```
// File: ./mazeTester.py

# -------------------------------------------------------------------

# DON'T CHANGE THIS FILE.

# This is the entry point to run the program.

# Refer to usage() for exact format of input expected to the program.

#

# __author__ = 'Jeffrey Chan'

# __copyright__ = 'Copyright 2024, RMIT University'

# -------------------------------------------------------------------



import sys

import time

import json

from typing import List


from maze.util import Coordinates

from maze.maze import Maze

from maze.arrayMaze import ArrayMaze

from maze.graphMaze import GraphMaze


from generation.mazeGenerator import MazeGenerator

from generation.recurBackGenerator import RecurBackMazeGenerator



# this checks if Visualizer has been imported properly.

# if not, likely missing some packages, e.g., matplotlib.
```

```
# in that case, regardless of visualisation flag, we should set the canVisualise flag to False which will
not call the visuslisation part.
canVisualise = True
try:
 from maze.maze_viz import Visualizer
except:
 Visualizer = None
 canVisualise = False




def usage():
 """
 Print help/usage message.
 """


 # On Teaching servers, use 'python3'
 # On Windows, you may need to use 'python' instead of 'python3' to get this to work
 print('python3 mazeTester.py', '<configuration file>')
 sys.exit(1)



#
# Main.
#
if __name__ == '__main__':
 # Fetch the command line arguments
```

```
args = sys.argv


if len(args) != 2:
 print('Incorrect number of arguments.')
 usage()



# open configuration file
fileName: str = args[1]
with open(fileName,"r") as configFile:
 # use json parser
 configDict = json.load(configFile)


 # assign to variables storing various parameters
 dsApproach: str = configDict['dataStructure']
 rowNum: int = configDict['rowNum']
 colNum: int = configDict['colNum']
 entrances: List[List[int]] = configDict['entrances']
 exits: List[List[int]] = configDict['exits']
 genApproach: str = configDict['generator']
 bVisualise: bool = configDict['visualise']


 #
 # Initialise maze object (which also selects which data structure implementation is used).
 #
 maze: Maze = None
 if dsApproach == 'array':
```

```
 maze = ArrayMaze(rowNum, colNum)
elif dsApproach == 'edge-list':
 maze = GraphMaze(rowNum, colNum, dsApproach)
elif dsApproach == 'inc-mat':
 maze = GraphMaze(rowNum, colNum, dsApproach)
else:
 print('Unknown data structure approach specified.')
 usage()

# add the entraces and exits
for [r,c] in entrances:
 maze.addEntrance(Coordinates(r, c))
for [r,c] in exits:
 maze.addExit(Coordinates(r, c))



#
# Generate maze
#
generator: MazeGenerator = None
if genApproach == 'recur':
 generator = RecurBackMazeGenerator()
else:
 print('Unknown generator approach specified.')
 usage()
```

```
  # timer for generation
  startGenTime : float = time.perf_counter()

  generator.generateMaze(maze)

  # stop timer
  endGenTime: float = time.perf_counter()

  print(f'Generation took {endGenTime - startGenTime:0.4f} seconds')

  # add/generate the entrances and exits
  generator.addEntrances(maze)
  generator.addExits(maze)

  #
  # Display maze.
  #
  if bVisualise and canVisualise:
   cellSize = 1
   visualiser = Visualizer(maze, cellSize)
   visualiser.show_maze()
```

```
// File: ./mazeTester_dataGen.py
# -------------------------------------------------------------------
# Maze tester with data generation.
```

```
# This is the entry point to run the program.
# Refer to usage() for exact format of input expected to the program.
#
# __author__ = 'Jeffrey Chan'
# __copyright__ = 'Copyright 2024, RMIT University'
# ----------------------------------------------------------------



import sys
import time
import random
import pandas

from typing import List

from maze.util import Coordinates
from maze.maze import Maze
from maze.arrayMaze import ArrayMaze
from maze.graphMaze import GraphMaze

from generation.mazeGenerator import MazeGenerator
from generation.recurBackGenerator import RecurBackMazeGenerator



# this checks if Visualizer has been imported properly.
# if not, likely missing some packages, e.g., matplotlib.
# in that case, regardless of visualisation flag, we should set the canVisualise flag to False which will
```

```
not call the visuslisation part.

# Flag set to false for quick testing

canVisualise = True

try:

 from maze.maze_viz import Visualizer

except:

 Visualizer = None

 canVisualise = False




def usage():

 """

 Print help/usage message.

 """


 # On Teaching servers, use 'python3'

 # On Windows, you may need to use 'python' instead of 'python3' to get this to work

 print('python3 mazeTester_dataGen.py')

 sys.exit(1)


def checkRangeInt(value, minVal: int, maxVal: int) -> bool:

 """

 Check if value is within the range [minVal, maxVal].

 Return the value if it is within the range.

 Print type errors if non-number is given.

 """
```

```
 if isinstance(value, int):
  if minVal <= value <= maxVal:
   return True
  else:
   print(f"Value {value} is not within the range [{minVal}, {maxVal}].")
 else:
  print(f"Type error: {value} is not an integer.")
