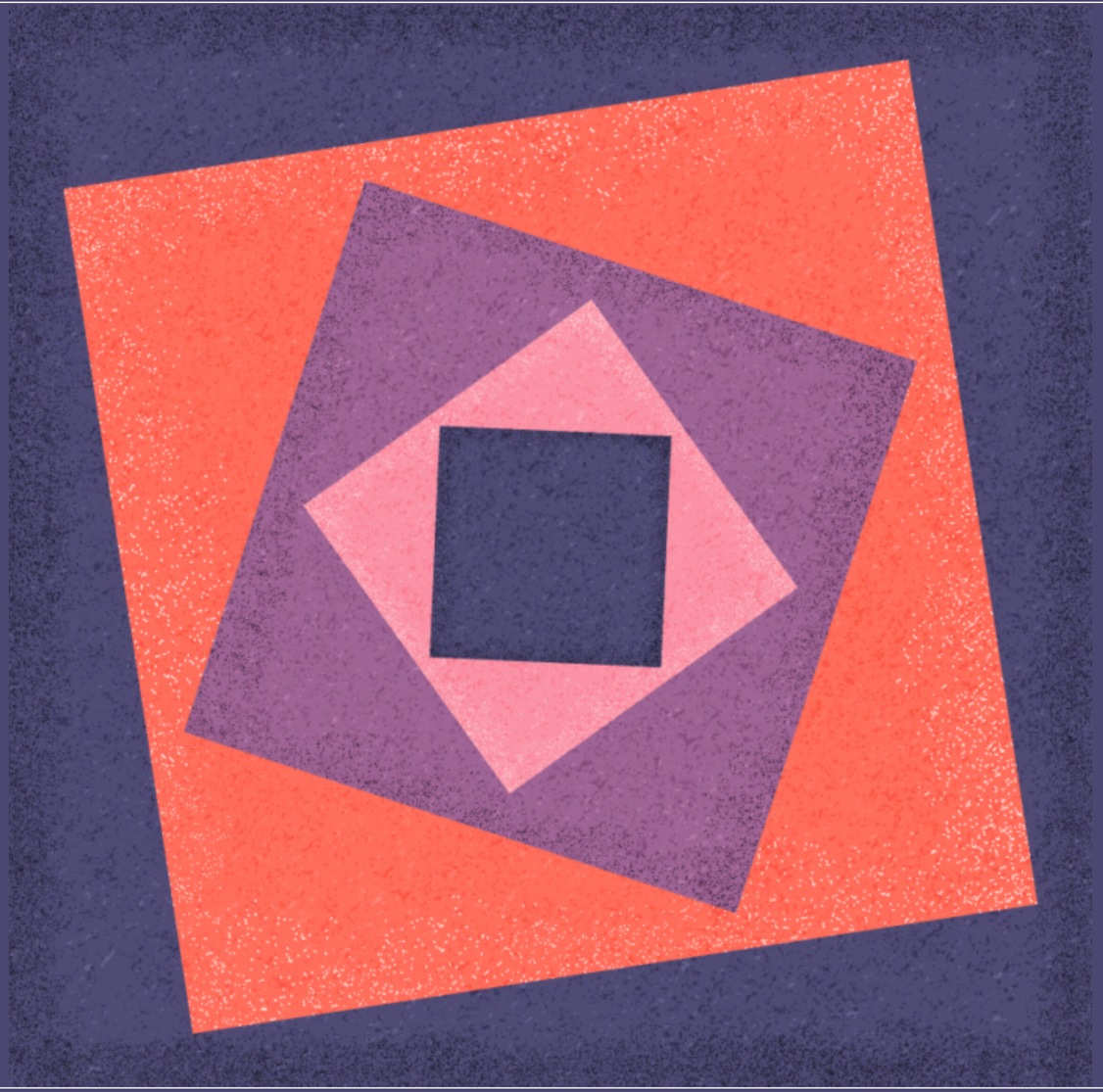
Hamiltonian Knight Tours

By Devin, Josh and Vikram



Introduction

- A knight can move 2 tiles one direction, 1 tile the perpendicular direction and vice versa
- Tile == vertex, legal move == edge

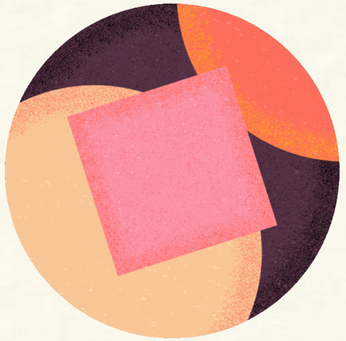
Inspiring questions

- With a standard 8 x 8 chess board and only legal moves, can every space be visited?
- Are there multiple ways to do so?
- Can we find a cycle (closed tour) where we return to same tile after?
- What about an n * n board?
- An m * n board?
- Can we apply this to an irregular shaped board? 3d board?

01 Define board shape + dimensions

02 Specify movement rules and goal

03 Find best path

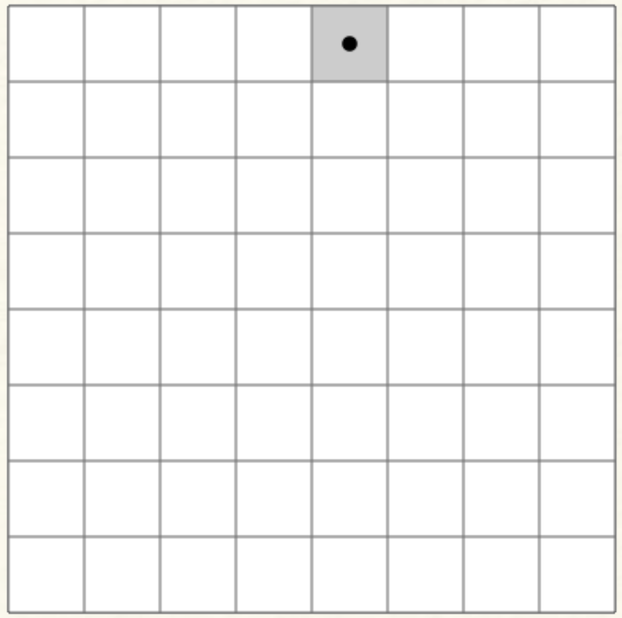


Problem History

- Ancient Arabic and Indian Manuscripts show it has been a proposed problem since the 9th century(chess born 6th century)
- Leonard Euler addressed this question in 1759 and presented his own solution
- 1823 Warnsdorff's rule was the 1st algorithm/procedure proposed
- Allen Schwenk shows which rectangular patterns work or not
- Cull and Conrad Etal similar conclusions

Helpful bits

- Open == Hamiltonian Path
- Closed == Hamiltonian Circuit
- For a closed tour # of black and white squares MUST be equal



