INDEX

Abstract

INTRODUCTION

NAÏVE ALGORITHM

BACKTRACKING

TIME COMPLEXITY

SPACE COMPLEXITY

WARNDORFF’S ALGORITHM

DIVIDE AND CONQUER APPRAOCH

IMPLEMENTATION

OUTPUT

CONCLUSION

REFERENCES

ABSTRACT

A knight must traverse all of the squares on an m × n chessboard but visit every square once and only once and return to the originated square where the knight moves in an L-shape route. Many great mathematicians attempted to solve the general problem. Their methods are based on either divide-and-conquer or backtracking dedicated for solving the 8 × 8 chessboard knight‘s tour problem. In our project we have implemented the knight’s tour using Warndorff’s rule and divide and conquer approach and analysed their running tie complexities.

INTRODUCTION

A knight’s tour is a series of moves made by a knight visiting every square of an n x n chessboard exactly once. The knight’s tour problem is the problem of constructing such a tour, given n. A knight’s tour is called closed if the last square visited is also reachable from the first square by a knight’s move, and open otherwise. Variations of the knight's tour problem involve chessboards of different sizes than the usual 8 × 8, as well as irregular (non-rectangular) boards. The knight's tour problem is an instance of the more general Hamiltonian path problem in graph theory. The problem of finding a closed knight's tour is similarly an instance of the Hamiltonian cycle problem. Unlike the general Hamiltonian path problem, the knight's tour problem can be solved in linear time.

Various studies were carried out on a smaller chessboard so that many skills frequently used by the artificial intelligence (AI) research community, like depth-first search, breadth-first search and heuristic methods, were adopted. Unfortunately, with the use of larger chessboards, the computation time rises unacceptably rapidly. From various studies it could be deduced what kinds of rectangular chessboards have knight's tours; in particular, an n×n chessboard has a closed knight's tour if and only if n is even and greater than 5, and an open knight's tour if and only if n is greater than 4. One approach was proposed, which seeked to solve the problem by dividing the board horizontally into two rectangular compartments. The tour has to visit all the squares in one compartment before proceeding to the second; this is called the bisected knight's tour problem. This approach was further refined by dividing the board into four rectangular compartments. This revision is called the quadrisected knight's tour problem. Domorya described a quadrisected open knight's tour on an 8×8 board and a closed knight's tour on a 7×7 board with its centre square missed (all squares are visited except the centre one). A linear time sequential algorithm was proposed to construct open knight's tours between any pair of squares on n×n boards for n⩾5.

A divide-and-conquer algorithm was proposed that can generate closed knight's tours on n×n or n×(n+2) boards in linear time (i.e., O(n2)) for all even n and n⩾10, and closed knight's tours missing one corner in linear time if n is odd and greater than 4. This algorithm can also construct closed knight's tours on n×(n+1) boards when n⩾6. This algorithm works by finding closed knight’s tour on smaller boards as bases. When facing a larger board, it is divided into four smaller pieces, generating a closed knight's tour for each compartment, and finally removing 4 edges and adding 4 edges at the inside corners to combine these four smaller closed knight's tours into a complete closed knight's tour for the larger board.

However, this algorithm is only able to find closed knight's tours on n×n, n×(n+1) or n×(n+2) boards when n⩾10. His method fails to deal with boards of any other arbitrary size. In this paper, we will solve the knight's tour problem completely.

POSSIBLE SOLUTION APPROACHES:

NAÏVE 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:

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.

ALGORITHM:

- 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")

TIME COMPLEXITY:
There are N^2 Cells and for each, we have a maximum of 8 possible moves to choose from, so the worst running time is O (8^{N^2}).

SPACE COMPLEXITY: O(N*N), size of the board and general recursive calls are not considered in this calculation.

No order of the xMove, yMove is wrong, but they will affect the running time of the algorithm drastically. For example, in the case if 8th choice of the move is the correct one, and before that the code ran 7 different wrong paths. It’s always a good idea a have a heuristic than to try backtracking randomly.

Thus, backtracking approach could only be used for boards which are smaller in size. The algorithm performs very poor with increase in size of the chess board that is for large values since the running time of this algorithm is of exponential order.