  return False


def generate_maze_instance():
    # Randomly generate n and m between 4 and 125
    n = random.randint(4, 125)
    m = random.randint(4, 125)


    # Determine the number of entrances and exits between 1 and 4
 # To add more randomness, hav
    entrances_count = random.randint(1, 4)
    exits_count = random.randint(max(1, entrances_count-2), min(4, entrances_count+2))


    def generate_boundary_coordinate(max_row, max_col):
        # Generate a random boundary coordinate
        if random.choice([True, False]):
            return [random.choice([-1, max_row]), random.randint(0, max_col)]
        else:
            return [random.randint(0, max_row), random.choice([-1, max_col])]


    # Generate entrances and exits
```

```
    entrances = [generate_boundary_coordinate(n-1, m-1) for _ in range(entrances_count)]
    exits = [generate_boundary_coordinate(n-1, m-1) for _ in range(exits_count)]

    # Set visualise to false to prevent block
    visualise = False

    # Create the JSON object
    maze_json = {
        "rowNum": n,
        "colNum": m,
        "entrances": entrances,
        "exits": exits,
        "generator": "recur",
        "visualise": visualise
    }

    return maze_json

def generate_mazes(num_mazes=10):
    return [generate_maze_instance() for _ in range(num_mazes)]

#
# Main.
#
if __name__ == '__main__':
  # Fetch the command line arguments
  args = sys.argv
```

```
# if len(args) != 2:
#  print('Incorrect number of arguments.')
#  usage()
#Files to store run output of default structure, adj list and adj matrix
runFileName="record.csv"
recordDF=pandas.DataFrame(columns=["DataStruct","row","col","runTime"])
structOfRuns=["array","inc-list","inc-mat"]
# open configuration file
# fileName: str = args[1]
# with open(fileName,"r") as configFile:
#  # use json parser
#  configDict = json.load(configFile)

#  # assign to variables storing various parameters
#  dsApproach: str = configDict['dataStructure']
#  rowNum: int = configDict['rowNum']
#  colNum: int = configDict['colNum']
#  entrances: List[List[int]] = configDict['entrances']
#  exits: List[List[int]] = configDict['exits']
#  genApproach: str = configDict['generator']
#  bVisualise: bool = configDict['visualise']

 #
 # Initialise maze object (which also selects which data structure implementation is used).
 #
```

```
fileRun= 10

configs=generate_mazes(fileRun) #Empty dict to populate structure

for dsApproach in structOfRuns:

 for i in range(fileRun):

  maze: Maze = None

  rowNum=configs[i]['rowNum']

  colNum=configs[i]['colNum']

  entrances: List[List[int]]=configs[i]['entrances']

  exits: List[List[int]]=configs[i]['exits']

  genApproach=configs[i]["generator"]

  bVisualise: bool = configs[i]['visualise']

  # recordDF.loc[i]=[dsApproach,rowNum,colNum]

  if dsApproach == 'array':

   maze = ArrayMaze(rowNum, colNum)

  elif dsApproach == 'edge-list':

   maze = GraphMaze(rowNum, colNum, dsApproach)

  elif dsApproach == 'inc-mat':

   maze = GraphMaze(rowNum, colNum, dsApproach)

  else:

   print('Unknown data structure approach specified.')

   usage()


  # add the entraces and exits

  for [r,c] in entrances:

   maze.addEntrance(Coordinates(r, c))

  for [r,c] in exits:

   maze.addExit(Coordinates(r, c))
```

```
#
# Generate maze
#
generator: MazeGenerator = None
if genApproach == 'recur':
 generator = RecurBackMazeGenerator()
else:
 print('Unknown generator approach specified.')
 usage()


# timer for generation
startGenTime : float = time.perf_counter()

generator.generateMaze(maze)

# stop timer
endGenTime: float = time.perf_counter()
timeRan=endGenTime - startGenTime
print(f'Generation took {timeRan:0.4f} seconds')

# add/generate the entrances and exits
generator.addEntrances(maze)
generator.addExits(maze)
```

```
    #
    # Display maze.
    #
    if bVisualise and canVisualise:
     cellSize = 1
     visualiser = Visualizer(maze, cellSize)
     visualiser.show_maze()

  #Print the first part of the df
  print(recordDF.head())



// File: ./generation\mazeGenerator.py
# -------------------------------------------------------------------
# DON'T CHANGE THIS FILE.
# Base class for maze generator.
#
# __author__ = 'Jeffrey Chan'
# __copyright__ = 'Copyright 2024, RMIT University'
# -------------------------------------------------------------------



from maze.maze import Maze
from maze.util import Coordinates

class MazeGenerator:
  """
```

```
Base class for a maze generator.
"""


def generateMaze(self, maze:Maze):
 """
    Generates a maze.  Will update the passed maze.


 @param maze Maze which we update on to generate a maze.
 """
 pass




def addEntrances(self, maze:Maze):
 """
 Add entrance(s) to the maze.


 @param maze Maze which we update on to generate a maze.
 """


 # when adding the entrances, we need to remove the relevant boundary wall
 for ent in maze.getEntrances():
  # need to figure out which direction to remove wall
  # entrance is at bottom, need to remove wall in "up" direction
  if ent.getRow() == -1:
   maze.removeWall(ent, Coordinates(0, ent.getCol()))
  # entrance is at top, need to remove wall in "down" direction
```

```
  elif ent.getRow() == maze.rowNum():
   maze.removeWall(ent, Coordinates(maze.rowNum()-1, ent.getCol()))
  # entrace is to the left, need to remove wall in "right" direction
  elif ent.getCol() == -1:
   maze.removeWall(ent, Coordinates(ent.getRow(), 0))
  # entrance is to the right, need to remove wall in "left" direction
  elif ent.getCol() == maze.colNum():
   maze.removeWall(ent, Coordinates(ent.getRow(), maze.colNum()-1))




def addExits(self, maze:Maze):
 """
 Add exit(s) to the maze.

 @param maze Maze which we update on to generate a maze.
 """

 # when adding the exits, we need to remove the relevant boundary wall
 for ext in maze.getExits():
  # need to figure out which direction to remove wall
  # exit is at bottom, need to remove wall in "up" direction
  if ext.getRow() == -1:
   maze.removeWall(ext, Coordinates(0, ext.getCol()))
  # exit is at top, need to remove wall in "down" direction
  elif ext.getRow() == maze.rowNum():
```

```
    maze.removeWall(ext, Coordinates(maze.rowNum()-1, ext.getCol()))
   # exit is to the left, need to remove wall in "right" direction
   elif ext.getCol() == -1:
    maze.removeWall(ext, Coordinates(ext.getRow(), 0))
   # exit is to the right, need to remove wall in "left" direction
   elif ext.getCol() == maze.colNum():
    maze.removeWall(ext, Coordinates(ext.getRow(), maze.colNum()-1))
```

// File: ./generation\recurBackGenerator.py

```
# -------------------------------------------------------------------
# DON'T CHANGE THIS FILE.
# Recursive backtracking maze generator.
#
# __author__ = 'Jeffrey Chan'
# __copyright__ = 'Copyright 2024, RMIT University'
# -------------------------------------------------------------------

from random import randint, choice
from collections import deque

from maze.maze import Maze
from maze.util import Coordinates
from generation.mazeGenerator import MazeGenerator
```

```
class RecurBackMazeGenerator(MazeGenerator):
 """
 Recursive backtracking maze generator.
