```
import sqlite3
from random import randint, shuffle
from copy import deepcopy
from crossword_grids import crosswords

#A class to represent the nodes within the Graph() class.
#The nodes can be thought of as each being a word on the crossword grid.

class WordLine:

    def __init__(self, word_line_range, n):

        #These represent the coordinates of where the word starts and ends.
        self.__start_row, self.__end_row, self.__start_col, self.__end_col = word_line_range

        #An identifier for each node
        self.__number = n

        #Looks at the coordinates to decide whether the word runs accross or down the crossword grid.
        if self.__start_row != self.__end_row:
            self.__direction = "down"
            self.__word_length = (self.__end_row - self.__start_row) + 1
        else:
            self.__direction = "across"
            self.__word_length = (self.__end_col - self.__start_col) + 1

        self.__intersections = []

        #These attributes store the strings of the words and clues (one will be French and one will be English).
        self.__filling_in_word, self.__clue_word = "", ""

        #Stores the result of the last query made for this node during the traversal.
        self.__query_result = []

        #Stores the index for the query_result to select the current word it is using during the traversal.
        self.__query_result_index = 0

        #Indicates whether it has has a query made for it yet
        self.__had_query = False

        #Indicates the gender of the word if it is an adjective
        self.__gender = ""

        #For indicating whether this word line has been filled in correctly on the interactive crossword grid.
        self.__correct = False

    def getNumber(self):
        return self.__number

    def getStartCol(self):
        return self.__start_col

    def getStartRow(self):
        return self.__start_row

    def getEndCol(self):
        return self.__end_col

    def getEndRow(self):
        return self.__end_row

    def getDirection(self):
        return self.__direction

    def getWordLength(self):
        return self.__word_length

    def getIntersections(self):
        return self.__intersections

    def getFillingInWord(self):
        return self.__filling_in_word

    def getClueWord(self):
        return self.__clue_word
```

```
    def getHadQuery(self):
        return self.__had_query

    def getGender(self):
        return self.__gender

    def getCorrect(self):
        return self.__correct

    def setCorrect(self, bool):
        self.__correct = bool

    #A method for the traversal.

    def checkQueryIndex(self):

        #Returns True if every word in the query has been tried in the traversal.
        if self.__query_result_index >= len(self.__query_result) - 1:
            return False
        else:
            return True

    def addIntersection(self, WordLineConnecting, coords):

        #Appends the node which this node intersects with and the coordinates where they intersect.
        self.__intersections.append((WordLineConnecting, coords))

    def addWord(self):

        self.__filling_in_word, self.__clue_word = self.__query_result[self.__query_result_index][0], self.__query

        if len(self.__query_result[self.__query_result_index]) > 3:
            self.__gender = self.__query_result[self.__query_result_index][3]
        else:
            self.__gender = ""


    #A method for the traversal.

    def addQuery(self, query_result):

        self.__had_query = True
        self.__query_result_index = 0
        self.__query_result = query_result

        #suffles so that the words used in the crossword are randomised
        shuffle(self.__query_result)

        self.addWord()


    #A method for the traversal.

    def switchQueryWord(self):

        #If the query_result_index has not exeeded the length of the query_result then incriment it.
        if self.checkQueryIndex():
            self.__query_result_index += 1
            self.addWord()


    #A method for the traversal.

    def resetNode(self):

        #Resets all of the attributes so that a new query_result can be tried.
        self.__filling_in_word, self.__clue_word, self.__gender = "", "", ""
        self.__query_result_index = 0
        self.__query_result = []
        self.__had_query = False

#A class to represent the graph based off of the crossword grid.

class Graph:

    def __init__(self, crossword, filling_in_language, topics, empty=False, debugging=False):
```

```
        #The empty crossword grid.
        self.__crossword = crossword

        #A list containing all of the WordLine() objects.
        self.__nodes = []

        #An identifier for each node.
        self.__n = 1

        #Indicates that the graph is empty.
        self.__empty = empty

        #A boolean which, when True, means information is printed to help with debugging.
        self.__debugging = debugging

        #Can be equal to "French" or "English". It indicates which way around the languages are in the crossword.
        self.__filling_in_language = filling_in_language

        #The language topics.
        self.__topics = topics

        if self.__filling_in_language == "French":
            self.__clue_language = "English"
        else:
            self.__clue_language = "French"

        self.__constructGraph(self.__crossword)

        #The try and except is used to catch recursion errors.
        try:
            self.__traversal(self.__nodes[0], [])
        except Exception as e:
            print(e)

    #A method used by __constructGraph() to find wordlines.