OPEN Knight's Tour

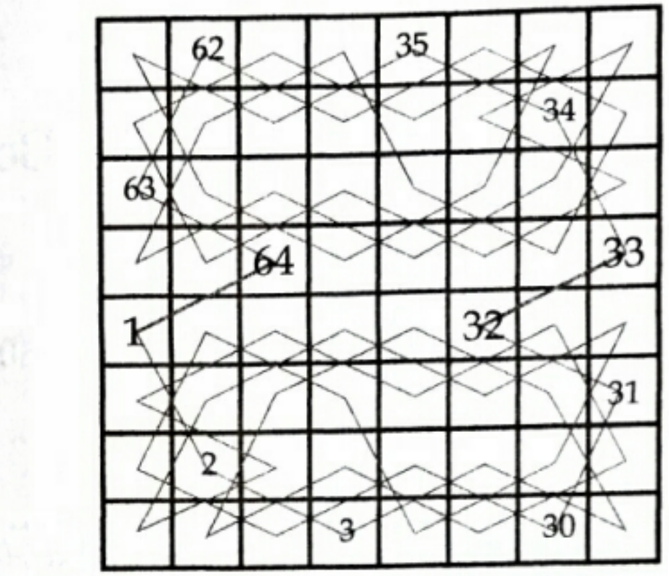
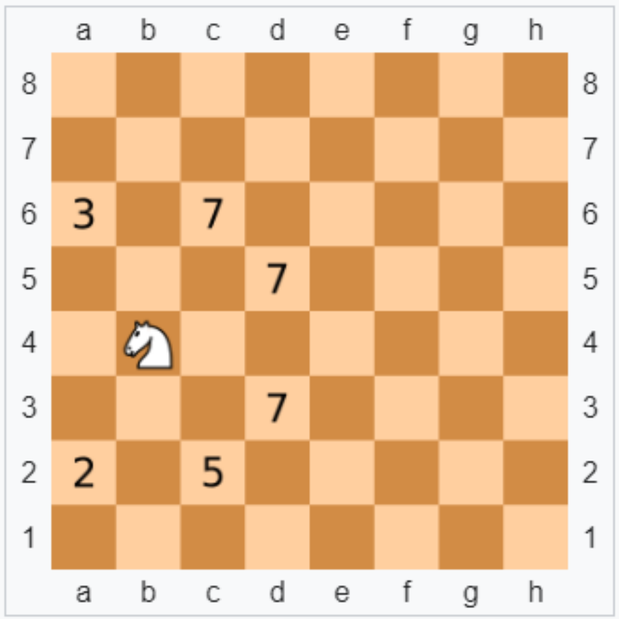
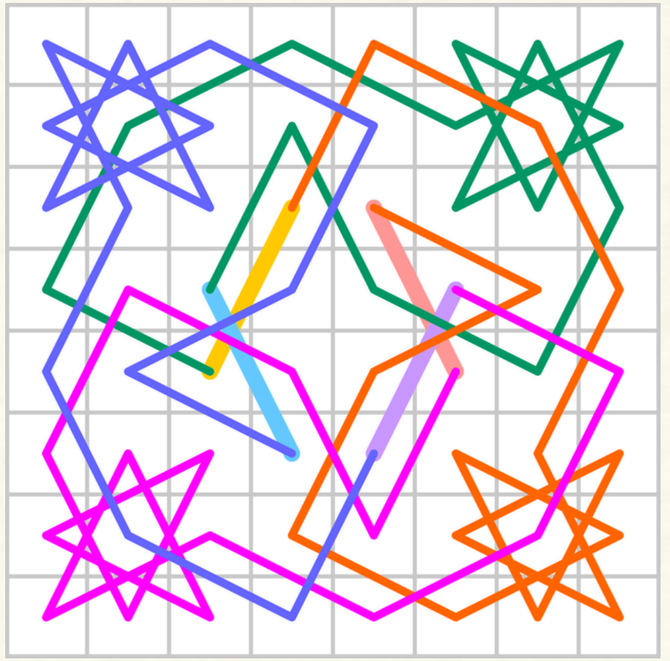


Figure 1: a knight's tour found by Euler



Warnsdorff's rule

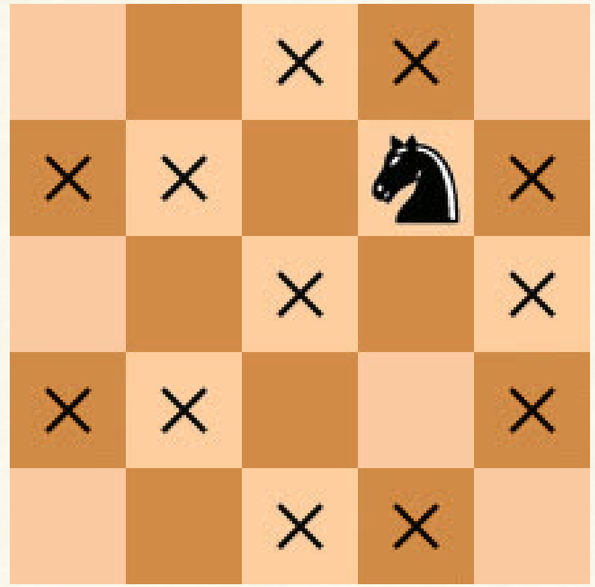


Problem History

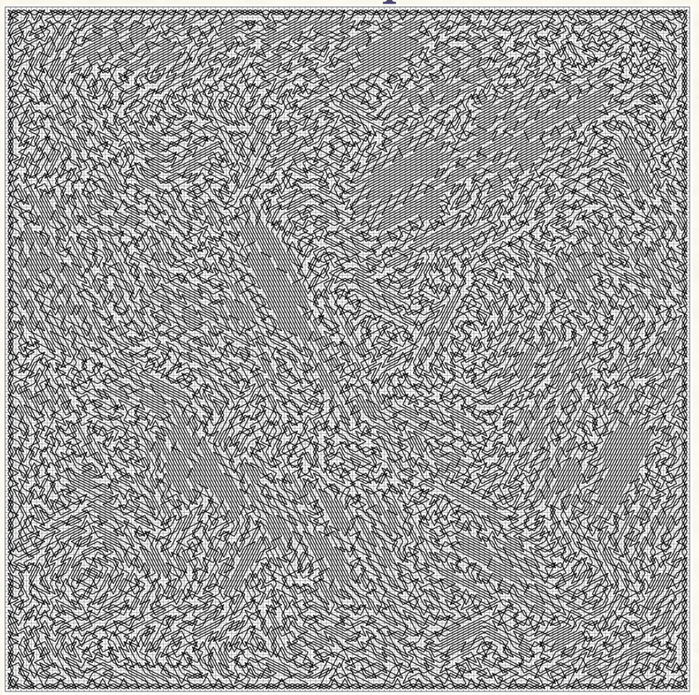
- Schwenk Proved: for any m × n board with m ≤ n, a **closed knight's tour** is always possible if none of these conditions are met:
  - m and n are both odd
  - m = 1, 2, or 4
  - m = 3 and n = 4, 6, or 8.
- Cull and Conrad proved that on any rectangular board whose smallest dimension is at least 5, there is a knight's tour. As well that for any m × n board with m ≤ n, a **knight's tour** is always possible if none of these conditions are met:
  - m = 1 or 2
  - m = 3 and n = 3, 5, or 6
  - m = 4 and n = 4

n	Number of directed tours (open and closed) on an *n* × *n* board (sequence A165134 in the OEIS)
1	1
2	0
3	0
4	0
5	1,728
6	6,637,920
7	165,575,218,320
8	19,591,828,170,979,904