 Overrides genrateMaze of parent class.
 """

 def generateMaze(self,maze: Maze):

  # make sure we start the maze with all walls there
  maze.initCells(True)

  # select starting cell
  startCoord : Coordinates = Coordinates(randint(0, maze.rowNum()-1), randint(0,
maze.colNum()-1))

  # run recursive backtracking/DFS from starting cell
  stack : deque = deque()
  stack.append(startCoord)
  currCell : Coordinates = startCoord
  visited : set[Coordinates] = set([startCoord])

  totalCells = maze.rowNum() * maze.colNum()

  while len(visited) < totalCells:
   # find all neighbours of current cell
```

```
  neighbours : list[Coordinates] = maze.neighbours(currCell)

  # filter to ones that haven't been visited and within boundary
  nonVisitedNeighs : list[Coordinates] = [neigh for neigh in neighbours if neigh not in visited and neigh.getRow() >= 0 and neigh.getRow() < maze.rowNum() and neigh.getCol() >= 0 and neigh.getCol() < maze.colNum()]

  # see if any unvisited neighbours
  if len(nonVisitedNeighs) > 0:
   # randomly select one of them
   neigh = choice(nonVisitedNeighs)

   # we move there and knock down wall
   maze.removeWall(currCell, neigh)

   # add to stack
   stack.append(neigh)

   # updated visited
   visited.add(neigh)

   # update currCell
   currCell = neigh
  else:
   # backtrack
   currCell = stack.pop()
```

```
// File: ./maze\arrayMaze.py

# ----------------------------------------------------------------------

# DON'T CHANGE THIS FILE.

# Array-based maze implementation.

# Provided as an example, please use this also as an example of what you

# need to do for the graph implementations.

#

# __author__ = 'Jeffrey Chan'

# __copyright__ = 'Copyright 2024, RMIT University'

# ----------------------------------------------------------------------




from typing import List

from maze.maze import Maze

from maze.util import Coordinates




class ArrayMaze(Maze):

    """

    Array implementation of a 2D, square cell maze.

    Provided as example of an implementation.

    """



    def __init__(self, rowNum:int, colNum:int):
```

```
        super().__init__(rowNum, colNum)
        # this grid storages both the cells, the walls and all the cells strounding the outer boundary of the maze
        # Hence we need 2*rowNum/colNum + 2
        self.m_grid = [[True for c in range(2*colNum+2)] for r in range(2*rowNum+2)]



    def initCells(self, addWallFlag:bool = False):

        super().initCells(addWallFlag)
        if addWallFlag:
            super().allWalls()
        # otherwise we don't need to do anything, as the cells are initiated already.



    def addWall(self, cell1:Coordinates, cell2:Coordinates)->bool:

        # checks if coordinates are valid
        assert(self.checkCoordinates(cell1) and self.checkCoordinates(cell2))

        # check if cells are adjacent
        if cell1.isAdjacent(cell2):
            # difference between the rows and columns for the two cells we adding a wall between
            diff:tuple[int,int] = (cell2.getRow() - cell1.getRow(), cell2.getCol() - cell1.getCol())
```

```
        # check if wall exist
        if self.m_grid[cell1.getRow()*2 + diff[0] + 2][cell1.getCol()*2 + diff[1] + 2]:
            return False
        else:
            # wall doesn't exist, hence we can add a wall there
            self.m_grid[cell1.getRow()*2 + diff[0] + 2][cell1.getCol()*2 + diff[1] + 2] = True

    return True



def removeWall(self, cell1:Coordinates, cell2:Coordinates)->bool:

    # checks if coordinates are valid
    assert(self.checkCoordinates(cell1) and self.checkCoordinates(cell2))

    # check if cells are adjacent
    if cell1.isAdjacent(cell2):
        # difference between the rows and columns for the two cells we are moving a wall between
        diff:tuple[int,int] = (cell2.getRow() - cell1.getRow(), cell2.getCol() - cell1.getCol())
        if not self.m_grid[cell1.getRow()*2 + diff[0] + 2][cell1.getCol()*2 + diff[1] + 2]:
            return False
        else:
            # wall does exist, hence we can remove a wall there
            self.m_grid[cell1.getRow()*2 + diff[0] + 2][cell1.getCol()*2 + diff[1] + 2] = False

    return True
```

```
    def hasWall(self, cell1:Coordinates, cell2:Coordinates)->bool:

        # checks if coordinates are valid
        assert(self.checkCoordinates(cell1) and self.checkCoordinates(cell2))

        # check if cells are adjacent
        if cell1.isAdjacent(cell2):
             # difference between the rows and columns for the two cells we are checking if a wall exists
between them
            diff:tuple[int,int] = (cell2.getRow() - cell1.getRow(), cell2.getCol() - cell1.getCol())
            return self.m_grid[cell1.getRow()*2 + diff[0] + 2][cell1.getCol()*2 + diff[1] + 2]
        else:
            # if not adjacent, then return False.
            return False



    def neighbours(self, cell:Coordinates)->List[Coordinates]:

        # checks if coordinates are valid
        assert(self.checkCoordinates(cell))

        # neighbour one cell below
        neighbours : List[Coordinates] = []
```

```
        if cell.getRow()-1 >= -1:
            neighbours.append(Coordinates(cell.getRow()-1, cell.getCol()))
        # neighbour one cell above
        if cell.getRow()+1 <= self.rowNum():
            neighbours.append(Coordinates(cell.getRow()+1, cell.getCol()))
        # neighbour one cell to the left
        if cell.getCol()-1 >= -1:
            neighbours.append(Coordinates(cell.getRow(), cell.getCol()-1))
        # neighbour one cell to the right
        if cell.getCol()+1 <= self.colNum():
            neighbours.append(Coordinates(cell.getRow(), cell.getCol()+1))

        return neighbours
```

// File: ./maze\edgeListGraph.py