    def __checkForSpaces(self, row, col, width, height, direction):

        checking = True
        word_line = False

        while checking:

            if col != width and direction == "across":
                col+=1
                if self.__crossword[row][col] == " ":
                    word_line = True
                else:
                    col-=1
                    checking = False

            elif row != height and direction == "down":
                row+=1
                if self.__crossword[row][col] == " ":
                    word_line = True
                else:
                    row-=1
                    checking = False

            else:
                checking = False

        #row is the end row of the wordline and col is the end col.
        #word_line is a boolean that indicates whether the word_line is valid (has a lenght of more than 1).

        return row, col, word_line


    #A method which converts the 2D array of the empty crossword grid into a graph.

    def __constructGraph(self, crossword):

        #This chunk of code checks whether for each cell in the crossword grid it is the first cell of a wordline.
        for row in range(len(crossword)):
            for col in range(len(crossword[row])):

                #Checks if the cell is one of the empty spaces where the letter will be written.
```

```
            if crossword[row][col] == " ":

                #Checks if the cell to the left isn't also an empty space.
                if (crossword[row][col-1] != " " and col != 0) or (col == 0):

                    word_line_row, word_line_col, word_line = self.__checkForSpaces(row, col, len(crossword[row

                    #Checks if __checkForSpaces() has indicated that this cell is the first cell of a wordline.
                    if word_line:

                        #Adds a node to the graph which represnets this wordline.
                        self.__nodes.append(WordLine((row, word_line_row, col, word_line_col), self.__n))
                        self.__n += 1

                #Checks if the cell to the above isn't also an empty space.
                if (crossword[row-1][col] != " " and row != 0) or (row == 0):

                    word_line_row, word_line_col, word_line = self.__checkForSpaces(row, col, len(crossword[row

                    if word_line:
                        self.__nodes.append(WordLine((row, word_line_row, col, word_line_col), self.__n))
                        self.__n += 1

    #This chunk finds and logs the intersections between these nodes (the coordinates where the words overlap t
    for node_1 in self.__nodes:

        #Goes through each coordinate of the node_1 wordline.
        for row in range(node_1.getStartRow(), node_1.getEndRow() + 1):
            for col in range(node_1.getStartCol(), node_1.getEndCol() + 1):

                for node_2 in self.__nodes:

                    #Goes through each coordinate of the node_2 wordline.
                    for row_2 in range(node_2.getStartRow(), node_2.getEndRow() + 1):
                        for col_2 in range(node_2.getStartCol(), node_2.getEndCol() + 1):

                            #Checks if there is an intersection between these two nodes (they can't be the same
                            if node_1 != node_2 and row == row_2 and col == col_2 and (node_2, (row, col)) not

                                node_1.addIntersection(node_2, (row, col))
                                node_2.addIntersection(node_1, (row, col))


#A method which returns a tuple of all the filling in words attributed to the nodes in the graph.

def __fillingInWords(self):

    words = []
    for node in self.__nodes:
        words.append(node.getFillingInWord())

    return tuple(words)


#A method which returns a tuple of all the clue words attributed to the nodes in the graph.

def __clueWords(self):

    words = []
    for node in self.__nodes:
        words.append(node.getClueWord())

    return tuple(words)


#Performs a query for the traversal and returns the result.