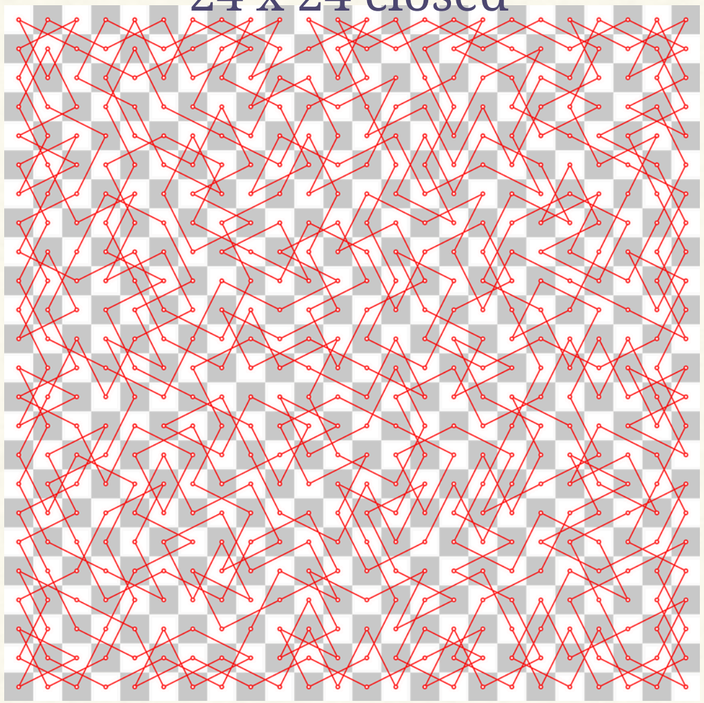
5 x 5 open



130 x 130 open



24 x 24 closed



Related Rules

Warnsdorff ‘s Rule

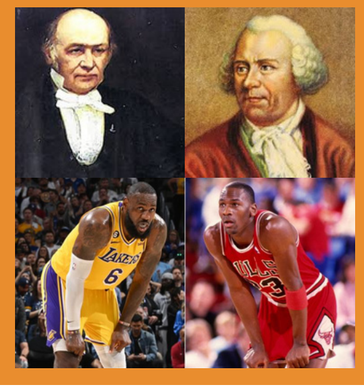
In 1823, Warnsdorff proposed a simple heuristic for finding knight’s tours. Warnsdorff’s Rule: Always move to an adjacent, unvisited square with minimal degree. Intuitively this seems like a logical rule to follow, since squares with lower degrees have fewer neighbors and therefore we will have fewer opportunities to visit them in the remainder of the path. It is essential to follow this rule if a square has degree 0, since otherwise we can never visit it. Similarly if we fail to visit a square of degree 1, then we will have only one more opportunity to visit it (and it must be the last square of the tour). It is also true that no tour can deviate from Warnsdorff’s Rule in the last several moves, although this is less obvious.

Coloring Rule

There exists a special case when considering odd numbers of N, for NxN matrices. You are bounded by your choice of starting position. The definition of an odd number is: 2k + 1, where k is an integer. The formula for the number of white and black squares in a chess board is as follows: Black = floor[(n*n)/2], White = (n*n) - floor[(n*n)/2]. If we use our defintion of an odd number in place of n, we find this: Black = floor(1/2(4k^2 + 4k + 1)), White = 4k^2 + 4k + 1 - Black, Black = 2k^2 + 2k, White = 2k^2+2k+1. This clearly shows that for an odd number of n, we will have one more white cell than black. Each knights move must go from black to white or white to black, thus we must start on the color with an additional cell. More generally we must start on a cell that lies on the diagonal or its subsequent alternating cells to find a knights tour.

Ira Pohl Tie Breaking

Ira Pohl (1996): “For each tie move, sum the number of moves available to it at the next level and pick the move yielding the maximum”



Average chess set:

Implementation 1:

Using a DFS algorithm with brute force guess-and-check. Made in C

How does it Work?

```
// Function to solve the Knight's tour problem
bool solve_knights_tour(int m, int n, int start_corner) {
    int **board = (int **)malloc(m * sizeof(int *));
    for (int i = 0; i < m; i++) {
        board[i] = (int *)malloc(n * sizeof(int));
    }

    // Initialization of the chessboard
    for (int x = 0; x < m; x++) {
        for (int y = 0; y < n; y++) {
            board[x][y] = -1;
        }
    }

    // All possible moves for a knight
    int move_x[8] = { 2, 1, -1, -2, -2, -1, 1, 2 };
    int move_y[8] = { 1, 2, 2, 1, -1, -2, -2, -1 };

    // Determine starting position based on input corner
    int start_x, start_y;
    switch (start_corner) {
        case 0: start_x = 0; start_y = 0; break;
        case 1: start_x = 0; start_y = n - 1; break;
        case 2: start_x = m - 1; start_y = 0; break;
        case 3: start_x = m - 1; start_y = n - 1; break;
        default: printf("Invalid starting corner\n"); return false;
    }

    // Starting position
    board[start_x][start_y] = 0;

    // Prompt user before starting the solution process
    printf("This might take a while. Do you want to proceed? (y/n): ");
    char response;
    scanf(" %c", &response);
    if (response != 'y' && response != 'Y') {
        printf("Aborted by user.\n");
        // Free allocated memory before exiting
        for (int i = 0; i < m; i++) {
            free(board[i]);
        }
        free(board);
        return false;
    }
```

1. Create an empty 2D based on inputted size
2. Set of move rules are defined in a list
3. Choose starting corner, count time, begin pathing
4. We parse through the list of 8 legal moves and select the 1st valid one. If not legal, go to 2nd, 3rd… If square is visited, mark it.
5. Backtrack (DFS) if no possible squares
6. If we run out of backtracks(back to 0), we fail and program stops.

```
// Utility function to solve the Knight's tour problem using backtracking
bool solve_knights_tour_util(int x, int y, int movei, int **board, int move_x[], int move_y[], int m, int n) {
    int k, next_x, next_y;
    if (movei == m * n) {
        return true;
    }

    // Try all next moves from the current coordinate x, y
    for (k = 0; k < 8; k++) {
        next_x = x + move_x[k];
        next_y = y + move_y[k];
        if (is_safe(next_x, next_y, board, m, n)) {
            board[next_x][next_y] = movei;
            if (solve_knights_tour_util(next_x, next_y, movei + 1, board, move_x, move_y, m, n)) {
                return true;
            } else {
                // Backtracking
                board[next_x][next_y] = -1;
            }
        }
    }

    return false;
}
```

```
    // Start from the determined starting square
    bool success = solve_knights_tour_util(start_x, start_y, 1, board, move_x, move_y, m, n);

    // Measure end time
    clock_t end_time = clock();
    double elapsed_time = (double)(end_time - start_time) / CLOCKS_PER_SEC;

    if (!success) {
        printf("Solution does not exist\n");
    } else {
        print_solution(board, m, n);
    }
```

```
29 14 23  6 27  2
22  7 28  3 18  5
15 10 13 24  1 26
12 21  8 17  4 19
 9 16 11 20 25  0
Run time: 0.01 seconds
```