```
# ------------------------------------------------------------------------
# Please COMPLETE the IMPLEMENTATION of this class.
# Adjacent list implementation.
#
# __author__ = 'Jeffrey Chan', <YOU>
# __copyright__ = 'Copyright 2024, RMIT University'
# ------------------------------------------------------------------------
```

```
from typing import List, Set, Dict, Tuple


from maze.util import Coordinates

from maze.graph import Graph




class EdgeListGraph(Graph):

    """

    Represents an undirected graph using a dictionary for edge storage.

    """


    def __init__(self, rowNum:int, colNum:int):

        #Empty adjacent list initializtion

        self.vertexSet: Set[Coordinates] = set()

        self.edgeDict: list[Coordinates, Dict[Coordinates, bool]] = {} #Actually acts like adjacency matrix
, decrease run time

        self.nRows = 0

        self.nCols = 0



    def addVertex(self, label: Coordinates):

        """Add a single vertex if it doesn't already exist."""

        if label not in self.vertexSet:

            self.vertexSet.add(label)

            self.edgeDict[label] = {} # Newly added node/vertex have no edges
```

```
def addVertices(self, vertLabels: List[Coordinates]):
    """Add multiple vertices, ensuring no duplicates."""
    for v in vertLabels:
        self.addVertex(v)

def addEdge(self, vert1: Coordinates, vert2: Coordinates, addWall: bool = False) -> bool:
    """Adds an edge to the graph. An edge is defined by two vertex labels."""
    if vert1 in self.vertexSet and vert2 in self.vertexSet and vert1.isAdjacent(vert2):
        if vert2 not in self.edgeDict[vert1]:
            self.edgeDict[vert1][vert2] = addWall
            self.edgeDict[vert2][vert1] = addWall  # Undirected graph
            return True
    return False


def updateWall(self, vert1: Coordinates, vert2: Coordinates, wallStatus: bool) -> bool:
    """Updates the wall status between two adjacent vertices."""
    if vert2 in self.edgeDict[vert1]:
        self.edgeDict[vert1][vert2] = wallStatus
        self.edgeDict[vert2][vert1] = wallStatus  # Undirected graph
        return True
    return False


def removeEdge(self, vert1: Coordinates, vert2: Coordinates) -> bool:
    """Removes an edge between two vertices."""
    if vert2 in self.edgeDict[vert1]:
        del self.edgeDict[vert1][vert2]
        del self.edgeDict[vert2][vert1]  # Undirected graph
        return True
```

```
        return False

    def hasEdge(self, vert1: Coordinates, vert2: Coordinates) -> bool:
        """Checks if an edge exists between two vertices."""
        return vert2 in self.edgeDict[vert1]

    def getWallStatus(self, vert1: Coordinates, vert2: Coordinates) -> bool:
        """Gets the wall status between two vertices."""
        return self.edgeDict.get(vert1, {}).get(vert2, False)

    def getEdgesList(self):
        for vert in self.edgeDict:
            x=vert.getRow
            y=vert.getCol
            print((x,y),self.edgeDict[vert])
        pass

    def neighbours(self, label: Coordinates) -> List[Coordinates]:
        """retrieves all the neighobors of a vertex"""
        return list(self.edgeDict[label].keys())
```

```
// File: ./maze\graph.py
# -------------------------------------------------
# DON'T CHANGE THIS FILE.
# Base class for graph implementations.
```

```python
#
# __author__ = 'Jeffrey Chan'
# __copyright__ = 'Copyright 2024, RMIT University'
# ------------------------------------------------

from typing import List

from maze.util import Coordinates


class Graph:
    """
    Base class for a graph.  Defines the interface.
    """

    def addVertex(self, label:Coordinates):
        """
        Adds a vertex to the graph.

        @param label Label of the added vertex (which is a Coordinate),
        """
        pass
```

```
    def addVertices(self, vertLabels:List[Coordinates]):
        """
        Adds a list of vertices to the graph.

        @param vertLabels List of labels of the added vertices,
        """
        pass



    def addEdge(self, vert1:Coordinates, vert2:Coordinates, addWall:bool = False)->bool:
        """
            Adds an edge to the graph.  An edge is defined by the two vertex labels, which are
Coordinates.

        @param vert1: Label of source vertex of added edge.
        @param vert2: Label of target vertex of added edge.
        @param addWall: Whether to add wall as well.  Default is False.

        @returns True if edge is successfully added, otherwise False.
        """
        pass



    def updateWall(self, vert1:Coordinates, vert2:Coordinates, wallStatus:bool)->bool:
        """
```

```
    Sets wall between vert1 and vert2.  Vert1 and vert2 should be adjacent.

    @param vert1: Label of source vertex.
    @param vert2: Label of target vertex.
    @param wallStatus: Whether to set wall or not.  True to set/add wall.

    @returns True if edge weight/bool is successfully set, otherwise False.
    """
    pass


def removeEdge(self, vert1:Coordinates, vert2:Coordinates)->bool:
    """
    Removes edge.  Edge must exist for the operation to succeed.

    @param vert1: Label of source vertex of removed edge.
    @param vert2: Label of target vertex of removed edge.

    @returns True if edge is successfully removed, otherwise False.
    """
    pass


def hasVertex(self, label:Coordinates)->bool:
    """
```

```
    Checks if label is a vertex in the graph.

    @param label: Label/Coordinate to check.