WARNDORFF’S ALGORITHM:

Wendorff’s rule is a heuristic for finding a single 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 (each move is made to the adjacent vertex with the least degree). When calculating the number of onward moves for each candidate square, we do not count moves that revisit any square already visited.

This rule may also more generally be applied to any graph. Although the Hamiltonian path problem is NP-hard in general, on many graphs that occur in practice this heuristic is able to successfully locate a solution in linear time. The knight's tour is such a special case.

Warnsdorff’s Rule:

- We can start from any initial position of the knight on the board.
- We always move to an adjacent, unvisited square with minimal degree (minimum number of unvisited adjacent).
- This algorithm may also more generally be applied to any graph.

A position Q is accessible from a position P if P can move to Q by a single Knight’s move, and Q has not yet been visited. The accessibility of a position P is the number of positions accessible from P.

ALGORITHM:

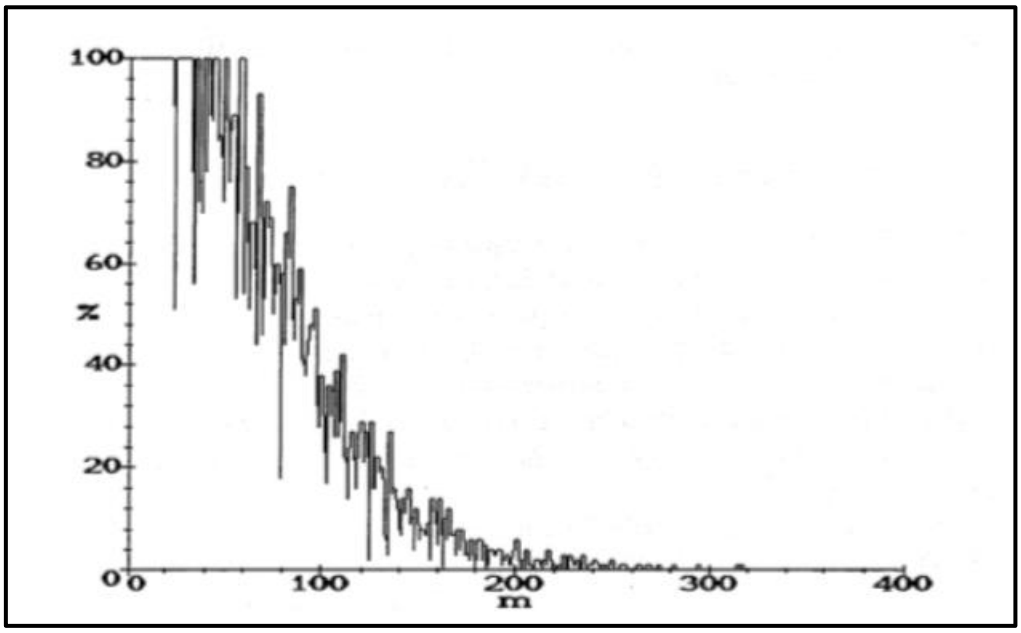
- Set P to be a random initial position on the board
- Mark the board at P with the move number “1”
- Do following for each move number from 2 to the number of squares on the board:
- let S be the set of positions accessible from P.
- Set P to be the position in S with minimum accessibility
- Mark the board at P with the current move number
- Return the marked board — each square will be marked with the move number on which it is visited.

Time complexity:

Warnsdorff's rule finds a tour in linear time. The time complexity of this heuristic is O (N × N) where N is the dimension of the chessboard

We saw that the Warnsdorff's algorithm is fast also for M > 6 (where m is the number of rows/columns), but it has a problem: it does not always succeed