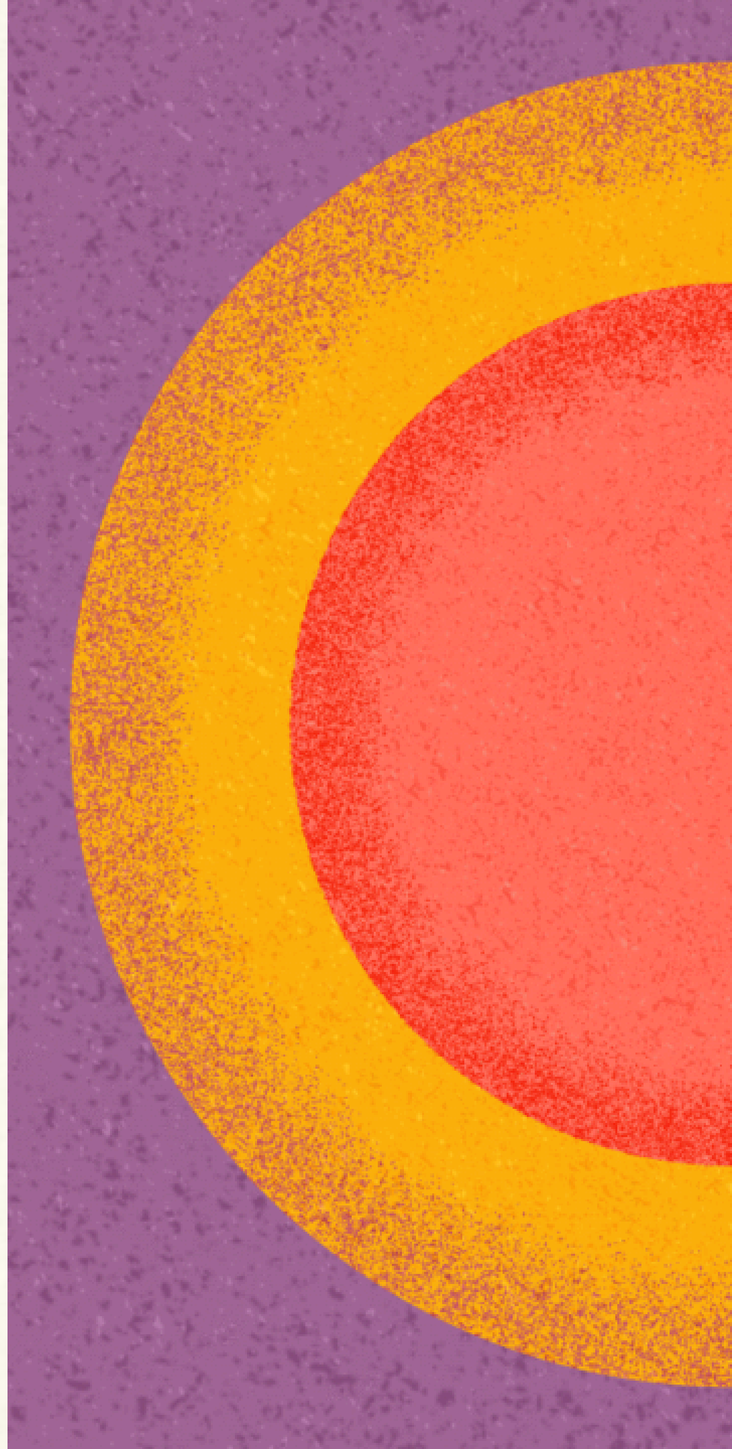
```
 0 27 22 15  6 11
23 16  7 12 21 14
28  1 26 19 10  5
17 24  3  8 13 20
 2 29 18 25  4  9
Run time: 2.38 seconds
```

```
vikram@ix-dev: ~/discret
This might take a while.
 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
Run time: 0.53 seconds
```

```
vikram@ix-dev: ~/discret
This might take a while.
46 33 20 29  4 13 18 63
35 30 47 32 19 62  3 12
48 45 34 21 28  5 14 17
59 36 31 50 61 16 11  2
44 49 60 37 22 27  6 15
55 58 41 24 51  8  1 10
40 43 56 53 38 23 26  7
57 54 39 42 25 52  9  0
Run time: 14.48 seconds
```

Matrix Overlays:

A geometric solution



How does it Work?

This algorithm implements Warnsdorff's Rule, The Coloring rule, and Ira Pohl's Tie Breaker to solve Knights tours using matrices. It determines effective start positions using the coloring rule and layers matrices to determine moves of minimum degree, with ties being severed via Ira Pohl's Rule.

Successes and Improvements:

This implementation is very quick to provide results in fact so quick that it can produce an open tour of a 100 x 100 chess board in less than three seconds and a 400x400 open tour in around 30 seconds. This is due to the selection proccess and the tie breaker method. The classical approach to this problem utilizes the Warnsdorff heuristic and a random tie-breaker which is extremely effective in small dimensions but fails as values of n grow. (see image)

The classical solution to this problem has a near zero likelihood of producing a open tour with a 400x400 sized chessboard, however this implementation produced one in thirty seconds (with a clever starting position). While this method is very effective in producing paths of large boards quickly it does not necessarily find a path every time in large dimensions. Some improvements to this could be using an even more effective tie breaker, such as Squirrel and Cull's Tie breaker.

