Lesson 12c: Backtracking

Introduction

Backtracking

Backtracking is an algorithmic technique for solving problems recursively by trying to build a solution incrementally. It incrementally builds candidates to the solutions and abandons a candidate ("backtracks") as soon as it determines that the candidate cannot lead to a valid solution (the candidate does not satisfy the constraints of the problem). Each candidate solution is only evaluated once.

Ex. A Sudoku being solved by backtracking. Each cell is tested for a valid number, moving "back" when there is a violation, and moving forward again until the puzzle is solved.

Difference between Recursion and Backtracking:

- In recursion, the function calls itself until it reaches a base case.
- In backtracking, we use recursion to explore all the possibilities until we get the best result for the problem.

Backtracking

Backtracking works incrementally, and it is an improvement to the naïve or brute force approach (where all configurations are generated and tried).

The backtracking technique searches for a solution to a problem among all the available options.

PseudoCode for **Backtracking** :

```
boolean findSolutions(n, other params) :
    if (found a solution):
        displaySolution()
        return True

    for (val = first to last) :
        if (isValid(val, n)):
            applyValue(val, n)
            if (findSolutions(n+1, other params))
                return True
            removeValue(val, n)
        return False
```

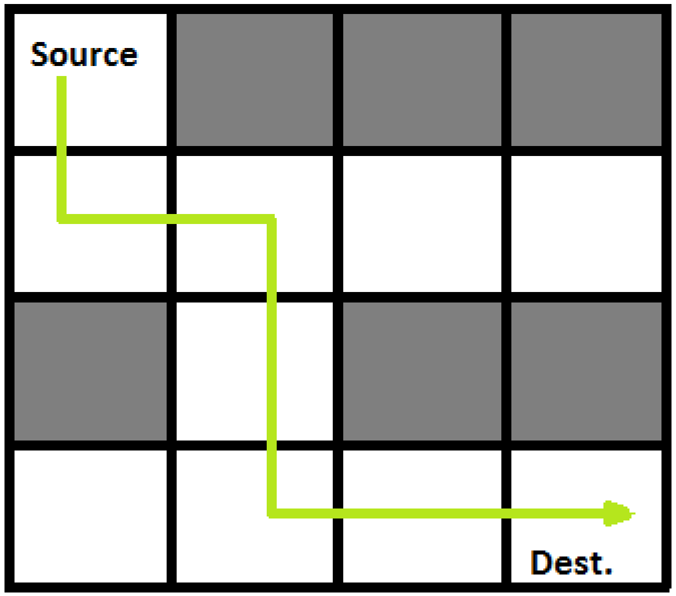
Example – Rat in a Maze (labyrinth)

A Maze is given as N*N binary matrix of blocks where source block is maze[0][0] and destination block is maze[N-1][N-1]. A rat starts from source and has to reach the destination. The rat can move only in vertical and horizontal directions

In the maze matrix, 0 means the block is a dead end and 1 means the block can be used in the path from source to destination.

```
maze =  [[1, 0, 0, 0],
         [1, 1, 1, 1],
         [0, 1, 0, 0],
         [1, 1, 1, 1]]
```

```
result =
1 0 0 0
1 1 0 0
0 1 0 0
0 1 1 1
```



Example – Rat in a Maze (labyrinth)

Approach: Implement a recursive function, which will follow a path and check if the path reaches the destination or not. If the path cannot reach the destination then **backtrack** and try other paths.

Steps:

1. Create a solution matrix, initially filled with 0’s.
2. Create a recursive function, which takes initial matrix, output matrix and position of rat (i, j).
3. If the position is out of the matrix or position not valid then discard it and return.
4. Mark the position output[i][j] as 1 and check if the current position is the destination or not. If destination is reached print the output matrix and return.
5. Recursively search for next position (movement)
6. Unmark position (i, j), i.e output[i][j] = 0 (backtracking)

Example – Rat in a Maze (labyrinth)

```
def solveMaze(maze):
    # Creating a n x n matrix
    res = [[0 for i in range(n)] for i in range(n)]
    res[0][0] = 1

    move_x = [-1, 1, 0, 0]   # x matrix for each direction
    move_y = [0, 0, -1, 1]   # y matrix for each direction

    if RatMaze(n, maze, move_x, move_y, 0, 0, res):
        # print solution
        for i in range(n):
            for j in range(n):
                print(res[i][j], end=' ')
            print()
    else:
        print('Solution does not exist')

# MAIN
n = 4  # Maze size
maze = [[1, 0, 0, 0],
        [1, 1, 0, 1],
        [0, 1, 0, 0],
        [1, 1, 1, 1]]

solveMaze(maze)
```

```
res = []
for i in range(n):
   res.append([0]*n)
```

Example – Rat in a Maze (labyrinth)

