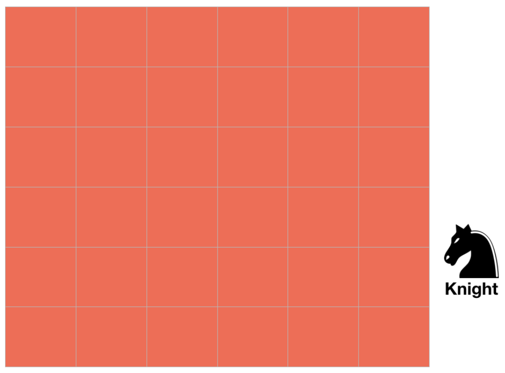
Knight's Tour Problem | Backtracking

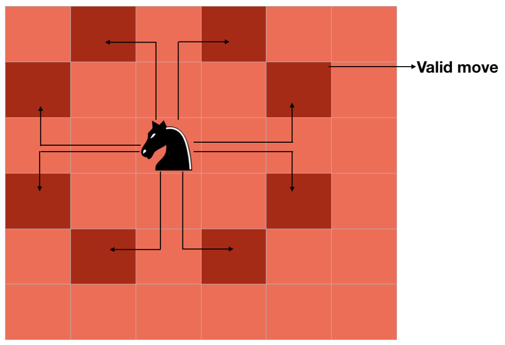
In the previous example of backtracking (/course/algorithms-backtracking/), you have seen that backtracking gave us a $O(n!)$ running time. Generally, backtracking is used when we need to check all the possibilities to find a solution and hence it is expensive. For the problems like N-Queen and Knight's tour, there are approaches which take lesser time than backtracking, but for a small size input like 4x4 chessboard, we can ignore the running time and the backtracking leads us to the solution.

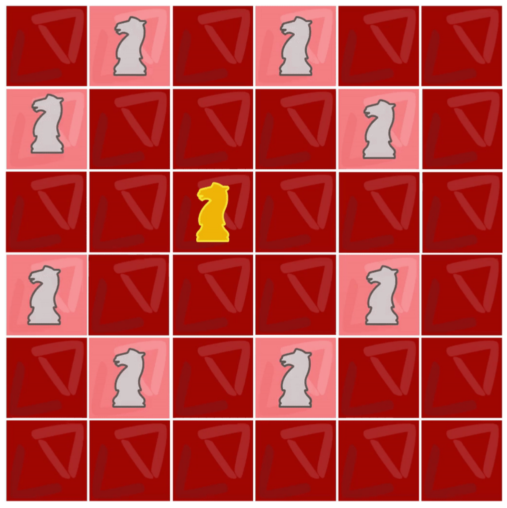
Knight's tour is a problem in which we are provided with a NxN chessboard and a knight.



6x6 Chessboard

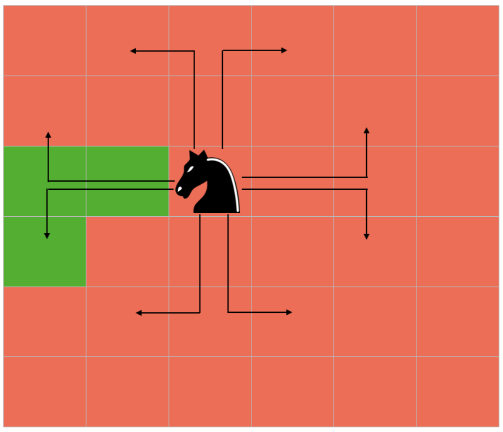
For a person who is not familiar with chess, the knight moves two squares horizontally and one square vertically, or two squares vertically and one square horizontally as shown in the picture given below.





Thus if a knight is at (3, 3), it can move to the (1, 2) ,(1, 4), (2, 1), (4, 1), (5, 2), (5, 4), (2, 5) and (4, 5) cells.

Thus, one complete movement of a knight looks like the letter "L", which is 2 cells long.



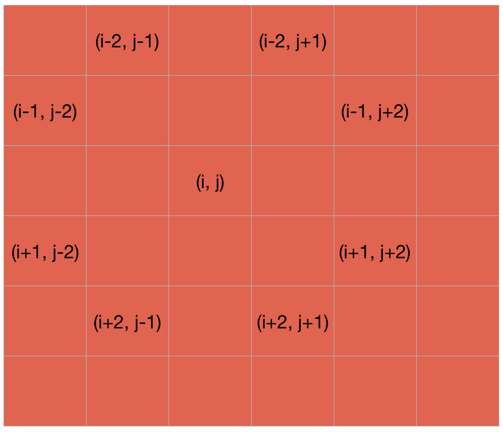
According to the problem, we have to make the knight cover all the cells of the board and it can move to a cell only once.

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**.