    @returns True if vertex exists in graph, otherwise False.
    """
    pass



def hasEdge(self, vert1:Coordinates, vert2:Coordinates)->bool:
    """
    Checks if label is a vertex in the graph.

    @param vert1: Label of source vertex to check.
    @param vert2: Label of target vertex to check.

    @returns True if edge exists in graph, otherwise False.
    """
    pass



def getWallStatus(self, vert1:Coordinates, vert2:Coordinates)->bool:
    """
    Gets the status of wall between vert1 and vert2.
```

```
        @param vert1: Label of source vertex.
        @param vert2: Label of target vertex

        @returns True if wall status was successfully retrieved, otherwise False.
        """
        pass



    def neighbours(self, label:Coordinates)->List[Coordinates]:
        """
        Retrieves all the neighbours of vertex/label.

        @param label: Label of vertex to obtain neighbours.

        @returns List of neighbouring vertices.  Returns empty list if no neighbours.
        """
        pass
```

// File: ./maze\graphMaze.py

```
# -------------------------------------------------------------------------

# MODIFY IF NEED TO.

# Graph implementation of a maze.

#

# __author__ = 'Jeffrey Chan'

# __copyright__ = 'Copyright 2024, RMIT University'

# -------------------------------------------------------------------------



from typing import List


from maze.maze import Maze

from maze.util import Coordinates

from maze.graph import Graph

from maze.edgeListGraph import EdgeListGraph

from maze.incidenceMatGraph import IncMatGraph





class GraphMaze(Maze):
    """
    Graph implementation of a 2D, square cell maze.
    """


    def __init__(self, rowNum:int, colNum:int, graphType:str):
        """
```

```
        Constructor.

        Has extra argument of the type of graph we will use as the underlying graph implementation.


        @param graphType: Name of underlying graph implementation.  [adjlist, adjmat].
        """


        super().__init__(rowNum, colNum)
        self.m_graph : Graph = None
        if graphType == 'edge-list':
            self.m_graph = EdgeListGraph(rowNum=rowNum,colNum=colNum)
        elif graphType == 'inc-mat':
            self.m_graph = IncMatGraph()




    def initCells(self, addWallFlag:bool = False):


        super().initCells()


        # add the vertices and edges to the graph
                self.m_graph.addVertices([Coordinates(r,c) for r in range(self.m_rowNum) for c in
range(self.m_colNum)])
        # add boundary vertices
        self.m_graph.addVertices([Coordinates(-1,c) for c in range(self.m_colNum)])
        self.m_graph.addVertices([Coordinates(r,-1) for r in range(self.m_rowNum)])
        self.m_graph.addVertices([Coordinates(self.m_rowNum,c) for c in range(self.m_colNum)])
        self.m_graph.addVertices([Coordinates(r,self.m_colNum) for r in range(self.m_rowNum)])
```

```
    # add adjacenies/edges to the graph
    # Scan across rows first
    for row in range(0, self.m_rowNum):
        for col in range(-1, self.m_colNum):
            self.m_graph.addEdge(Coordinates(row,col), Coordinates(row,col+1), addWallFlag)

    # scan columns now
    for col in range(0, self.m_colNum):
        for row in range(-1, self.m_rowNum):
            self.m_graph.addEdge(Coordinates(row,col), Coordinates(row+1,col), addWallFlag)



def addWall(self, cell1:Coordinates, cell2:Coordinates)->bool:

    # checks if coordinates are valid
    assert(self.checkCoordinates(cell1) and self.checkCoordinates(cell2))

    # only can add wall if adjacent
    if self.m_graph.hasEdge(cell1, cell2):
        self.m_graph.updateWall(cell1, cell2, True)
        return True

    # in all other cases, we return False
    return False
```

```
def removeWall(self, cell1:Coordinates, cell2:Coordinates)->bool:

    # checks if coordinates are valid
    assert(self.checkCoordinates(cell1) and self.checkCoordinates(cell2))

    # only can remove wall if adjacent
    if self.m_graph.hasEdge(cell1, cell2):
        self.m_graph.updateWall(cell1, cell2, False)
        return True

    # in all other cases, we return False
    return False



def hasWall(self, cell1:Coordinates, cell2:Coordinates)->bool:
    return self.m_graph.getWallStatus(cell1, cell2)



def neighbours(self, cell:Coordinates)->List[Coordinates]:
    return self.m_graph.neighbours(cell)
```

```
// File: ./maze\incidenceMatGraph.py

# -------------------------------------------------------------------------

# Please COMPLETE the IMPLEMENTATION of this class.

# Adjacent matrix implementation.

#

# __author__ = 'Jeffrey Chan', <YOU>

# __copyright__ = 'Copyright 2024, RMIT University'

# -------------------------------------------------------------------------




from typing import List, Dict, Tuple, Set

from maze.util import Coordinates

from maze.graph import Graph


class IncMatGraph(Graph):

    """

    Represents an undirected graph using an optimized adjacency matrix.

    """



    def __init__(self):

        self.vertexList: List[Coordinates] = []  # Store vertices

        self.vertexIndex: Dict[Coordinates, int] = {}  # Map each vertex to an index

        self.edgeMatrix: List[List[bool]] = []  # 2D list for adjacency matrix

        self.nVertices = 0  # Track the number of vertices
```

```python
def addVertex(self, label: Coordinates):
    """Add a single vertex if it doesn't already exist."""
    if label not in self.vertexIndex:
        self.vertexIndex[label] = self.nVertices
        self.vertexList.append(label)
        self.nVertices += 1

        # Expand the adjacency matrix for the new vertex
        for row in self.edgeMatrix:
            row.append(False)  # Add False to each existing row for the new vertex
        self.edgeMatrix.append([False] * self.nVertices)  # Add a new row for the new vertex

def addVertices(self, vertLabels: List[Coordinates]):
    """Add multiple vertices, ensuring no duplicates."""