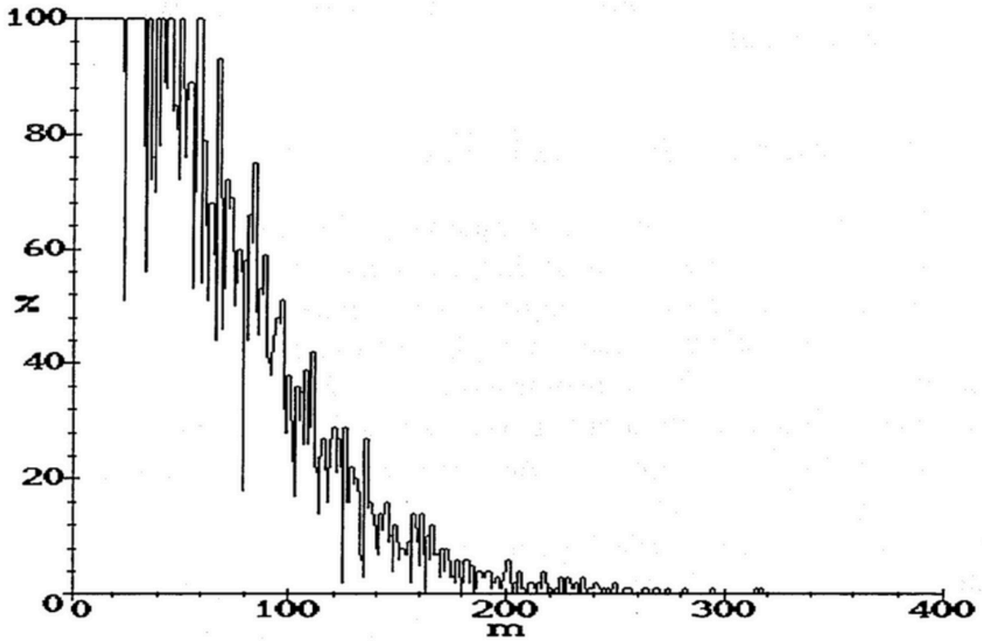


Figure 2: Success rate of random tiebreak rule.

In the case of this 5x5 matrix the coloring rule applies so we pick a white cell to start.

0	0	0	0	0
0	0	0	0	0
0	0	0	0	0
0	0	0	0	0
0	0	0	0	0

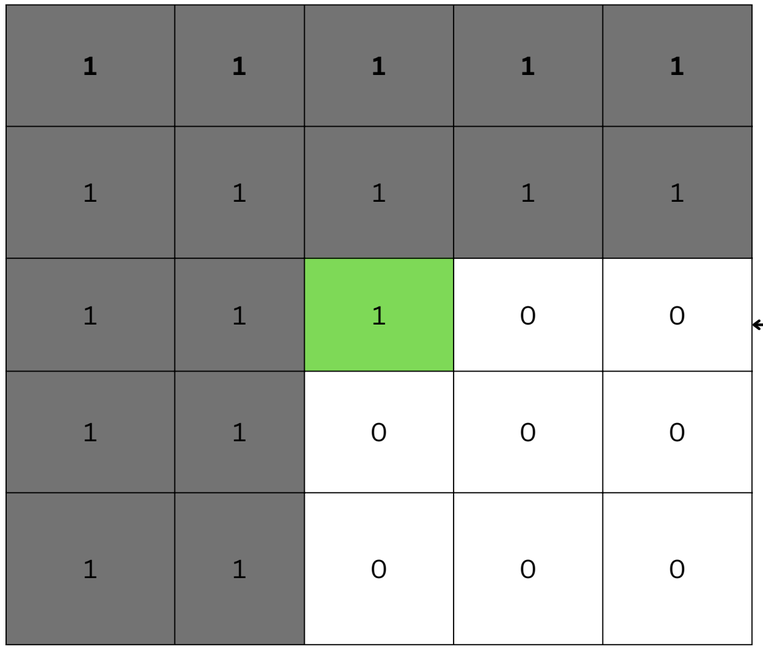
Initialize with a blank 5x5 matrix (representing a checker board)
Each zero represents an unfilled square on the checker board

1	1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1	1
1	1	0	0	0	0	0	1	1
1	1	0	0	0	0	0	1	1
1	1	0	0	0	0	0	1	1
1	1	0	0	0	0	0	1	1
1	1	0	0	0	0	0	1	1
1	1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1	1

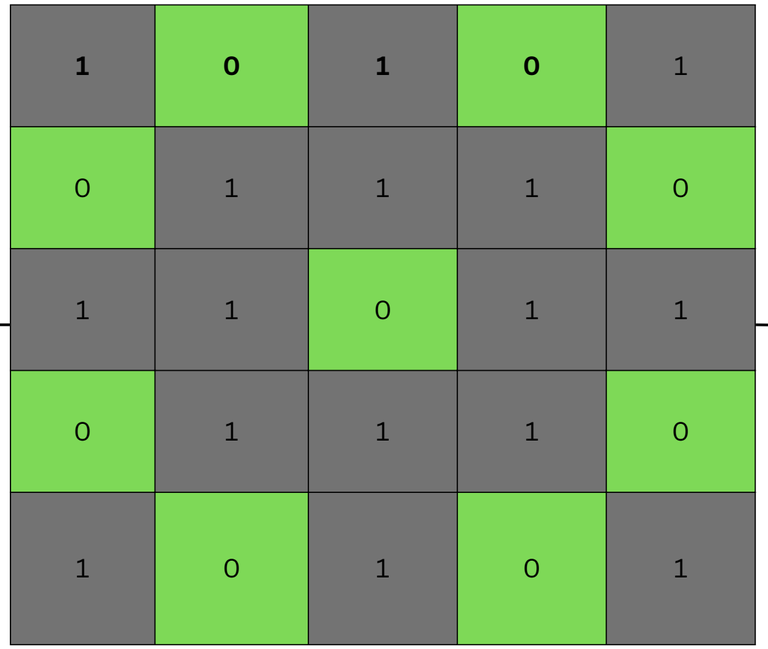
Overlay the checkerboard with a border of ones

1	1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1	1
1	1	1	0	0	0	0	1	1
1	1	0	0	0	0	0	1	1
1	1	0	0	0	0	0	1	1
1	1	0	0	0	0	0	1	1
1	1	0	0	0	0	0	1	1
1	1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1	1

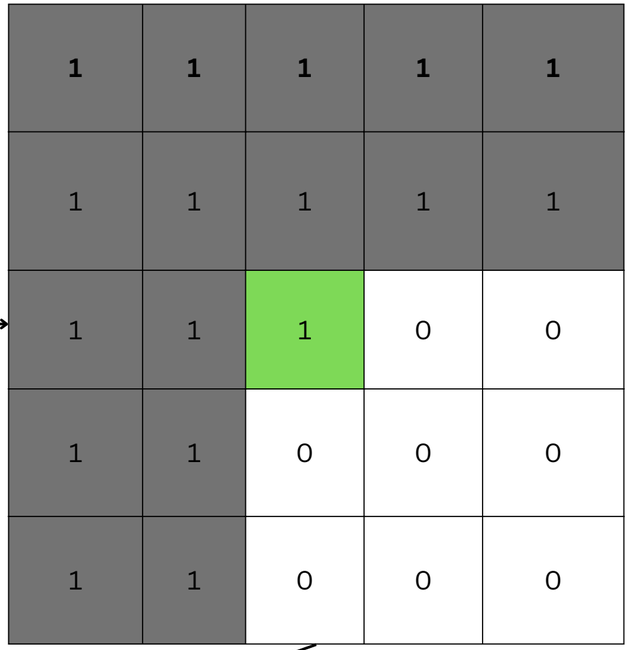
Select an arbitruary starting point,
or a blank check on the checker board
and mark it with a one, representing our first move