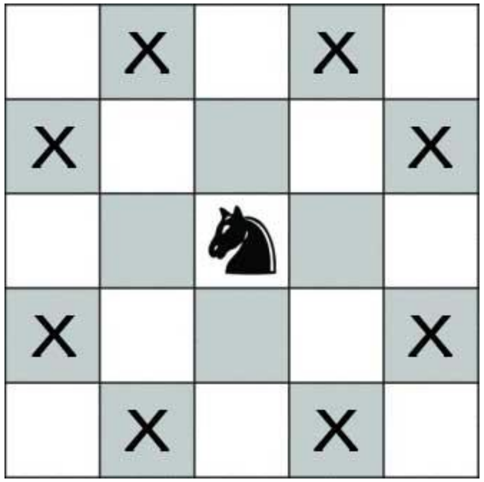
```python
# To check if x, y is a valid index for N * N Maze
def isValid(n, maze, x, y, res):
    if 0<=x<n and 0<=y<n and maze[x][y] == 1 and res[x][y] == 0:
        return True
    return False


# A recursive function to solve Maze problem
def RatMaze(n, maze, move_x, move_y, x, y, res):

    if x == n-1 and y == n-1: # if (position (x,y) is goal/destiny)
        return True

    for i in range(4):
        x_new = x + move_x[i] # Generate new value of x
        y_new = y + move_y[i] # Generate new value of y

        # Check if maze[x][y] is valid
        if isValid(n, maze, x_new, y_new, res):
            res[x_new][y_new] = 1   # mark x, y as part of solution path
            if RatMaze(n, maze, move_x, move_y, x_new, y_new, res):
                return True
            res[x_new][y_new] = 0   # backtracking
    return False
```

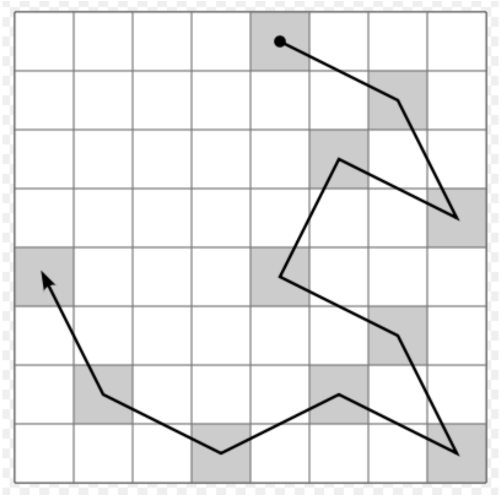
Exercise - Knight's Tour

The "Knight's Tour" is an ancient puzzle in which the objective is to move a knight, starting from one square on a chessboard, to every other position, landing on each square only once.

Problem:
N*N board and the Knight placed on the first position of an empty board. Moving according to the rules of chess, the knight must visit each square exactly once. Print the order of each cell in which they are visited.

```
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
```





Exercise - Knight's Tour

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;
   }
}
```

Exercise - Knight's Tour

We start from an empty solution vector and add items one by one (an item is a Knight’s move). When we add an item, we check if this item violates the problem constraint. If it does then we remove it and try other alternatives.

If none of the alternatives works out then we go back (backtracking) and remove the item added in the previous stage. If we reach the initial stage, then 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.

```
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 8 moves. Choose one of them).
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. If false is returned by the initial call of
    recursion then "no solution exists" )
```

Exercise - Knight's Tour

```python
def isSafe(x, y, board):
    # To check if i,j are valid indexes for N*N chessboard
    if(x >= 0 and y >= 0 and x < n and y < n and board[x][y] == -1):
        return True
    return False

def solveKTUtil(n, board, curr_x, curr_y, move_x, move_y, pos):
    #  A recursive function to solve Knight Tour problema

    if(pos == n**2):
        return True

    # Try all next moves from the current coordinate x, y
    for i in range(8):
        new_x = curr_x + move_x[i]
        new_y = curr_y + move_y[i]
        if(isSafe(new_x, new_y, board)):
            board[new_x][new_y] = pos
            if(solveKTUtil(n, board, new_x, new_y, move_x, move_y, pos+1)):
                return True
             # Backtracking
            board[new_x][new_y] = -1
    return False

# MAIN
n = 8 # Chessboard Size
solveKT(n)
```

Exercise - Knight's Tour

```python
def printSolution(n, board):
    for i in range(n):
        for j in range(n):
            print(board[i][j], end=' ')
        print()

def solveKT(n):
    # This function solves the Knight Tour problem using Backtracking
    # It returns false if no complete tour is possible, otherwise returns true

    board = [[-1 for i in range(n)]for i in range(n)] # Initialization of Board matrix