    for v in vertLabels:
        self.addVertex(v)

def addEdge(self, vert1: Coordinates, vert2: Coordinates, addWall: bool = False) -> bool:
    """Adds an edge to the graph. An edge is defined by two vertex labels."""
    if vert1 in self.vertexIndex and vert2 in self.vertexIndex and vert1.isAdjacent(vert2):
        i, j = self.vertexIndex[vert1], self.vertexIndex[vert2]
        if not self.edgeMatrix[i][j]:  # Add edge only if it doesn't exist
            self.edgeMatrix[i][j] = addWall
            self.edgeMatrix[j][i] = addWall  # Symmetric for undirected graph
            return True
    return False
```

```
def updateWall(self, vert1: Coordinates, vert2: Coordinates, wallStatus: bool) -> bool:
    """Updates the wall status between two adjacent vertices."""
    if vert1 in self.vertexIndex and vert2 in self.vertexIndex:
        i, j = self.vertexIndex[vert1], self.vertexIndex[vert2]
        self.edgeMatrix[i][j] = wallStatus
        self.edgeMatrix[j][i] = wallStatus  # Symmetric for undirected graph
        return True
    return False


def removeEdge(self, vert1: Coordinates, vert2: Coordinates) -> bool:
    """Removes an edge between two vertices."""
    if vert1 in self.vertexIndex and vert2 in self.vertexIndex:
        i, j = self.vertexIndex[vert1], self.vertexIndex[vert2]
        if self.edgeMatrix[i][j]:
            self.edgeMatrix[i][j] = False
            self.edgeMatrix[j][i] = False  # Symmetric for undirected graph
            return True
    return False


def hasVertex(self, label: Coordinates) -> bool:
    """Checks if a vertex exists in the graph."""
    return label in self.vertexIndex


def hasEdge(self, vert1: Coordinates, vert2: Coordinates) -> bool:
    """Checks if an edge exists between two vertices."""
    if vert1 in self.vertexIndex and vert2 in self.vertexIndex:
        i, j = self.vertexIndex[vert1], self.vertexIndex[vert2]
```

```
            return self.edgeMatrix[i][j]
        return False

    def getWallStatus(self, vert1: Coordinates, vert2: Coordinates) -> bool:
        """Gets the wall status between two vertices."""
        if vert1 in self.vertexIndex and vert2 in self.vertexIndex:
            i, j = self.vertexIndex[vert1], self.vertexIndex[vert2]
            return self.edgeMatrix[i][j]
        return False

    def neighbours(self, label: Coordinates) -> List[Coordinates]:
        """Retrieves all the neighbours of a vertex."""
        if label in self.vertexIndex:
            index = self.vertexIndex[label]
            return [self.vertexList[j] for j in range(self.nVertices) if self.edgeMatrix[index][j]]
        return []

// File: ./maze\maze.py
# -------------------------------------------------
# DON'T CHANGE THIS FILE.
# Base class for maze implementations.
#
# __author__ = 'Jeffrey Chan'
# __copyright__ = 'Copyright 2024, RMIT University'
# -------------------------------------------------

from typing import List
```

```
from maze.util import Coordinates



class Maze:
    """
    Base (abstract) class for mazes.
    """



    def __init__(self, rowNum:int, colNum:int):
        """
        Constructor.

        @param rowNum: number of rows in the maze.
        @param colNum: number of columns in the maze
        """
        self.m_rowNum = rowNum
        self.m_colNum = colNum

        # entrances and exits
        self.m_entrance = list()
        self.m_exit = list()




    def initCells(self, addWallFlag:bool = False):
```

```
    """
    Initialises the cells in the maze.
    Override to customise behaviour.

    @param addWallFlag: Whether we should also add the walls between cells.  Default is False.
    """
    pass



def addWall(self, cell1:Coordinates, cell2:Coordinates)->bool:
    """
    Adds a wall between cells cell1 and cell2.
    cell1 and cell2 should be adjacent.
    Override to customise behaviour.

    @param cell1: Coordinates of cell1.
    @param cell2: Coordinates of cell2.

    @return True if successfully added a wall, otherwise False in all other cases.
    """
    pass



def removeWall(self, cell1:Coordinates, cell2:Coordinates)->bool:
    """
```

```
    Removes a wall between cells cell1 and cell2.
    cell1 and cell2 should be adjacent.
    Override to customise behaviour.

    @param cell1: Coordinates of cell1.
    @param cell2: Coordinates of cell2.

    @return True if successfully removed a wall, otherwise False in all other cases.
    """
    pass



def allWalls(self):
    """
    Add walls between all cells in the maze.
    """

    # add walls to the left and bottom of a 2d traversal of cells
    for r in range(-1,self.m_rowNum):
        for c in range(-1,self.m_colNum):
            self.addWall(Coordinates(r,c), Coordinates(r+1,c))
            self.addWall(Coordinates(r,c), Coordinates(r,c+1))

    # add the wall along the right maze boundary, and top maze boundary
    for r in range(0,self.m_rowNum):
        self.addWall(Coordinates(r,self.m_colNum-1), Coordinates(r,self.m_colNum))
```

```
        for c in range(0,self.m_colNum):
            self.addWall(Coordinates(self.m_rowNum-1, c), Coordinates(self.m_rowNum, c))




    def addEntrance(self, cell: Coordinates)->bool:
        """
        Adds an entrance to the maze.  A maze can have more than one entrance, so this method can be called more than once.

        @return True if successfully added an entrance, otherwise False.
        """

        # check if cell of entrance is valid
        assert(self.checkCoordinates(cell))

        # check if cell of the entrance is on the boundary of the maze, as an entrance should only be added along the boundary
        if (cell.getRow() == -1 and cell.getCol() >= 0 and cell.getCol() < self.m_colNum) \
                        or (cell.getRow() == self.m_rowNum and cell.getCol() >= 0 and cell.getCol() < self.m_colNum) \
            or (cell.getCol() == -1 and cell.getRow() >= 0 and cell.getRow() < self.m_rowNum) \
                        or (cell.getCol() == self.m_colNum and cell.getRow() >= 0 and cell.getRow() < self.m_rowNum):

            self.m_entrance.append(cell)
```

```
            return True
        else:
            # not on the boundary
            return False



    def addExit(self, cell: Coordinates)->bool:
        """
         Adds an exit to the maze.  A maze can have more than one exit, so this method can be called more than once.

