by the algorithm. In the majority of cases, a state space tree for a backtracking algorithm is constructed in the manner of depth first search. If the current node is promising, its child is generated by adding the first remaining legitimate option for the next component of a solution, and the processing moves to this child. If the current node turns out to be nonpromising, the algorithm backtracks to the node's parent to consider the next possible option for its last component; if there is no such option, it backtracks one more level up the tree, and so on. Finally, if the algorithm reaches a complete solution to the problem, it either stops (if just one solution is required) or continues searching for other possible solutions.

Algorithm Used:

[Backtracking Algorithms]

isValid(x, y, solution)

Input –Place x and y and the solution matrix.

Output –Check whether the (x,y) is in place and not assigned yet.

Begin

if 0 ≤ x ≤ Board Size and 0 ≤ y ≤ Board Size, and (x, y) is empty, then

return true

End

knightTour(x, y, move, sol, xMove, yMove)

Input –(x, y) place, number of moves, solution matrix, and possible x and y movement lists.

Output –The updated solution matrix if it exists.

Begin

if move = Board Size * Board Size, then //when all squares are visited

return true

for k := 0 to number of possible xMovement or yMovement, do

xNext := x + xMove[k]

yNext := y + yMove[k]

if isValid(xNext, yNext, sol) is true, then

sol[xNext, yMext] := move

if knightTour(xNext, yNext, move+1, sol, xMove, yMove), then

return true

else

remove move from the sol[xNext, yNext] to backtrack

done

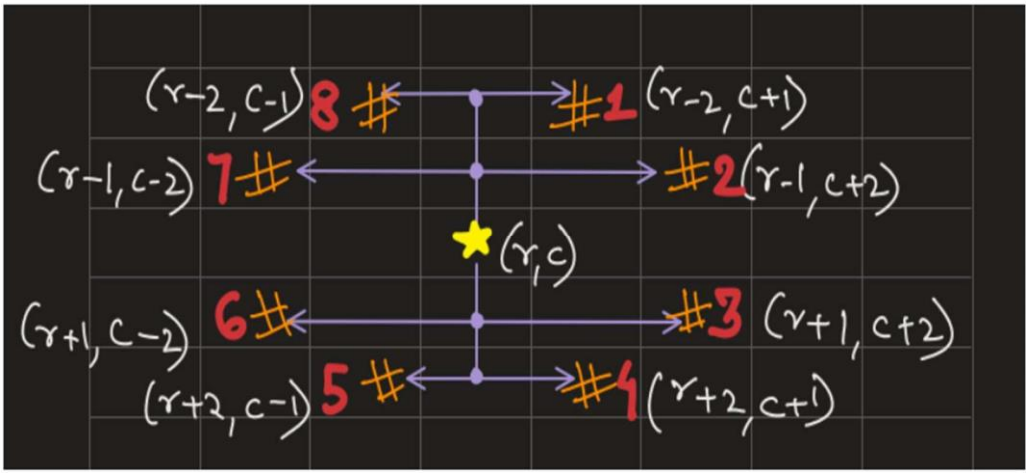
return false

End

Methodology:

We are starting with the chess grid with all cells filled with 0 (empty cells). Now, we need to fill these cells so that the configuration becomes valid. For every cell, we will try to place the knight (give the cell number with value = current move), and then call the recursive function for all the possible 8 moves as the next cell and move number greater by 1.

We should try to visualize how a knight moves on a chessboard.

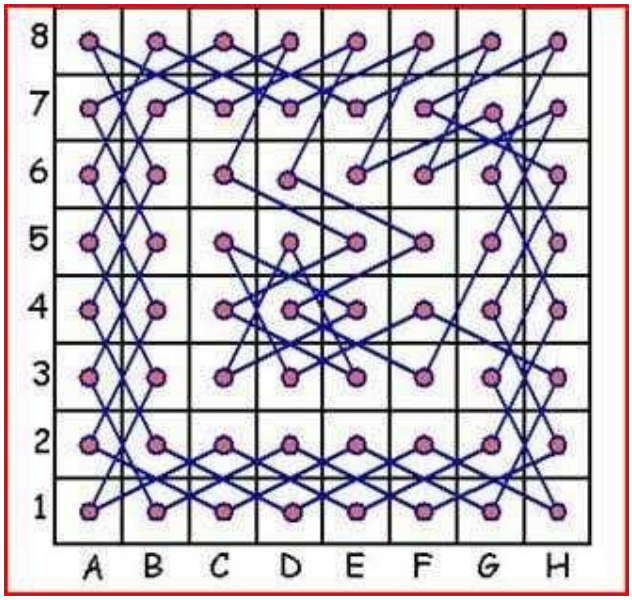


If the current cell is (r, c), then we will try for all possible 8 moves (in clockwise direction), i.e. in the order(r-2, c+1), (r-1, c+2), (r+1, c+2), (r+2, c+1), (r+1, c-1), **(r+1, c-2), (r-1, c-2) and (r-2, c-1). After making a move to one of the 8 possible,** we will check if the current cell is valid or not, i.e. whether the value of the row or column has gone out of bound or not (< 0 or >= size of grid).