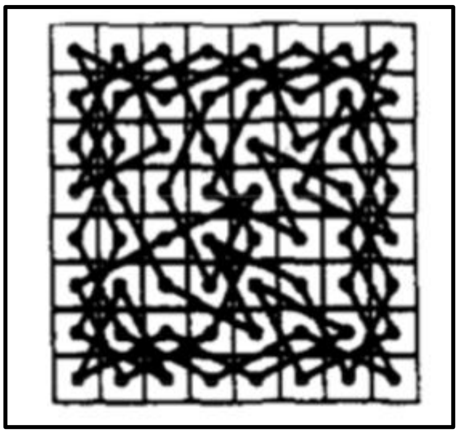
The algorithm produces a successful tour over 85% of the time on most boards with m less than 50, and it succeeds over 50% of the time on most boards with m less than 100. However, for m > 200 the success rate is less than 5%, and for m > 325 there were no successes at all. These observations suggest that the success rate of Warnsdorff's random algorithm rapidly goes to 0 as m increases. This can also be seen in the below graph



DIVIDE AND CONQUER APPRAOCH:

Divide and conquer approach could be used for larger values of m which was proposed by Ian Parberry. He proposed various theorems and corollaries, describe the various cases for which there exists a structured knight's tour.

There exists a knight's tour for all m>6. To construct such tours, large boards need to be split into smaller boards until the base case is reached: a board with a given closed solution. Here, we consider only large boards that are multiples of our base case (board 8×8):



- For all n x n boards where $n \geq 12$ is divisible by 4, there exists a quadrisected knight's tour that is symmetric under a $180°$ rotation. Such a tour can be constructed in time $O(n^2)$.
- For all $n \geq 6$ there exists a structured knight's tour on an $n * (n+1)$ board. Such a tour can be constructed in time $O(n^2)$.
- For all odd $n \geq 5$ there exists a structured knight's tour on an n x n board that is missing one of its corner squares. Such a tour can be constructed in time $O(n^2)$.
- For all n x n boards where $n \geq 10$ is divisible by 2 but not by 4, there exists a knight's tour that is symmetric under a $90°$ rotation. Such a tour can be constructed in time $O(n^2)$.
- For all n x n boards where $n \geq 8$ is even, there exists a knight's tour that is symmetric under a $180°$ rotation. Such a tour can be constructed in time $O(n^2)$.

Once we know the tour for our base case, we can solve boards with m=k8, where k∈ℕ. When k≥2, we split the board into 44 boards. Intuitively, if the new boards have k≥2 they are split recursively until we reach boards with m=8.

For convenience, to solve the tour we will start with the board in the bottom left corner. Once we reach the upper corner of this board, instead to make move D (figure b) and complete the tour in the bottom left board, we will make move E (figure c) and we enter in the upper left board. Once we explore all the upper left board, instead of closing the tour with move A we go to the upper right board with move F. With the same reasoning we complete the upper right board and then the bottom right board. We then re-enter in the bottom left board and we complete the tour.

The image below is a graphic representation of the algorithm:

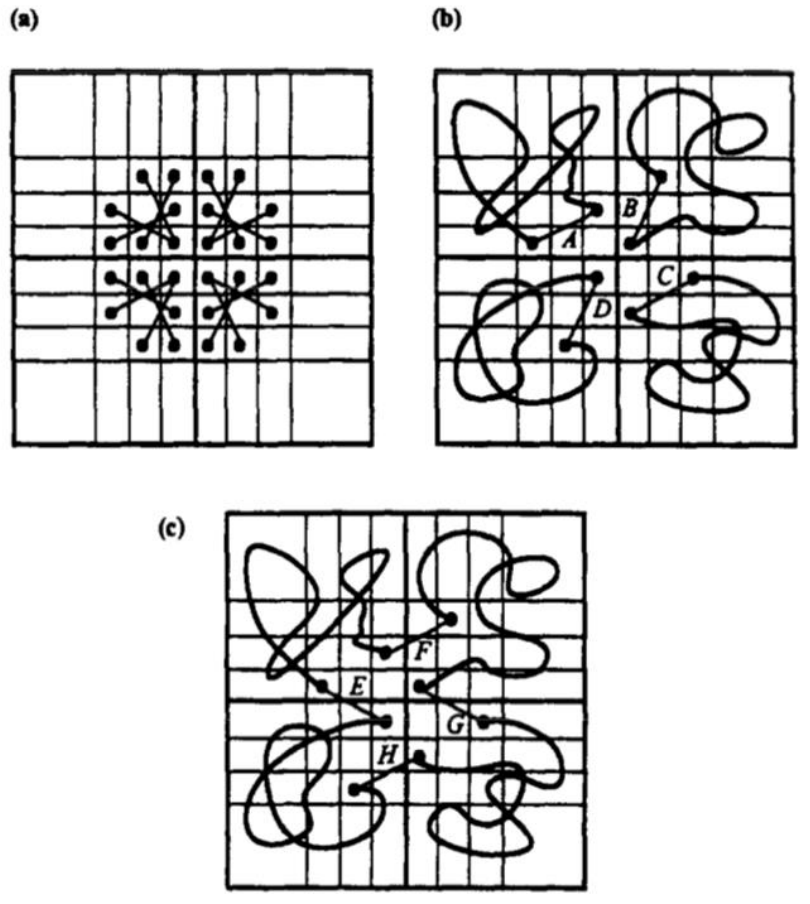
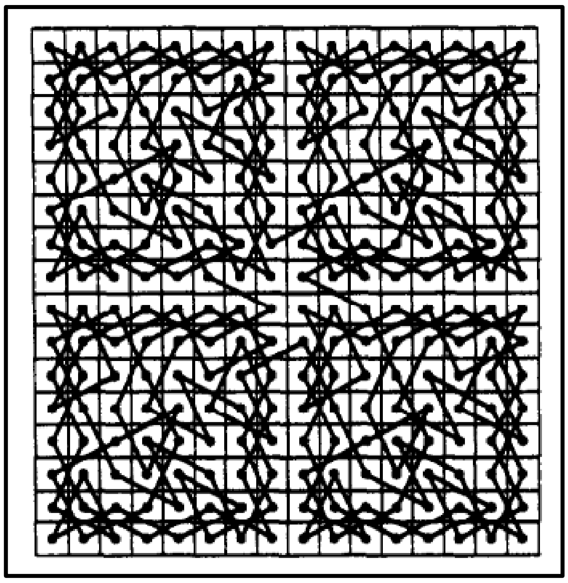


Fig. 3. How to combine four structured knight's tours into one: (a) the moves at the inside corners, (b) the edges A, B, C, D to be deleted, and (c) the replacement edges E, F, G, H.

Every time the tour is closed such as the base case. This *closeness* allows us to break the tour in the corners, having a one-step connection between the first and last visited cell in the sub-board.

TIME COMPLEXITY:

The running time T(n) required for the construction of a knight's tour on an n x n board is given by the following recurrence:$T(8) = O(1)$, and for $n \geq 16$ a power of 2, $T(n) = 4\,T\left(\frac{n}{2}\right) + O(1)$. This recurrence has solution$T(n) = O(n^2)$. Therefore (using the standard argument), the running time for all even $n \geq 6\ is\ O(n^2)$.

