Imagine a rectangular piece of graph paper. We could create a maze by coloring some of the lines to make walls.
We call that maze well-formed if it has exactly two exterior openings (an entrance and an exit), and exactly one path that connects the two openings.

1. Design a data structure that can represent a rectangular maze.
    An array of edges from 1 cell to another
2. Create an algorithm that determines whether the maze is well-formed.
    Recursive algorithm to get all paths:
        Start: call recursive function with an entrance edge (edge: (start coords, end coords))
        Base cases: edge already seen (cycle encountered) | end found | no unseen neighbors
        Recursive Case: recursive function with all outgoing edges from current cell
            returns array of solution arrays
            return the result, prepending current cell to each solution array
    Iterative algorithm to get all paths:
            Initialize a stack with an entrance cell coords
            initialize a dictionary with traversed edges
            while stack not empty:
                pop current cell
                add unseen accessible neighbors to stack
                add seen neighbors, if not current cell's parent, to dictionary entry
                each time end found:
                    use dictionary to reverse the current path
                        delete edges after saving in solution array
            return array of arrays of solution
3. Report the solution if the maze is well-formed.
    not well formed:
        # entrances not 2
        no solution
        solution path contains cycle
            above algorithms will return > 1 solution array if cycle exists

***** edge cases:**
    m < 1 | n < 1, entrances != 2, cycles in non-solution branches
***** if non-solution branches not allowed to have cycles, check for D.A.G. is sufficient:**
    DFS algorithm with seen dict:
        if cell in seen, return []

**** Recursive
```
def solutionRec(self, root, parent, end, seen):
    edgeAlreadyTraversed = (root, parent) in seen or (parent, root) in seen
    if not root or edgeAlreadyTraversed :
        return []

    # base case: found exit
    if root == end:
        return [[root]]

    # deep copy of traversal history, for backtracking
    newseen = seen.copy()
    newseen[(parent, root)] = True

    # Visit neighbors of current node
    allNeighbors = self.getNeighbors(root)
    ret = []
    for n in allNeighbors:
        if n == parent:
            continue
        ret += [[root] + x for x in self.solutionRec(n, root, end, newseen)]
    return ret

def allSolutionsDFSRec(self):
    ends = Maze.getEnds(self)
    if len(ends) != 2:
        return []
    seen = {}
    return self.solutionRec(ends[0], None, ends[1], seen)
```

*** Iterative
```
def allSolutionsFoundDFS(self):
    '''
    Return all map solutions found else [].
    '''
    ends = Maze.getEnds(self)  # [(row, col), (row, col)]
    if len(ends) != 2:         # Invalid maze: Not 2 entrances
        return []

    # map to preserve path order
    childparentdict = {ends[0]: [None]}

    # Stack of unvisited nodes
    toCheck = [ends[0]]
    solutions = []
    while toCheck:
        curNode = toCheck.pop()
        if curNode == ends[1]:
            solutions.append(self.calculateSolution(curNode, childparentdict))

        # Visit neighbors of current node
        allNeighbors = self.getNeighbors(curNode)
        for n in allNeighbors:
            if n not in childparentdict:      # unvisited node:
                childparentdict[n] = [curNode]
                toCheck.append(n)
            else:  # this edge has already been traversed
                if curNode in childparentdict[n] or n in childparentdict[curNode]:
                    continue
                else: # unique edge, visited node
                    childparentdict[n].append(curNode)
                    toCheck.append(n)
    return solutions
```

```
***

class Maze:
    def __init__(self, m, n, edges):
        self.m = m
        self.n = n
        self.edges = edges
        self.solution = []

    def solutionRec(self, root, parent, end, seen):
        edgeAlreadyTraversed = (root, parent) in seen or (parent, root) in seen
        if not root or edgeAlreadyTraversed :
            return []

        # base case: found exit
        if root == end:
            return [[root]]

        # deep copy of traversal history, for backtracking
        newseen = seen.copy()
        newseen[(parent, root)] = True

        # Visit neighbors of current node
        allNeighbors = self.getNeighbors(root)
        ret = []
        for n in allNeighbors:
            if n == parent:
                continue
            ret += [[root] + x for x in self.solutionRec(n, root, end, newseen)]
        return ret

    def allSolutionsDFSRec(self):
        ends = Maze.getEnds(self)
        if len(ends) != 2:
            return []
        seen = {}
        return self.solutionRec(ends[0], None, ends[1], seen)

    def calculateSolution(self, node, childparentdict):
        ret = []
        while node:
            ret.append(node)
            tempnode = childparentdict[node][-1]
            del childparentdict[node][-1]
            node = tempnode
        return ret

    def allSolutionsFoundDFS(self):
        '''
        Return all map solutions found else [].
        '''
        ends = Maze.getEnds(self)  # [(row, col), (row, col)]
        if len(ends) != 2:         # Invalid maze: Not 2 entrances
            return []

        # map to preserve path order
        childparentdict = {ends[0]: [None]}

        # Stack of unvisited nodes
        toCheck = [ends[0]]
        solutions = []
```

```python
        while toCheck:
            curNode = toCheck.pop()
            if curNode == ends[1]:
                solutions.append(self.calculateSolution(curNode, childparentdict))
            # Visit neighbors of current node
            allNeighbors = self.getNeighbors(curNode)
            for n in allNeighbors:
                if n not in childparentdict:      # unvisited node:
                    childparentdict[n] = [curNode]
                    toCheck.append(n)
                else:  # this edge has already been traversed
                    if curNode in childparentdict[n] or n in childparentdict[curNode]:
                        continue
                    else: # unique edge, visited node
                        childparentdict[n].append(curNode)
                        toCheck.append(n)
        return solutions

    @staticmethod
    def getEnds(maze):
        endsArr = []
        for i in range(maze.m):
            if (i, 0, i, -1) in maze.edges:
                endsArr.append((i, 0))
            if (i, maze.n, i, maze.n-1) in maze.edges:
                endsArr.append((i, maze.n-1))
        for i in range(maze.n):
            if (0, i, -1, i) in maze.edges:
                endsArr.append((0, i))
            if (maze.m, i, maze.m-1, i) in maze.edges:
                endsArr.append((maze.m-1, i))
        return endsArr

    @staticmethod
    def reversePathSingle(endNode, childparentdict):
        solution = [endNode]
        parent = childparentdict[endNode]
        while parent:
            solution.append(parent)
            parent = childparentdict[parent]
        return solution

    def getNeighbors(self, node):
        neighborsarr = []
        for direction in [Dir.Left, Dir.Right, Dir.Top, Dir.Bottom]:
            neighbor = self.tryGetNeighbor(node, direction)
            if neighbor:  # set parent of neighbor to curNode
                neighborsarr.append(neighbor)
        return neighborsarr

    def tryGetNeighbor(self, node, dir):
        dirNode = {
            Dir.Left: lambda x:   (x[0], x[1], x[0], x[1]-1),
            Dir.Right: lambda x:  (x[0], x[1], x[0], x[1]+1),
            Dir.Top: lambda x:    (x[0], x[1], x[0]-1, x[1]),
            Dir.Bottom: lambda x: (x[0], x[1], x[0]+1, x[1])
        }[dir](node)
        if dirNode in self.edges:
            return (self.edges[dirNode][0], self.edges[dirNode][1])
        return None
```