Also, the current move can be invalid if it had been already visited in any of the previous moves. This is because we are not allowed to visit any cell twice, i.e., we must visit every cell exactly once. Hence, we will also return (invalid configuration), if the current cell is already filled with value > 0 (a move number).

Thus, these are the checks to find whether the current cell is an invalid move or not:

1. row < 0 or row >= n (invalid row)



2. col < 0 or col >= n (invalid column)

3. chess[row][col]> 0 0
(already filled)

Now the only condition left is the base case?
We will print the current configuration when we have already visited all the cells exactly once, i.e. the current move is equal to ^2 where n =n size of grid.

But before printing, we should –

- fill the current cell with the last move (= n^2).
- And after printing, we will make the current cell back to 0 (empty), as we are backtracking (to print all possible configurations),
- and then return.

Code:

```
#include <stdio.h>
#define N 8

int is_valid(int i, int j, int sol[N+1][N+1]) {  if (i>=1 && i<=N &&
j>=1 && j<=N && sol[i][j] ==-1)    return 1;
 return 0; }

int knight_tour(int sol[N+1][N+1], int i, int j, int step_count, int x_move[], int y_move[]) {  if
(step_count == N*N)    return 1;

 int k;  for (k=0; k<8; k++) {    int
next_i = i+x_move[k];    int
next_j = j+y_move[k];

   if(is_valid(i+x_move[k], j+y_move[k], sol)) {      sol[next_i] [next_j] = step_count;
if (knight_tour(sol, next_i, next_j, step_count+1, x_move, y_move))       return 1;
    sol[i+x_move[k]] [j+y_move[k]] = -1; // backtracking
  }
}

 return 0; }

int start_knight_tour() {
 int sol[N+1] [N+1];

 int i, j;  for (i=1; i<=N;
i++) {    for (j=1; j<=N;
j++) {      sol[i][j] = -1;
```

```
  }
}

 int x_move[] = {2, 1, -1, -2, -2, -1, 1, 2};  int y_move[] =
{1, 2, 2, 1, -1, -2, -2, -1};  sol [1][1] = 0; // placing knight
at cell (1, 1)  if (knight_tour(sol, 1, 1, 1, x_move,
y_move)) {
  for (i=1; i<=N; i++) {      for
(j=1; j<=N; j++) {
printf("%d\t",sol[i][j]);
    }
    printf("\n");
  }
  return 1;
 }  return 0;
}

int main () {
 printf ("%d\n", start_knight_tour());
return 0; }
```

Output:

Output

```
/tmp/Nwe9YuovHy.o
0   59  38  33  30  17  8   63
37  34  31  60  9   62  29  16
58  1   36  39  32  27  18  7
35  48  41  26  61  10  15  28
42  57  2   49  40  23  6   19
47  50  45  54  25  20  11  14
56  43  52  3   22  13  24  5
51  46  55  44  53  4   21  12
1
```

Analysis:

Time Complexity:

There are N*N i.e., N ^2 cells in the board and we have a maximum of 8 choices to make from a cell, so we can write the worst-case running time as O (8^N2) But we don't have 8 choices for each cell. For example, from the first cell, we only have two choices.

Even considering this, our running time will be reduced by a factor and will become O(kN ^2) instead of O (8 N ^2).This is also indeed an extremely bad running time.

Space Complexity:

The maximum number of recursive calls/depth of the recursion tree can be equal to the number of cells. Hence the recursion stack call will take (n ^2) space complexity and at max

O(n2) However, we are not using any extra data structure, hence the solution is said to have O(1)auxiliary space.

Conclusion:

Example of the problems that can be attacked by backtracking algorithm does not mean that backtracking solution is the best approach for those problems. There may exist other algorithmic approach that gives optimal solution for those problems. Ant based algorithm and genetic algorithm are impractical as they execute for long time to produce solution of knight's tour. For standard 8×8 chessboard, our algorithm gives moderate time solution compared to other efficient solutions.

Bibliography:

http://www.cis.upenn.edu/~matuszek/cit594-2009/Lectures/35-backtracking.ppt
http://mathworld.wolfram.com/KnightsTour.html http://en.wikipedia.org/wiki/Knight%27s_tour
https://www.geeksforgeeks.org/the-knights-tour-problem-backtracking-1/