def __query(self, substring_values, node):

    #Creates a connection to the database.
    with sqlite3.connect('database.db') as connection:

        cur = connection.cursor()

        if len(self.__topics) == 1:
            self.__topics.append("filler")
```

```
            #A string that represents an SQL query.
            query_string = f"""
            SELECT {self.__filling_in_language}Words.{self.__filling_in_language}WordForCrossword, {self.__clue_la
            FrenchWords.WordClass
            FROM {self.__filling_in_language}Words, {self.__clue_language}Words, Translations
            WHERE ({self.__filling_in_language}Words.{self.__filling_in_language}WordID = Translations.{self.__fil
            AND ({self.__clue_language}Words.{self.__clue_language}WordID = Translations.{self.__clue_language}Word
            AND ({self.__filling_in_language}Words.WordLength = {node.getWordLength()})
            AND ({self.__filling_in_language}Words.{self.__filling_in_language}WordForCrossword NOT IN {self.__fill
            AND (FrenchWords.Topic IN {tuple(self.__topics)})
            """

            #Adds all of the substring queries to the query_string.
            for substring_value in substring_values:
                query_string += f"AND (SUBSTR({self.__filling_in_language}WordForCrossword, {substring_value[0]},1)

            #Execute the query string.
            results = cur.execute(query_string).fetchall()

            #If the clues are in english, the english adjectives need the extra detail of what gender to translate
            if self.__filling_in_language == "French":

                results = [list(x) for x in results]

                for result in results:

                    #If the word is an adjective then perfrom a query to fetch its gender.
                    if result[2] == "adjective":
                        query_string_2 = f"""
                        SELECT FrenchGender.Gender
                        FROM FrenchGender, FrenchWords, EnglishWords, Translations
                        WHERE (FrenchWords.FrenchWordID = FrenchGender.FrenchWordID)
                        AND (FrenchWords.FrenchWordID = Translations.FrenchWordID)
                        AND (EnglishWords.EnglishWordID = Translations.EnglishWordID)
                        AND (FrenchWords.FrenchWordForCrossword = '{result[0]}')
                        """
                        result.append(cur.execute(query_string_2).fetchone()[0])

        #(Debugging).
        if self.__debugging: print(f"query result: {results}")

        return results


    #A recursive method which trys to fill in the empty grid with words through depth first traversal of the graph.

    def __traversal(self, current_node, visited):

        #A variable for storing the letters that the word must contain for the SUBSTR part of the query.
        substring_values = self.__substringValues(current_node)

        #A boolean which indicates whether the method will continue traversing in the current iteration.
        traverse = False

        #(Debgugging).
        if self.__debugging: print(f"\ncurrent_node: {current_node.getNumber()}, substring_values: {substring_value

        #Runs if the current_node does not have a query result attributed to it.
        if not current_node.checkQueryIndex() and not current_node.getHadQuery():

            #(Debgugging).
            if self.__debugging: print(f"performing query (node {current_node.getNumber()} does not have a query ye

            #Perform a query and store the results.
            #result is a 2D array in the format ((1st word result, traslation), (2nd word result, traslation)).
            result = self.__query(substring_values, current_node)

            #Checks if no results have been found for the query
            if result == []:

                #If it is the first node being traversed and no query results are found then the traversal must end
                if visited==[]:

                    #(Debugging).
                    if self.__debugging: print("End of traversal")

                #No results have been found so it returns to the previous node visited.
```

```
                else:

                    #(Debugging).
                    if self.__debugging: print(f"going back from node {current_node.getNumber()} to {visited[-1].ge

                    visited_store = visited[-1]
                    visited.pop(-1)
                    self.__traversal(visited_store, visited)

            #At least one query result has been found.
            else:

                #(Debugging).
                if self.__debugging: print("Added query result")

                #Store the query result in the current_node object.
                current_node.addQuery(result)

                #Continuing the depth-first traversal.
                traverse = True

        #Runs if the current_node has a query_result but every word from that query result has been tried.
        elif not current_node.checkQueryIndex() and current_node.getHadQuery():

            if visited==[]:

                if self.__debugging: print("End of traversal")

            #The current_node is reset and the program returns to the previous node visited.
            else:

                current_node.resetNode()

                #(Debugging).
                if self.__debugging:
                    print(f"reset node {current_node.getNumber()}")
                    print(f"going back from node {current_node.getNumber()} to {visited[-1].getNumber()}")

                visited_store = visited[-1]
                visited.pop(-1)
                self.__traversal(visited_store, visited)

        #Else is ran when there is a query result stored in the current_node but there are still some words left to
        else:

            #Try the next word.
            current_node.switchQueryWord()

            #Continuing the depth-first traversal.
            traverse = True

        #Continue the depth-first traversal if traverse is True.
        if traverse:

            #(Debugging).