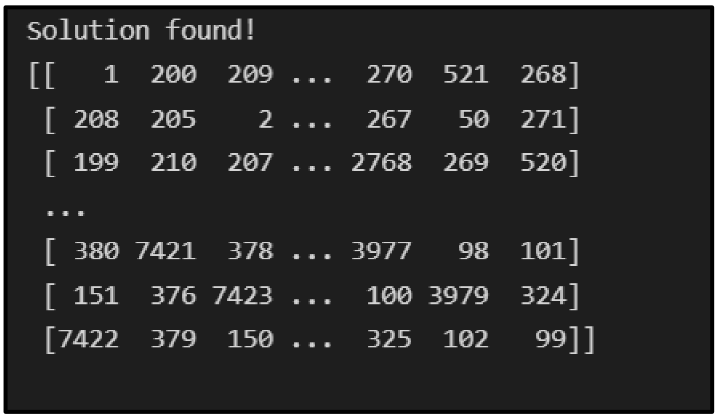


A 16 x 16 knight's tour constructed from the 8 x 8 knight's tour

IMPLEMENTATION:

```
def GetMoves(x,y,n):
    pos_x = (2, 1, 2, 1, -2, -1, -2, -1)
    pos_y = (1, 2, -1, -2, 1, 2, -1, -2)
    possible_moves = []
    # check if the move in a certain direction is legal
    for i in range(8):
        if x+pos_x[i] >= 0 and x+pos_x[i] < n and y+pos_y[i] >= 0 and
y+pos_y[i] < n and chess_board[x+pos_x[i]][y+pos_y[i]] == 0:
            possible_moves.append([x+pos_x[i], y+pos_y[i]])
    return possible_moves
# n: number of rows/columns, x & y are the coordinates
def Solve(x,y,n,chess_board):
    # check if the initial points are correct
    if x<n and y<n:
        x = x
        y = y
        chess_board[x][y] = 1
        counter = 2
        for i in range((n*n)-1):
            possible_moves = GetMoves(x,y,n)
            if len(possible_moves)==0:
                print("No solution found from ",x,",",y)
                break
            minimum = possible_moves[0]
            # comparing the degree of every cell in the list of possible moves
and getting the one with minimal degree
            for p in possible_moves:
                if len(GetMoves(p[0],p[1],n)) <=
len(GetMoves(minimum[0],minimum[1],n)):
                    minimum = p
            x = minimum[0]
            y = minimum[1]
            #print(x,y)
            chess_board[x][y] = counter
            counter += 1
        if counter==n*n+1:
            print("Solution found!")
    else:
        print("insert correct initial points")
    print(np.array(chess_board))
# set the number of rows/columns (nxn) in the chess board
n=100
chess_board = [[0 for i in range(n)] for j in range(n)]
# initial position
x=0
y=0
Solve(x,y,n,chess_board)
# resetting the chessboard to recompile this cell
chess_board = [[0 for i in range(n)] for j in range(n)]
```

OUTPUT:



```
Solution found!
[[   1  200  209 ...  270  521  268]
 [ 208  205    2 ...  267   50  271]
 [ 199  210  207 ... 2768  269  520]
 ...
 [ 380 7421  378 ... 3977   98  101]
 [ 151  376 7423 ...  100 3979  324]
 [7422  379  150 ...  325  102   99]]
```

