

The Knight's tour problem

Backtracking is an algorithmic paradigm that tries different solutions until finds a solution that “works”. Problems that are typically solved using the backtracking technique have the 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 incrementally and is an optimization over the Naive solution where all possible configurations are generated and tried. For example, consider the following Knight's Tour problem.

Problem Statement:

Given a $N \times N$ board with the Knight placed on the first block of an empty board. Moving according to the rules of chess knight must visit each square exactly once. Print the order of each cell in which they are visited.

Example:

Input :

$N = 8$

Output:

```
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
```

The path followed by Knight to cover all the cells

Following is a chessboard with 8×8 cells. Numbers in cells indicate the move number of Knight.

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

Let us first discuss the Naive algorithm for this problem and then the Backtracking algorithm.

Naive Algorithm for Knight's tour

The Naive Algorithm is to generate all tours one by one and check if the generated tour satisfies the constraints.

```
while there are untried tours
{
    generate the next tour
    if this tour covers all squares
    {
        print this path;
    }
}
```

Backtracking works in an incremental way to attack problems. Typically, we start from an empty solution vector and one by one add items (Meaning of item varies from problem to problem. In the context of Knight's tour problem, an item is a Knight's move). 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 works 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. If the solution vector becomes complete then we print the solution.

Backtracking Algorithm for Knight's tour

Following is the Backtracking algorithm for Knight's tour problem.

```
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" )
```

Following are implementations for Knight's tour problem. It prints one of the possible solutions in 2D matrix form. Basically, the output is a 2D 8*8 matrix with numbers from 0 to 63 and these numbers show steps made by Knight.

```

def solveKnightTour(n):

    chess = [[0]*n for i in range(n)]
    const = 1
    solveKnightTourHelper(chess,const,0,0)
    # return chess

def solveKnightTourHelper(chess,const,x,y):
    if x<0 or y<0 or x>=len(chess) or y>=len(chess) or chess[x][y]>0:
        return
    if const == len(chess)**2:
        chess[x][y] = const
        print(chess)
        chess[x][y]=0
        return
    chess[x][y] = const
    solveKnightTourHelper(chess, const+1, x+2, y+1)
    solveKnightTourHelper(chess, const+1, x+1, y+2)
    solveKnightTourHelper(chess, const+1, x-1, y+2)
    solveKnightTourHelper(chess, const+1, x-2, y+1)
    solveKnightTourHelper(chess, const+1, x-2, y-1)
    solveKnightTourHelper(chess, const+1, x-1, y-2)
    solveKnightTourHelper(chess, const+1, x+1, y-2)
    solveKnightTourHelper(chess, const+1, x+2, y-1)
    chess[x][y]=0

print(solveKnightTour(5))

```