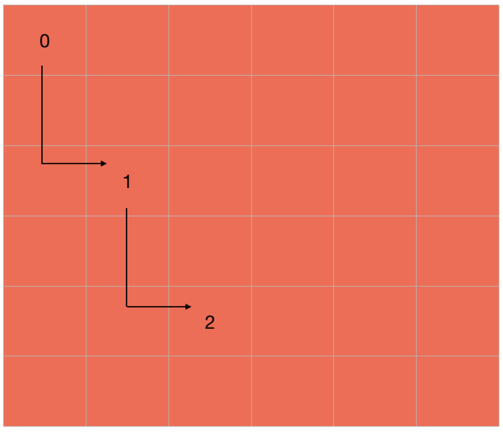
Approach to Knight's Tour Problem

Similar to the N-Queens problem, we start by moving the knight and if the knight reaches to a cell from where there is no further cell available to move and we have not reached to the solution yet (not all cells are covered), then we backtrack and change our decision and choose a different path.

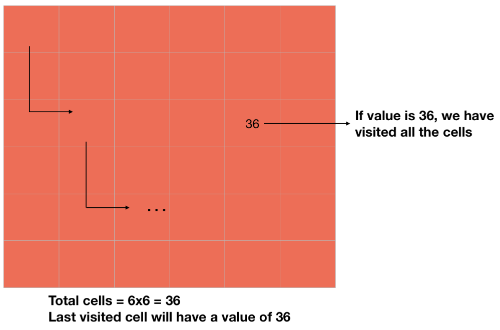
So from a cell, we choose a move of the knight from all the moves available, and then recursively check if this will lead us to the solution or not. If not, then we choose a different path.



As you can see from the picture above, there is a maximum of 8 different moves which a knight can take from a cell. So if a knight is at the cell (i, j), then the moves it can take are - (i+2, j+1), (i+1, j+2), (i-2,j+1), (i-1, j+2), (i-1, j-2), (i-2, j-1), (i+1, j-2) and (i+2, j-1).



We keep the track of the moves in a matrix. This matrix stores the step number in which we visited a cell. For example, if we visit a cell in the second step, it will have a value of 2.


36
...
If value is 36, we have visited all the cells
Total cells = 6x6 = 36
Last visited cell will have a value of 36

This matrix also helps us to know whether we have covered all the cells or not. If the last visited cell has a value of N*N, it means we have covered all the cells.

Thus, our approach includes starting from a cell and then choosing a move from all the available moves. Then we check if this move will lead us to the solution or not. If not, we choose a different move. Also, we store all the steps in which we are moving in a matrix.

Now, we know the basic approach we are going to follow. So, let's develop the code for the same.

Code for Knight's Tour

Let's start by making a function to check if a move to the cell (i, j) is valid or not - `IS-VALID(i, j, sol)`. As mentioned above, 'sol' is the matrix in which we are storing the steps we have taken.

A move is valid if it is inside the chessboard (i.e., i and j are between 1 to N) and if the cell is not already occupied (i.e., `sol[i][j] == -1`). We will make the value of all the unoccupied cells equal to -1.

```
IS-VALID(i, j, sol)
  if (i>=1 and i<=N and j>=1 and j<=N and sol[i][j]==-1)
    return TRUE
  return FALSE
```

As stated above, there is a maximum of 8 possible moves from a cell (i, j). Thus, we will make 2 arrays so that we can use them to check for the possible moves.

```
x_move = [2, 1, -1, -2, -2, -1, 1, 2]
y_move = [1, 2, 2, 1, -1, -2, -2, -1]
```

Thus if we are on a cell (i, j), we can iterate over these arrays to find the possible move i.e., (i+2, j+1), (i+1, j+2), etc.

Now, we will make a function to find the solution. This function will take the solution matrix, the cell where currently the knight is (initially, it will be at (1, 1)), the step count of that cell and the two arrays for the move as mentioned above - `KNIGHT-TOUR(sol, i, j, step_count, x_move, y_move)`.

We will start by checking if the solution is found. If the solution is found (`step_count == N*N`), then we will just return true.

```
KNIGHT-TOUR(sol, i, j, step_count, x_move, y_move)
  if step_count == N*N
    return TRUE
```

Our next task is to move to the next possible knight's move and check if this will lead us to the solution. If not, then we will select the different move and if none of the moves are leading us to the solution, then we will return false.