DIVIDE AND CONQUER IMPLEMENTATION:

```
### KNIGHT MOVES
KnightMoves = {(-2,-1),(-1,-2),(+1,-2),(+2,-1),(+2,+1),(+1,+2),(-1,+2),(-
2,+1)}
### Already given closed knight's tour for 8x8 chess board (base case).
tour8x8 = {
    (1,1): [(2,3),(3,2)], (2,3): [(3,1),(1,1)],
    (3,1): [(1,2),(2,3)], (1,2): [(2,4),(3,1)],
    (2,4): [(1,6),(1,2)], (1,6): [(2,8),(2,4)],
    (2,8): [(3,6),(1,6)], (3,6): [(1,5),(2,8)],
    (1,5): [(2,7),(3,6)], (2,7): [(4,8),(1,5)],
    (4,8): [(6,7),(2,7)], (6,7): [(8,8),(4,8)],
    (8,8): [(7,6),(6,7)], (7,6): [(8,4),(8,8)],
    (8,4): [(6,5),(7,6)], (6,5): [(7,7),(8,4)],
    (7,7): [(8,5),(6,5)], (8,5): [(7,3),(7,7)],
    (7,3): [(8,1),(8,5)], (8,1): [(6,2),(7,3)],
    (6,2): [(4,1),(8,1)], (4,1): [(2,2),(6,2)],
    (2,2): [(1,4),(4,1)], (1,4): [(2,6),(2,2)],
    (2,6): [(1,8),(1,4)], (1,8): [(3,7),(2,6)],
    (3,7): [(5,8),(1,8)], (5,8): [(6,6),(3,7)],
    (6,6): [(8,7),(5,8)], (8,7): [(6,8),(6,6)],
    (6,8): [(4,7),(8,7)], (4,7): [(3,5),(6,8)],
    (3,5): [(4,3),(4,7)], (4,3): [(5,5),(3,5)],
    (5,5): [(3,4),(4,3)], (3,4): [(1,3),(5,5)],
    (1,3): [(2,1),(3,4)], (2,1): [(4,2),(1,3)],
    (4,2): [(5,4),(2,1)], (5,4): [(4,6),(4,2)],
    (4,6): [(3,8),(5,4)], (3,8): [(1,7),(4,6)],
    (1,7): [(2,5),(3,8)], (2,5): [(3,3),(1,7)],
    (3,3): [(5,2),(2,5)], (5,2): [(4,4),(3,3)],
    (4,4): [(6,3),(5,2)], (6,3): [(7,1),(4,4)],
    (7,1): [(8,3),(6,3)], (8,3): [(7,5),(7,1)],
    (7,5): [(5,6),(8,3)], (5,6): [(6,4),(7,5)],
    (6,4): [(4,5),(5,6)], (4,5): [(5,7),(6,4)],
    (5,7): [(7,8),(4,5)], (7,8): [(8,6),(5,7)],
    (8,6): [(7,4),(7,8)], (7,4): [(8,2),(8,6)],
    (8,2): [(6,1),(7,4)], (6,1): [(5,3),(8,2)],
    (5,3): [(7,2),(6,1)], (7,2): [(5,1),(5,3)],
    (5,1): [(3,2),(7,2)], (3,2): [(1,1),(5,1)]
}
### n is the number of rows/columns of the board.
### It is split succesfully only if n is a multiple of 8.
def Split(n):
    if n/8 == floor(n/8):
        new_lenght = n/2
        return int(new_lenght)
    else:
        print("Failed to split", n, "in two parts")
        sys.exit('ERROR: Failed to split into 8x8 sub-boards')
class Chessboard:
    def __init__(self, rows, columns):
        self.rows = rows
        self.columns = columns
        self.KnightPathList = {}
        self.knightTour = {}
    def SetPathList(self, KnightPathList):
        self.KnightPathList = KnightPathList.copy()
    def GetPathList(self):
        return self.KnightPathList.copy()
    def SetTour(self, tour):
        self.knightTour = tour.copy()
    def GetTour(self):
        return self.knightTour.copy()
    def GetRows(self):
        return self.rows
    def GetColumns(self):
        return self.columns
    def FindPathList(self):
        nRows = self.GetRows()
        nColumns = self.GetColumns()
        ## Base case: we set the tour for the above given chess boards
        if (nRows == 8) and (nColumns == 8):
            self.SetPathList(tour8x8)
            return
        ## Recursive case: we split the boards (into 4 equal subBoards) until
we find a base case
        newRows = Split(nRows)
        newColumns = Split(nColumns)
        topLeftBoard = Chessboard(newRows, newColumns)
        topRightBoard = Chessboard(newRows, newColumns)
        bottomLeftBoard = Chessboard(newRows, newColumns)
        bottomRightBoard = Chessboard(newRows, newColumns)
        topLeftBoard.FindPathList()
        topRightBoard.FindPathList()
        bottomLeftBoard.FindPathList()
        bottomRightBoard.FindPathList()
        ## Build new path list from bottom left, makes sense for coordinates
        bottomLeftPL = bottomLeftBoard.GetPathList()
        bottomRightPL = bottomRightBoard.GetPathList()
        topLeftPL = topLeftBoard.GetPathList()
        topRightPL = topRightBoard.GetPathList()
        newBottomLeftPL = {}
        newBottomRightPL = {}
        newTopLeftPL = {}
        newTopRightPL = {}
        ## In this step, we update the coordinates of every sub-path list
starting from the bottom left PL.
        for position in bottomLeftPL:
            newBottomLeftPL[position] = []
            for nextSquare in bottomLeftPL[position]:
                newBottomLeftPL[position].append(nextSquare)
        for position in bottomRightPL:
            ## Add columns of bottom-left to column indeces of bottom-right
            newPosition = tuple(map(lambda i, j: int(i + j), position,
(bottomLeftBoard.GetColumns(), 0)))
            newBottomRightPL[newPosition] = []
            for nextSquare in bottomRightPL[position]:
                ## Add columns of bottom-left to column indeces of bottom-
right
                newNextSquare = tuple(map(lambda i, j: int(i + j), nextSquare,
(bottomLeftBoard.GetColumns(), 0)))
                newBottomRightPL[newPosition].append(newNextSquare)
        for position in topLeftPL:
            ## Add rows of bottom-left to row indeces of top-left
            newPosition = tuple(map(lambda i, j: int(i + j), position, (0,
bottomLeftBoard.GetRows())))
            newTopLeftPL[newPosition] = []
            for nextSquare in topLeftPL[position]:
                ## Add rows of bottom-left to row indeces of top-left
                newNextSquare = tuple(map(lambda i, j: int(i + j), nextSquare,
(0, bottomLeftBoard.GetRows())))
                newTopLeftPL[newPosition].append(newNextSquare)
        for position in topRightPL:
            ## Add rows & columns to top-right
            newPosition = tuple(map(lambda i, j: int(i + j), position,