    # move_x and move_y define next move of Knight.
    move_x = [2, 1, -1, -2, -2, -1, 1, 2] # move_x is for next value of x coordinate
    move_y = [1, 2, 2, 1, -1, -2, -2, -1] # move_y is for next value of y coordinate

    board[0][0] = 0    # Since the Knight is initially at the first block
    pos = 1            # Step counter for knight's position

    # Checking if solution exists or not
    if(not solveKTUtil(n, board, 0, 0, move_x, move_y, pos)):
        print("Solution does not exist")
    else:
        printSolution(n, board)
```

https://www.geeksforgeeks.org/the-knights-tour-problem-backtracking-1/

Lesson 12d: Backtracking Exercise

Sudoku

Sudoku is a puzzle consisting of 81 cells distributed in a 9x9 grid that is divided into 3x3 subgrids. The rules for filling cells are summarized with:

- Each box must have a number between 1 and 9.
- At each row, column and subgrid, there must be the 9 numbers, without repeating any of them.

In all Sudokus, there are some input numbers, called clues, which serve as a starting point to solve the puzzle, as well as providing a unique solution.

				6		3	5	
6	4						2	
1		9			8	7		
		2	9		7			
9								5
			1		2	8		
		6	5			1		9
	2						7	4
	9	8		7				

To represent a Sudoku we can use a 2D array in which each element of the array corresponds to a cell and contains one number. If the cell is empty we will save a 0 in that position of the array.

Implement the `is_valid` function that receives as parameters a 2D array representing a Sudoku, a number (between 1 and 9) and a row and column value (between 0 and 8) and check if it is valid to put a number in that position of the Sudoku. Remember that putting a number in a cell is valid if this value is not repeated within the same row, the same column or in the corresponding 3x3 subgrid, and in addition, the cell is not occupied by another number.

Make a main program that reads a Sudoku from the file "sudoku1.txt", creates the matrix *sudoku*, prints it, and asks the user to enter the *num*, *row* and *column* to fill the Sudoku. If the position is valid, update the Sudoku. The process continues until the user enters num= -1

Sudoku – user input

```python
import numpy as np

def is_valid(sudoku, n, r, c):
    r_ini_subgrid = (r//3) *3
    c_ini_subgrid = (c//3) *3
    valid = (sudoku[r,c] == 0
       and n not in sudoku[r,:] and n not in sudoku[:,c]
       and n not in sudoku[r_ini_subgrid:r_ini_subgrid+3, c_ini_subgrid:c_ini_subgrid+3])
    return valid


#main
sudoku = np.loadtxt("sudoku1.txt", dtype='int', delimiter=" ", skiprows=0)
print(sudoku)
num = int(input("Num: "))
while(num!=-1):
    row = int(input("Row: "))
    col = int(input("Column: "))
    valid = is_valid(sudoku, num, row, col)
    if (valid):
        print("The number is VALID at this position")
        sudoku[row, col] = num
    else:
        print("The number is NOT VALID at this position")
    print(sudoku)
    num = int(input("Num: "))
```

Sudoku - Backtracking

Naive Approach:

Generate all possible configurations of numbers from 1 to 9 to fill the empty cells.

- For every unassigned position fill the position with a number from 1 to 9.
- After filling ALL the unassigned positions check if the matrix is safe or not.
- If safe print, else try other cases.

Sudoku using Backtracking:

- Assign numbers one by one to empty cells, BUT before assigning a number, check whether it is valid to assign this number (the same number is not present in the current row, current column and current 3X3 subgrid).
- If it is valid, assign the number, and recursively check whether this assignment leads to a solution or not.
- If the assignment doesn’t lead to a solution, then discard it and try the next number for the current empty cell.
- If none of the number (1 to 9) leads to a solution, return false and print no solution exists.

Sudoku – Backtracking (solved)

```
import numpy as np
def is_valid(sudoku, n, r, c):
    r_ini_subgrid = (r//3) *3
    c_ini_subgrid = (c//3) *3
    valid = (sudoku[r,c] == 0
       and n not in sudoku[r,:] and n not in sudoku[:,c]
       and n not in sudoku[r_ini_subgrid:r_ini_subgrid+3, c_ini_subgrid:c_ini_subgrid+3])
    return valid

def solve_sudoku(sudoku):
    for rov in range(0, 9):
        for col in range(0, 9):
            if sudoku[rov][col] == 0:
                for val in range(1, 10):
                    if is_valid(sudoku, val, rov, col):
                        sudoku[rov][col] = val
                        if solve_sudoku(sudoku):
                            return True
                        sudoku[rov][col] = 0   #Backtracking
                return False   #Solution failed, backtrack for next value
    return True

### MAIN
sudoku = np.loadtxt("sudoku1.txt", dtype='int', delimiter=" ", skiprows=0)
if(solve_sudoku(sudoku)):
    print(sudoku)
else:
    print ("No solution exists")
```