As mentioned above, to find the possible moves, we will iterate over the x_move and the y_move arrays.

```
KNIGHT-TOUR(sol, i, j, step_count, x_move, y_move)
  ...
  for k in 1 to 8
    next_i = i+x_move[k]
    next_j = j+y_move[k]
```

Now, we have to check if the cell (i+x_move[k], j+y_move[k]) is valid or not. If it is valid then we will move to that cell - `sol[i+x_move[k]][j+y_move[k]] = step_count` and check if this path is leading us to the solution ot not - `if KNIGHT-TOUR(sol, i+x_move[k], j+y_move[k], step_count+1, x_move, y_move)`.

```
KNIGHT-TOUR(sol, i, j, step_count, x_move, y_move)
  ...
  for k in 1 to 8
    ...
    if IS-VALID(i+x_move[k], j+y_move[k])
      sol[i+x_move[k]][j+y_move[k]] = step_count
      if KNIGHT-TOUR(sol, i+x_move[k], j+y_move[k], step_count+1, x_move, y_move)
        return TRUE
```

If the move (i+x_move[k], j+y_move[k]) is leading us to the solution - `if KNIGHT-TOUR(sol, i+x_move[k], j+y_move[k], step_count+1, x_move, y_move)`, then we are returning true.

If this move is not leading us to the solution, then we will choose a different move (loop will iterate to a different move). Also, we will again make the cell (i+x_move[k], j+y_move[k]) of solution matrix -1 as we have checked this path and it is not leading us to the solution, so leave it and thus it is backtracking.

```
KNIGHT-TOUR(sol, i, j, step_count, x_move, y_move)
  ...
  for k in 1 to 8
    ...
      sol[i+x_move[k], j+y_move[k]] = -1
```

At last, if none the possible move returns us false, then we will just return false.

```
KNIGHT-TOUR(sol, i, j, step_count, x_move, y_move)
  if step_count == N*N
    return TRUE

  for k in 1 to 8
    next_i = i+x_move[k]
    next_j = j+y_move[k]
    if IS-VALID(next_i, next_j, sol)
      sol[next_i][next_j] = step_count
      if KNIGHT-TOUR(sol, next_i, next_j, step_count+1, x_move, y_move)
        return TRUE
      sol[next_i, next_j] = -1

  return FALSE
```

We have to start the KNIGHT-TOUR function by passing the solution, x_move and y_move matrices. So, let's do this.

As stated earlier, we will initialize the solution matrix by making all its element -1.

```
for i in 1 to N
  for j in 1 to N
    sol[i][j] = -1
```

The next task is to make x_move and y_move arrays.

```
x_move = [2, 1, -1, -2, -2, -1, 1, 2]
y_move = [1, 2, 2, 1, -1, -2, -2, -1]
```

We will start the tour of the knight from the cell (1, 1) as its first step. So, we will make the value of the cell(1, 1) 0 and then call `KNIGHT-TOUR(sol, 1, 1, 1, x_move, y_move)`.

```
START-KNIGHT-TOUR()
  sol =[][]
  for i in 1 to N
    for j in 1 to N
      sol[i][j] = -1

  x_move = [2, 1, -1, -2, -2, -1, 1, 2]
  y_move = [1, 2, 2, 1, -1, -2, -2, -1]

  sol[1, 1] = 0

  if KNIGHT-TOUR(sol, 1, 1, 1, x_move, y_move)
    return TRUE
  return FALSE
```

C **Python** **Java**

```
#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;
}
```

</>

Take a note that the order of the x_move and y_move arrays are going to affect the running time of the algorithm drastically. Think of a case, when a person chooses 6 wrong long paths and finally reaching the goal in the 7th path and another case when the person took the correct path in the first turn. This is also similar. The order given here is a tested one.

</>

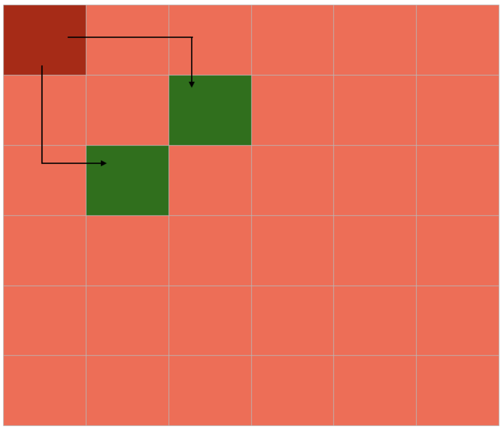
As discussed in the first chapter, we should not learn anything as a rule. It's always good to have a heuristic than to apply the backtracking algorithm blindly. For example, if we know that our destination is in the east direction, then checking the paths leading in the east first is a wise technique apart from checking all the paths.

Analysis of Code

We are not going into the full time complexity of the algorithm. Instead, we are going to see how bad the algorithm is.

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 $O(8^{N^2})$.

But we don't have 8 choices for each cell. For example, from the first cell, we only have two choices.



Two choices for the first cell

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

So, this chapter was to make you familiar with the backtracking algorithm and not about the optimization. You can look for more examples on the backtracking algorithm with the backtracking tag of the BlogsDope (https://www.codesdope.com/blog/tag/backtracking/?tag=backtracking).

“ The computer was born to solve problems that did not exist before. ”

- Bill Gates

PREV (/course/algorithms-backtracking/) (/course/algorithms-greedy-algorithm/) NEXT