            if self.__debugging:
                print("traversing")
                print(f"selected: {current_node.getFillingInWord()}")

            #Add the current node to the visited list.
            visited.append(current_node)

            for intersection in current_node.getIntersections():

                #(Debugging).
                if self.__debugging: print(f"intersection: from {current_node.getNumber()} to {intersection[0].getN

                #If the intersecting node has not been visited yet then visit it.
                if intersection[0] not in visited:

                    #(Debugging).
                    if self.__debugging:
                        print(f"traversing from node {current_node.getNumber()} to {intersection[0].getNumber()}")
                        visited_numbers = []
                        for node in visited:
                            visited_numbers.append(node.getNumber())
                            print(f"visited: {visited_numbers}")
```

```
                self.__traversal(intersection[0], visited)

        #(Debugging).
        if self.__debugging:
            visited_numbers = []
            for node in visited:
                visited_numbers.append(node.getNumber())

            return visited_numbers


    #A method for the traversal which finds the letters that the word must contain for the SUBSTR part of the query

    def __substringValues(self, node):

        substring_values = []

        #Goes through each intersection with the current node.
        #intersections is a list of lists with the format ((intersecting node, (row of intersection, col of interse
        for intersection in node.getIntersections():

            #Checks if the filling_in_word of the intersecting node has a word attributed to it yet.
            if intersection[0].getFillingInWord() != "":

                #Finds the position of the letter (node_word_index) within the current node that is being intersect
                if node.getDirection() == "down":
                    node_word_index = intersection[1][0] - node.getStartRow() + 1
                else:
                    node_word_index = intersection[1][1] - node.getStartCol() + 1

                #Finds which letter is being intersected with (intersecting_letter)
                if intersection[0].getDirection() == "down":
                    intersecting_letter_index = intersection[1][0] - intersection[0].getStartRow() + 1
                    intersecting_letter = intersection[0].getFillingInWord()[intersecting_letter_index - 1]
                else:
                    intersecting_letter_index = intersection[1][1] - intersection[0].getStartCol() + 1
                    intersecting_letter = intersection[0].getFillingInWord()[intersecting_letter_index - 1]

                substring_values.append((node_word_index, intersecting_letter, node.getNumber()))

        return substring_values


    #A method which returns a boolean for whether the graph has been completed or not.

    def completed(self):

        completed = True

        for node in self.__nodes:
            if node.getFillingInWord() == "" or node.getClueWord() == "":
                completed = False

        if self.__empty: completed = False

        return completed

    def displayIntersections(self):

        for node in self.__nodes:
            for vals in node.getIntersections(): print(f"node{node.getNumber()}: node{vals[0].getNumber()} {vals[1
            print()

    def displayNodes(self):

        for node in self.__nodes:
            print(f"Node{node.getNumber()}: (word_length: {node.getWordLength()}, filling_in_word: {node.getFilling

    def getNodes(self):
        return self.__nodes


    #A method which constructs a 2D array to represent the graph.

    def getGrid(self):
```

```
        self.__crossword_display = self.__crossword

        for row in range(len(self.__crossword)):
            for col in range(len(self.__crossword[row])):

                for node in self.__nodes:

                    for row_2 in range(node.getStartRow(), node.getEndRow() + 1):
                        for col_2 in range(node.getStartCol(), node.getEndCol() + 1):

                            if row == row_2 and col == col_2:

                                if node.getDirection() == "down":
                                    index = row - node.getStartRow()
                                else:
                                    index = col - node.getStartCol()

                                self.__crossword_display[row][col] = node.getFillingInWord()[index]

        return self.__crossword_display

def main(size, topics, language):

    i = 0

    shuffle(crosswords)

    crossword = crosswords[i]

    graph = Graph(crossword, "French", topics, empty=True)

    while not graph.completed() and i < len(crosswords):

        #Checks if the size of the crossword is the size you selected.
        if len(crosswords[i]) == size:

            #Construct the graph based on the crossword grid.
            graph = Graph(deepcopy(crosswords[i]), language, topics, debugging=False)

        i += 1

    if graph.completed():

        return graph.getGrid(), graph.getNodes()

    else:

        return [["fail"],["fail"]], []

if __name__ == "__main__":
    print(main(8,["diversite","criminalite"],"French"))
```