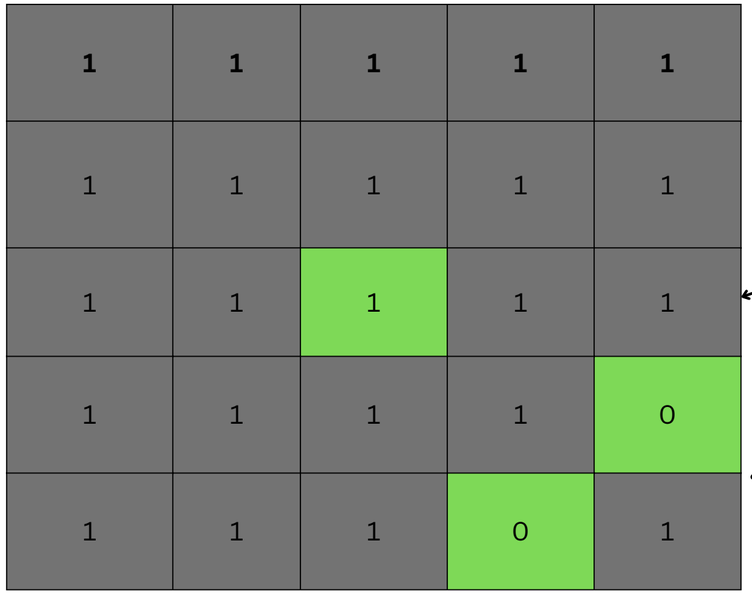


Take a 5x5 sub-matix centered around the selected point



This 5x5 matrix represents all the possible knights moves from the center position.





Now we will logically “Or” the two matrices together to determine possible moves from our selected starting poistion

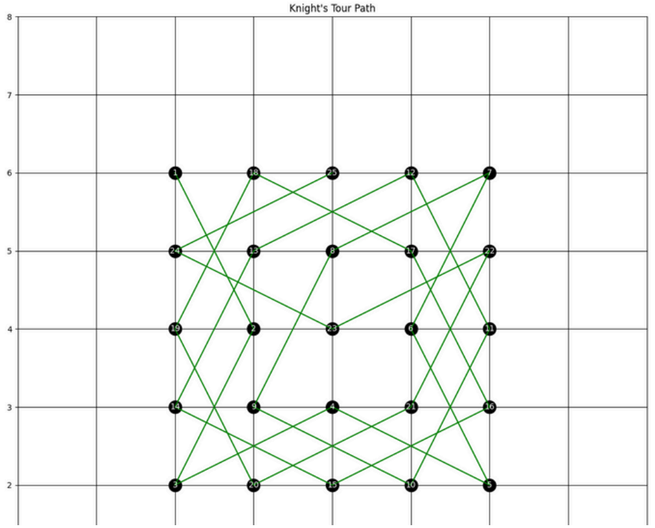
1	1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1	1
1	1	1	0	0	0	0	1	1
1	1	0	0	0	0	0	1	1
1	1	0	0	0	0	0	1	1
1	1	0	0	0	0	0	1	1
1	1	0	0	0	0	0	1	1
1	1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1	1

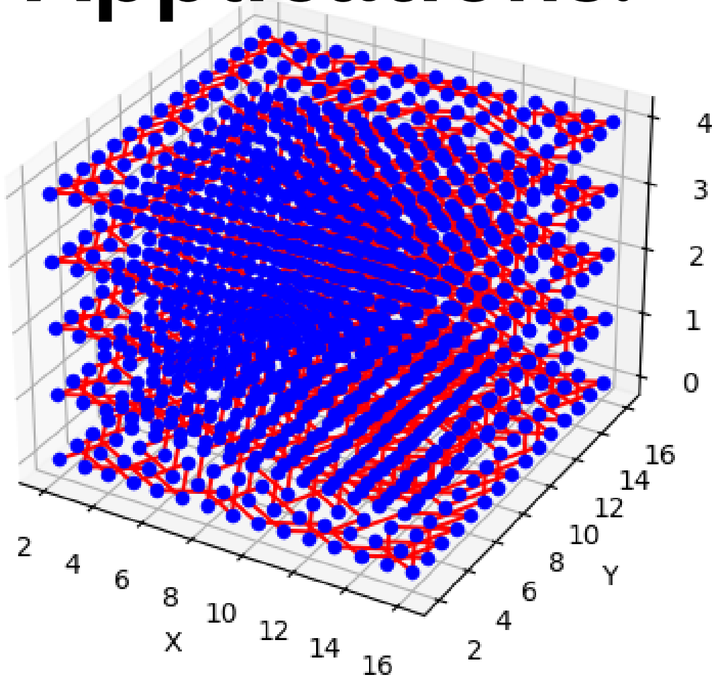
1	1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1	1
1	1	1	0	0	0	0	1	1
1	1	0	0	2	0	0	1	1
1	1	0	0	0	0	0	1	1
1	1	0	0	0	0	0	1	1
1	1	0	0	0	0	0	1	1
1	1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1	1

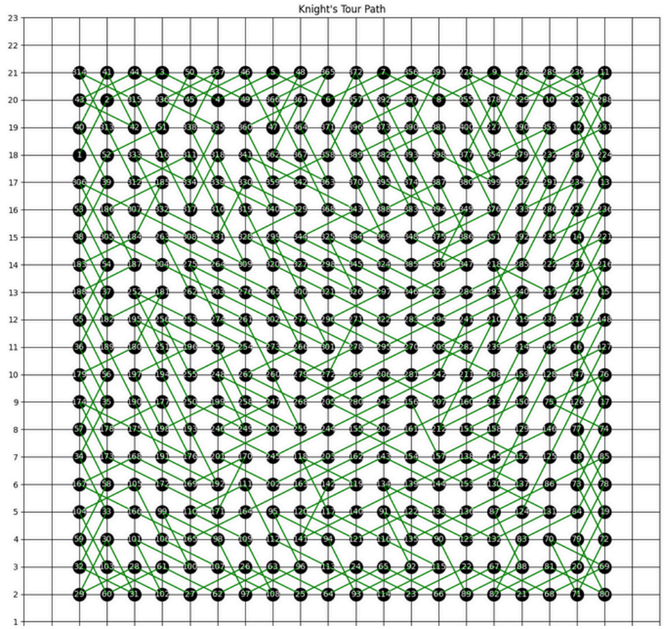
Now we have determined the possible moves from our arbitruary starting position,
next we would repeate the last two steps on the found possible moves, to determine
which potential move is the least accesible. (Warnsdorff)