        @return True if successfully added an exit, otherwise False.
        """

        # check if cell of exit is valid
        assert(self.checkCoordinates(cell))

        # check if cel of exitl is on the boundary of the maze, as an exit should only be added along the boundary
        if (cell.getRow() == -1 and cell.getCol() >= 0 and cell.getCol() < self.m_colNum) \
                    or (cell.getRow() == self.m_rowNum and cell.getCol() >= 0 and cell.getCol() < self.m_colNum) \
            or (cell.getCol() == -1 and cell.getRow() >= 0 and cell.getRow() < self.m_rowNum) \
                    or (cell.getCol() == self.m_colNum and cell.getRow() >= 0 and cell.getRow() < self.m_rowNum):
```

```
        self.m_exit.append(cell)

        return True
    else:
        # not on boundary
        return False


def getEntrances(self)->List[Coordinates]:
    """
    @returns list of entrances that the maze has.
    """
    return self.m_entrance


def getExits(self)->List[Coordinates]:
    """
    @returns list of exits that the maze has.
    """
    return self.m_exit


def hasWall(self, cell1:Coordinates, cell2:Coordinates)->bool:
```

```
        """
        Checks if there is a wall between cell1 and cell2.
        Override if need to customise behaviour

        @returns True, if there is a wall.

        """
        pass



    def rowNum(self)->int:
        """
        @returns The number of rows the maze has.
        """

        return self.m_rowNum



    def colNum(self)->int:
        """
        @return The number of columns the maze has.
        """

        return self.m_colNum
```

```
    def checkCoordinates(self, coord:Coordinates)->bool:
        """
        Checks if the coordinates is a valid one.

        @param coord: Cell/coordinate to check if it is a valid one.

        @returns True if coord/cell is valid, otherwise False.
        """

        return coord.getRow() >= -1 and coord.getRow() <= self.m_rowNum and coord.getCol() >= -1 and coord.getCol() <= self.m_colNum



    def isPerfect(self)->bool:
        """
        Checks if the maze is perfect.
        Please feel free to make your own implementation to evaluate if your generated mazes are perfect.  You will
        not be assessed for this by for your own checking.  Please do not submit your implementation when submitting in Canvas.
        If you do accidentally, we will replace this file with the existing one when testing, but ideally better if you didn't.

        @returns True if the generated maze is perfect, or False if not.
```

```
    """
    pass


// File: ./maze\maze_viz.py
# -------------------------------------------------
# DON'T CHANGE THIS FILE.
# Visualiser, original code from https://github.com/jostbr/pymaze writteb by Jostein Brændshøi
# Subsequentially modified by Jeffrey Chan.
#
# __author__ = 'Jostein Brændshøi, Jeffrey Chan'
# __copyright__ = 'Copyright 2024, RMIT University'
# -------------------------------------------------

# MIT License

# Copyright (c) 2021 Jostein Brændshøi

# Permission is hereby granted, free of charge, to any person obtaining a copy
# of this software and associated documentation files (the "Software"), to deal
# in the Software without restriction, including without limitation the rights
# to use, copy, modify, merge, publish, distribute, sublicense, and/or sell
# copies of the Software, and to permit persons to whom the Software is
# furnished to do so, subject to the following conditions:

# The above copyright notice and this permission notice shall be included in all
# copies or substantial portions of the Software.
```

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE
AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM,
OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE
SOFTWARE.


```
import matplotlib.pyplot as plt

from maze.maze import Maze
from maze.util import Coordinates

import time


class Visualizer(object):
    """Class that handles all aspects of visualization.


    Attributes:
        maze: The maze that will be visualized
        cell_size (int): How large the cells will be in the plots
```

```
    height (int): The height of the maze
    width (int): The width of the maze
    ax: The axes for the plot
"""

def __init__(self, maze :Maze, cellSize):
    self.m_maze = maze
    self.m_cellSize = cellSize
    self.m_height = (maze.rowNum()+2) * cellSize
    self.m_width = (maze.colNum()+2) * cellSize
    self.m_ax = None


def show_maze(self):
    """Displays a plot of the maze without the solution path"""

    # create the plot figure and style the axes
    fig = self.configure_plot()

    # plot the walls on the figure
    self.plot_walls()

    # plot the entrances and exits on the figure
    self.plotEntExit()

    # display the plot to the user
    plt.show()
```

```
    plt.show(block=False)  # Show the plot without blocking the execution
    time.pause(3)  # Keep the plot open for 3 seconds
    plt.close(fig)  # Close the plot after 3 seconds



def plot_walls(self):
    """
    Plots the walls of a maze. This is used when generating the maze image.