(bottomLeftBoard.GetColumns(), bottomLeftBoard.GetRows())))
            newTopRightPL[newPosition] = []
            for nextSquare in topRightPL[position]:
                ## Add rows & columns to top-right
                newNextSquare = tuple(map(lambda i, j: int(i + j), nextSquare,
(bottomLeftBoard.GetColumns(), bottomLeftBoard.GetRows())))
                newTopRightPL[newPosition].append(newNextSquare)

        ## We have to delete edges A,B,C,D and replace them with edges
E,D,F,G.
        ## To do so, we have to find the coordinate in the chessboard.

                                           ## Relevant squares in 8x8 (into a
16x16):
        A1 = (bottomLeftBoard.GetColumns()-2, bottomLeftBoard.GetRows()+1) #A1
= (6,9)
        A2 = (bottomLeftBoard.GetColumns(),   bottomLeftBoard.GetRows()+2) #A2
= (8,10)
        B1 = (bottomLeftBoard.GetColumns()+1, bottomLeftBoard.GetRows()+1) #B1
= (9,9)
        B2 = (bottomLeftBoard.GetColumns()+2, bottomLeftBoard.GetRows()+3) #B2
= (10,11)
        C1 = (bottomLeftBoard.GetColumns()+1, bottomLeftBoard.GetRows()-1) #C1
= (9,7)
        C2 = (bottomLeftBoard.GetColumns()+3, bottomLeftBoard.GetRows()  ) #C2
= (11,8)
        D1 = (bottomLeftBoard.GetColumns()-1, bottomLeftBoard.GetRows()-2) #D1
= (7,6)
```

```
        D2 = (bottomLeftBoard.GetColumns(),   bottomLeftBoard.GetRows()  ) #D2
= (8,8)
        ## Pop edge A
        newTopLeftPL[A1].remove(A2)
        newTopLeftPL[A2].remove(A1)
        ## Pop edge B
        newTopRightPL[B1].remove(B2)
        newTopRightPL[B2].remove(B1)
        ## Pop edge C
        newBottomRightPL[C1].remove(C2)
        newBottomRightPL[C2].remove(C1)
        ## Pop edge D
        newBottomLeftPL[D1].remove(D2)
        newBottomLeftPL[D2].remove(D1)
        ## Add edge E
        newBottomLeftPL[D2].append(A1)
        newTopLeftPL[A1].append(D2)
        ## Add edge F
        newTopLeftPL[A2].append(B2)
        newTopRightPL[B2].append(A2)
        ## Add edge G
        newTopRightPL[B1].append(C2)
        newBottomRightPL[C2].append(B1)
        ## Add edge H
        newBottomRightPL[C1].append(D1)
        newBottomLeftPL[D1].append(C1)

        newCompletePL = {**newBottomLeftPL, **newBottomRightPL,
**newTopLeftPL, **newTopRightPL}
        self.SetPathList(newCompletePL)
        return
    ## Given a path list, develop the tour of the board
    def FindTour(self):
        startingPosition = (1,1)
        currentPosition = startingPosition
        pathList = self.GetPathList()
        visitedSquares = {startingPosition: True}
        tour = {}
        while True:
            foundNextStep = False
            for square in pathList[currentPosition]:
                if square not in visitedSquares:
                    visitedSquares[square] = True
                    tour[currentPosition] = square
                    currentPosition = square
                    foundNextStep = True
                    break
            ## If all next squares have already been visited...
            if not foundNextStep:
                if tuple(map(lambda i,j: i - j, currentPosition,
startingPosition)) in KnightMoves:
                    tour[currentPosition] = startingPosition
                    break
                ## ... or the tour is broken.
                else:
                    print("ERROR: position", currentPosition, "can't go
anywhere.")
                    break
        if len(visitedSquares) != (self.GetRows() * self.GetColumns()):
            print("Tour ends after", len(visitedSquares), "steps instead of",
self.GetRows() * self.GetColumns() ,".")
        self.SetTour(tour)
    def PrintTour(self):
        nextStep = self.GetTour()
        startingPosition = (1,1)
        currentPosition = nextStep[startingPosition]
        tourMatrix = np.zeros((self.GetRows(),self.GetColumns()),int)
        visitedPositions = {startingPosition: 1}
        tourMatrix[self.GetRows()-startingPosition[1],startingPosition[0]-1] =
1
        positionCounter = 1
        while currentPosition !=  startingPosition:
            positionCounter += 1
            visitedPositions[currentPosition] = positionCounter
            tourMatrix[int(self.GetRows()-
currentPosition[1]),int(currentPosition[0]-1)] = positionCounter
            currentPosition = nextStep[currentPosition]
        print(tourMatrix)
    def TourIsComplete(self):
        return len(self.GetTour()) == self.GetColumns() * self.GetRows()
    def TourIsLegal(self):
        tourNextStep = self.GetTour()
        previousPosition = (1,1)
        currentPosition = tourNextStep[previousPosition]
        isLegal = True
        for i in range(self.GetColumns() * self.GetRows()):
            if tuple(map(lambda i,j: i - j, currentPosition,
previousPosition)) not in KnightMoves:
                isLegal = False
                break
            previousPosition = currentPosition
            currentPosition = tourNextStep[currentPosition]
        return isLegal
    def CheckTour(self):
        print("Complete: \t", self.TourIsComplete())
        print("Legal: \t\t", self.TourIsLegal())
```

```
n = 8
board = Chessboard(n,n)
print(board.GetRows())
print(board.GetColumns())
print("Building knight path list...")
board.FindPathList()
print("Building knight tour...")
print()
board.FindTour()
board.PrintTour()
```

```
Building knight path list...
Building knight tour...

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