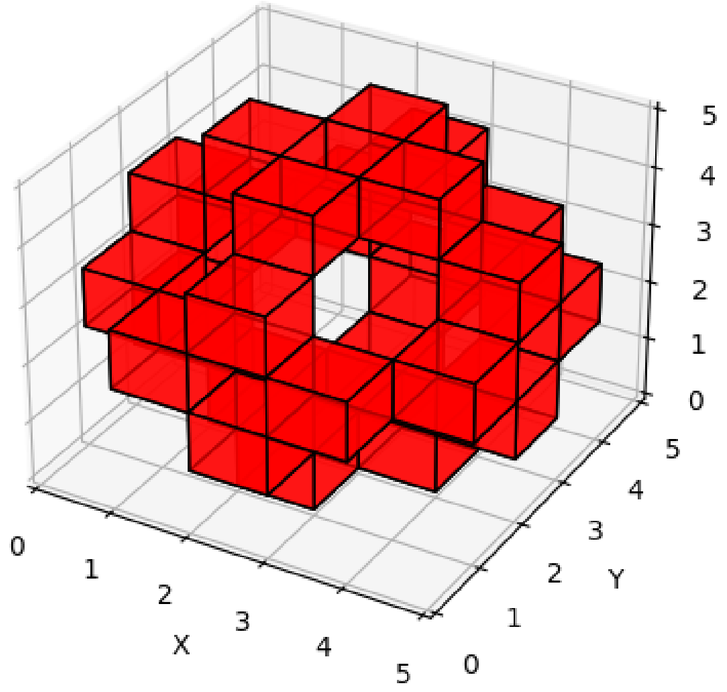
Since each of these available cells have degree = 6 - 1 (the already filled cell), we will employ Ira Pohl's tie-breaker method. (since this is an empty board, and the available cells are identical it does not matter which you select (by Ira Pohl they each have degree sums of 18)

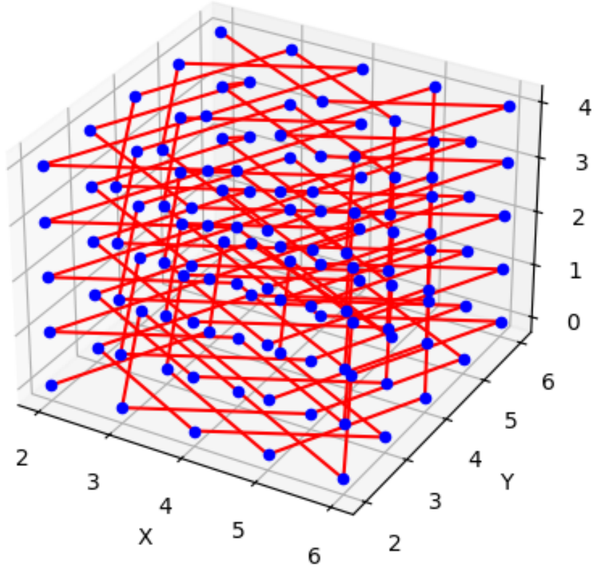
Visualizations and Applications:

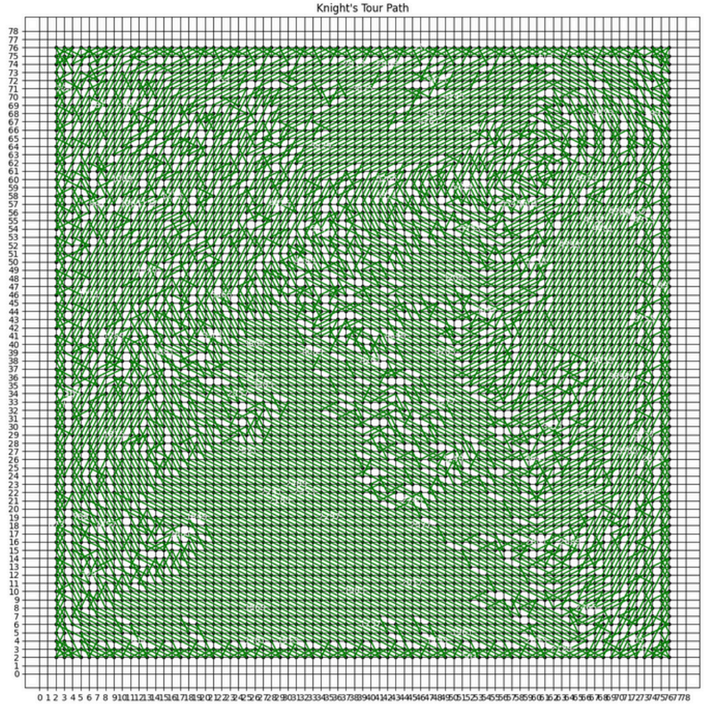












Implementation 3:

OOP Centered Program

- Creates abstract data types that are personalized for use and flexibility
- Allows for closer inspection at each step
- Fails gracefully
- Finding a knight tour is no longer the purpose of the code, but rather just a part of a ‘graph’ data type
    -Can implement multiple path finding algorithms to compare/analyze

Step 1
Initializing

```python
class Node:
    def __init__(self, cord: tuple):
        self.cord = cord
        self.valid = True
        self.degree = 0
        self.connections = []


    def __repr__(self):
        return f'Node({self.cord})'

    def __str__(self):
        return f'Node @ row: {self.cord[0]}| col: {self.cord[1]}'

    def __lt__(self, other):
        return self.degree < self.degree

    def set_connection(self, other):
        if other not in self.connections:
            self.connections.append(other)
            self.degree += 1

    def remove_connection(self, other):
        if other in self.connections:
            self.connections.remove(other)
            self.degree -= 1

    def sort_connections(self):
        def sort_key(node):
            return node.degree
        self.connections.sort(key=sort_key)
```

```python
class Board:
    def __init__(self, array: list[list]):
        self.array = array

        self.valid_nodes = [] #this stores the vertex set
        self.edges = set() #this stores the edge set
        self.dictionary = {} #this stores the conversion between nodes and their cord
        self.order = 0
        self.size = 0
        self.set_board()


    def set_edge(self, node1, node2):
        if node1.valid and node2.valid:
            node1.set_connection(node2)
            node2.set_connection(node1)
            self.edges.add((node1,node2))
            self.size += 1

    def remove_edge(self, edge:tuple):
        edge[0].remove_connection(edge[1])
        edge[1].remove_connection(edge[0])

    def sort_nodes(self):
        for node in self.valid_nodes:
            node.sort_connections()
        def sort_key(node):
            return node.degree
        self.valid_nodes.sort(key=sort_key)



    def set_board(self):
        size = (len(self.array), len(self.array[0]))
        for row in range(size[0]):
            for col_index in range(size[1]):
                cord = (row, col_index)
                node = Node(cord)
                self.dictionary[cord] = node
                if self.array[row][col_index] == 1:
                    node.valid = False

                else:
                    self.valid_nodes.append(node)
                    self.order +=1
                    if ((cord[0] >= 2) and (cord[1] >= 1)):
                        self.set_edge(node, self.dictionary[(cord[0]-2,cord[1]-1)] )

                    if ((cord[0] >= 1) and (cord[1] >= 2)):
                        self.set_edge(node, self.dictionary[(cord[0]-1,cord[1]-2)] )

                    if ((cord[0] >= 2) and (cord[1] <= size[1]-2)):
                        self.set_edge(node, self.dictionary[(cord[0]-2,cord[1]+1)] )

                    if ((cord[0] >= 1) and (cord[1] <= size[1]-3)):
                        self.set_edge(node, self.dictionary[(cord[0]-1,cord[1]+2)] )

        self.sort_nodes
```