    """
    for r in range(0, self.m_maze.rowNum()):
        for c in range(0, self.m_maze.colNum()):
            # if self.maze.initial_grid[i][j].is_entry_exit == "entry":
            #     self.ax.text(j*self.cell_size, i*self.cell_size, "START", fontsize=7, weight="bold")
            # elif self.maze.initial_grid[i][j].is_entry_exit == "exit":
            #     self.ax.text(j*self.cell_size, i*self.cell_size, "END", fontsize=7, weight="bold")

            # top
            if self.m_maze.hasWall(Coordinates(r-1,c), Coordinates(r,c)):
                self.m_ax.plot([(c+1)*self.m_cellSize, (c+1+1)*self.m_cellSize],
                               [(r+1)*self.m_cellSize, (r+1)*self.m_cellSize], color="k")
            # left
            if self.m_maze.hasWall(Coordinates(r,c-1), Coordinates(r,c)):
                self.m_ax.plot([(c+1)*self.m_cellSize, (c+1)*self.m_cellSize],
                               [(r+1)*self.m_cellSize, (r+1+1)*self.m_cellSize], color="k")
```

```
        # do bottom boundary
        for c in range(0, self.m_maze.colNum()):
            # top
                                    if     self.m_maze.hasWall(Coordinates(self.m_maze.rowNum()-1,c)),
Coordinates(self.m_maze.rowNum(),c)):
                self.m_ax.plot([(c+1)*self.m_cellSize, (c+1+1)*self.m_cellSize],
                                        [(self.m_maze.rowNum()+1)*self.m_cellSize,
(self.m_maze.rowNum()+1)*self.m_cellSize], color="k")

        # do right boundary
        for r in range(0, self.m_maze.rowNum()):
            # left
                                    if     self.m_maze.hasWall(Coordinates(r,self.m_maze.colNum()-1),
Coordinates(r,self.m_maze.colNum())):
                                    self.m_ax.plot([(self.m_maze.colNum()+1)*self.m_cellSize,
(self.m_maze.colNum()+1)*self.m_cellSize],
                    [(r+1)*self.m_cellSize, (r+1+1)*self.m_cellSize], color="k")



    def plotEntExit(self):
        """
        Plots the entrances and exits in the displayed maze.
        """

        for ent in self.m_maze.getEntrances():
            # check direction of arrow
```

```
            # upwards arrow
            if ent.getRow() == -1:
                self.m_ax.arrow((ent.getCol()+1.5)*self.m_cellSize, (ent.getRow()+1)*self.m_cellSize, 0,
self.m_cellSize*0.6, head_width=0.1)
            # downwards arrow
            elif ent.getRow() == self.m_maze.rowNum():
                self.m_ax.arrow((ent.getCol()+1.5)*self.m_cellSize, (ent.getRow()+2)*self.m_cellSize, 0,
-self.m_cellSize*0.6, head_width=0.1)
            # rightward arrow
            elif ent.getCol() == -1:
                self.m_ax.arrow((ent.getCol()+1)*self.m_cellSize, (ent.getRow()+1.5)*self.m_cellSize,
self.m_cellSize*0.6, 0, head_width=0.1)
            # leftward arrow
            elif ent.getCol() == self.m_maze.colNum():
                self.m_ax.arrow((ent.getCol()+2)*self.m_cellSize, (ent.getRow()+1.5)*self.m_cellSize,
-self.m_cellSize*0.6, 0, head_width=0.1)

        for ext in self.m_maze.getExits():
            # downwards arrow
            if ext.getRow() == -1:
                self.m_ax.arrow((ext.getCol()+1.5)*self.m_cellSize, (ext.getRow()+1.8)*self.m_cellSize, 0,
-self.m_cellSize*0.6, head_width=0.1)
            # upwards arrow
            elif ext.getRow() == self.m_maze.rowNum():
                self.m_ax.arrow((ext.getCol()+1.5)*self.m_cellSize, (ext.getRow()+1.2)*self.m_cellSize, 0,
self.m_cellSize*0.6, head_width=0.1)
            # leftward arrow
```

```
            elif ext.getCol() == -1:
                        self.m_ax.arrow((ext.getCol())*self.m_cellSize,  (ext.getRow()+1.5)*self.m_cellSize,
-self.m_cellSize*0.6, 0, head_width=0.1)
            # leftward arrow
            elif ext.getCol() == self.m_maze.colNum():
                    self.m_ax.arrow((ext.getCol()+1.2)*self.m_cellSize, (ext.getRow()+1.5)*self.m_cellSize,
self.m_cellSize*0.6, 0, head_width=0.1)




    def configure_plot(self):
        """Sets the initial properties of the maze plot. Also creates the plot and axes"""

        # Create the plot figure
        fig = plt.figure(figsize = (7, 7*self.m_maze.rowNum() / self.m_maze.colNum()))

        # Create the axes
        self.m_ax = plt.axes()

        # Set an equal aspect ratio
        self.m_ax.set_aspect("equal")

        # Remove the axes from the figure
        self.m_ax.axes.get_xaxis().set_visible(False)
        self.m_ax.axes.get_yaxis().set_visible(False)

        # title_box = self.m_ax.text(0, self.m_maze.rowNum() + self.m_cellSize + 0.1,
```

```
        #                    r"{}$\times${}".format(self.m_maze.rowNum(), self.m_maze.colNum()),
        #                    bbox={"facecolor": "gray", "alpha": 0.5, "pad": 4}, fontname="serif", fontsize=15)

        return fig




// File: ./maze\util.py
# -------------------------------------------------
# DON'T CHANGE THIS FILE.
# Utility classes and methods.
#
# __author__ = 'Jeffrey Chan' modified by 'Elham Naghizade'
# __copyright__ = 'Copyright 2024, RMIT University'
# -------------------------------------------------




class Coordinates:
    """
    Represent coordinates for maze cells.
    """

    def __init__(self, row:int, col:int):
        """
        Constructor.
```

```
    @param row: Row of coordinates.
    @param col: Column of coordinates.
    """

    self.m_r = row
    self.m_c = col


def getRow(self)->int:
    """
    @returns Row of coordinate.
    """
    return self.m_r


def getCol(self)->int:
    """
    @returns Column of coordinate.
    """
    return self.m_c


def isAdjacent(self, other:"Coordinates")->bool:
```

```
    """
    Determine if two coordinates are adjacent to each other.
    """
    if (abs(self.m_r - other.getRow()) == 1 and self.m_c == other.getCol()) or\
            (self.m_r == other.getRow() and abs(self.m_c - other.getCol()) == 1):
        return True
    else:
        return False



def __eq__(self, other:"Coordinates"):
    """
    Define == operator.

    @param other: Other coordinates that we are comparing with.
    """
    if other != None:
        return self.m_r == other.getRow() and self.m_c == other.getCol()
    else:
        return False



def __hash__(self):
    """
```

Returns has value of Coordinates. Needed for being a key in dictionaries.

```
"""

return hash(str(self.m_r)+'|'+str(self.m_c))
```