```
n = 128
board = Chessboard(n,n)
print("Building knight path list...")
board.FindPathList()
print("Building knight tour...")
print()
board.FindTour()
board.PrintTour()
```

```
Building knight path list...
Building knight tour...

[[ 3954  3973  3938 ...  7891  7916  7873]
 [ 3937  3970  3953 ...  7872  7877  7890]
 [ 3974  3955  3972 ...  7889  7874  7917]
 ...
 [16357     2 16365 ... 12118 12153 12120]
 [    4 16343 16384 ... 12155 12132 12129]
 [    1 16358     3 ... 12130 12119 12154]]
```

```
print("Path length:", len(board.GetPathList()))
print("Tot. squares:", board.GetColumns() * board.GetRows())
```

```
Path length: 16384
Tot. squares: 16384
```

CONCLUSION:

The warnsdorff heuristic though solves Hamiltonian path problem does not perform very well in case of large values of size of chess board(knight’s tour). The divide and conquer algorithm could be readily implemented in parallel. However, the algorithm is only able to find closed knight's tours on n×n, n×(n+1) or n×(n+2) boards when n⩾10. And does not perform well when dealt with boards of any arbitrary size. Various solutions with neural networks have also been proposed to solve the problem which enable parallel computation.

REFERENCES:

1. https://iq.opengenus.org/knights-tour-problem/
2. https://www.iiitb.ac.in/Ganaka/projects/Chitrakavya/downloads/00281427.pdf
3. https://www.codesdope.com/course/algorithms-knights-tour-problem/
4. https://link.springer.com/chapter/10.1007/978-981-13-5802-9_16
5. https://sites.science.oregonstate.edu/math_reu/proceedings/REU_Proceedings/Proceedings2004/2004Ganzfried.pdf
6. https://en.wikipedia.org/wiki/Knight%27s_tour#:~:text=Warnsdorff%27s%20rule%20is%20a%20heuristic,revisit%20any%20square%20already%20visited