What it does

‘Board’ concept

[[1,1,1,0,0,1,1,1],
[1,1,1,0,0,1,1,1],
[1,1,0,0,0,0,1,1],
[0,0,0,0,0,0,0,0],
[1,0,0,0,0,0,0,1],
[1,1,0,0,0,0,1,1],
[1,0,0,1,1,0,0,1],
[1,1,1,1,1,1,1,1]]

Input ‘Board’

```
================ RESTART: /Users/josh/Desktop/Knight Tour/Knight_tour.py ================
>>> demo = Board([[1,1,1,0,0,1,1,1], [1,1,1,0,0,1,1,1], [1,1,0,0,0,0,1,1], [0,0,0,0,0,0,0,0],
[1,0,0,0,0,0,0,1], [1,1,0,0,0,0,1,1], [1,0,0,1,1,0,0,1],[1,1,1,1,1,1,1,1]])
```

```
>>> demo.valid_nodes
[Node((0, 3)), Node((0, 4)), Node((1, 3)), Node((1, 4)), Node((2, 2)), Node((2, 3)), Node((2, 4)),
 Node((2, 5)), Node((3, 0)), Node((3, 1)), Node((3, 2)), Node((3, 3)), Node((3, 4)), Node((3, 5)),
 Node((3, 6)), Node((3, 7)), Node((4, 1)), Node((4, 2)), Node((4, 3)), Node((4, 4)), Node((4, 5)),
 Node((4, 6)), Node((5, 2)), Node((5, 3)), Node((5, 4)), Node((5, 5)), Node((6, 1)), Node((6, 2)),
 Node((6, 5)), Node((6, 6))]
>>>
```

Abstract data representation of ‘Board’

Step 2
Define an algorithm

```
def longest_path(self):
    '''returns longest path found'''


    def long(options, used, wanted)->list:
        amt_opt = len(options)
        if amt_opt == 0:
            return []


        longest_length = 0
        new_addition =[]
        for attempt in range(amt_opt):
            node = options[attempt]
            if node not in used:
                path = [node]
                new_used = used.copy()
                new_used.add(node)
                available = [x for x in node.connections if x not in new_used]
                children = long(available, new_used, wanted)
                for child in children:
                    path.append(child)
                path_len = len(path)

                if path_len == wanted:
                    return path

                if path_len>longest_length:
                    longest_length = path_len
                    new_addition = path


        return new_addition


    return long(self.valid_nodes, set(), self.order)
```

How it works

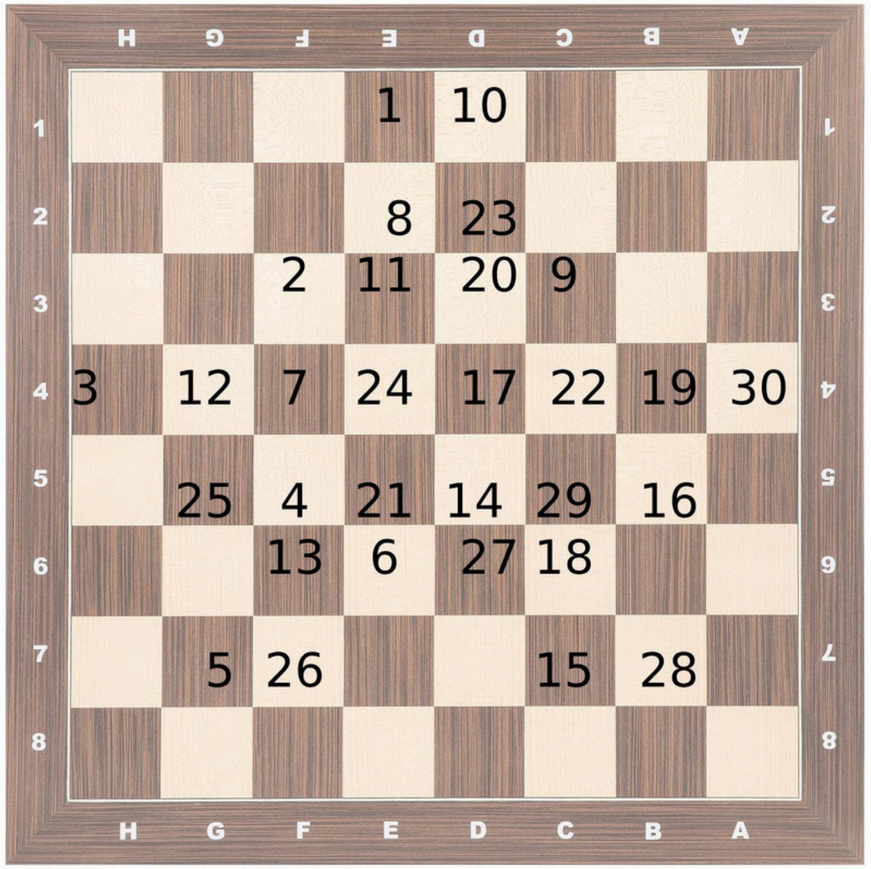
Let L be a list of all vertices, sorted by degree(least to greatest).
Set all vertices in L as unmarked.
Let P be the longest path found so far.
Let Cur be the current path.

Start recursive algorithm:

Base case: If L is empty; return P

Else, for each vertex, V, in L:
-If V is unmarked:
  --Add V to Cur
  --Mark V
  --Call algorithm with L as the neighbors of V(also sorted)
  --Add the result to Cur
  --If Cur is longer than P, P becomes Cur

Return P




```
>>> demo.longest_path()
[Node((0, 3)), Node((2, 2)), Node((3, 0)), Node((4, 2)), Node((6, 1)), Node((5, 3)), Node
((3, 2)), Node((1, 3)), Node((2, 5)), Node((0, 4)), Node((2, 3)), Node((3, 1)), Node((5,
2)), Node((4, 4)), Node((6, 5)), Node((4, 6)), Node((3, 4)), Node((5, 5)), Node((3, 6)),
Node((2, 4)), Node((4, 3)), Node((3, 5)), Node((1, 4)), Node((3, 3)), Node((4, 1)), Node(
(6, 2)), Node((5, 4)), Node((6, 6)), Node((4, 5)), Node((3, 7))]
```

Comparing 3 Implementations

	Speed	Flexibility
Implementation 1: Naive Algorithm	- Brute force -Takes the longest time -Struggles with large boards -Biased by starting point	-Limited boards - Must start in corners
Implementation 2: Smart Algorithm	-Highly efficient -Applies multiple ‘rules’ to lessen time -visually intuitive	-More types of boards allowed
Implementation 3: Graph Structure	-Would have multiple algorithms of varying times	-Can take all boards, even non-Euclidean -Discrete steps allow for deeper analysis -Can do things other than finding longest path