Contents

1 Basic Test Results 2

2 wordsearch.py 3

1 Basic Test Results

```
Starting tests...
Sat 08 Jun 2024 12:46:19 IDT
48fc978885dbdbc5e83d99ef569b36babd3d977b  -


Archive:  /tmp/bodek.51vlmx5k/intro2cs2/ex5/daniel.rez/presubmission/submission
  inflating: src/wordsearch.py


Running presubmit code tests...
5 passed tests out of 5 in test set named 'ex5'.
result_code    ex5    5    1
Done running presubmit code tests

Finished running the presubmit tests

Additional notes:

Make sure to thoroughly test your code.

```

2 wordsearch.py

```python
#####################################################################
# FILE : wordsearch.py
# WRITER :  daniel_riazanov, daniel.rez , 336119300
# EXERCISE : intro2cs ex5 2024
# DESCRIPTION: Implementation of search algorithm in matrix based on set of directions. Module practises matrix's,
# dictionaries, command line arguments and file operations. In this exercise I tried to achieve better performance by
# implementing creative solution - mapping all the locations for each char in matrix, which allowed instant excess to
# limited numbers of starting search points instead of blind and repetitive matrix traversing. Consequently, performance
# depends more on the number of directions to search and the length of the words, rather than the size of the matrix.
# STUDENTS I DISCUSSED THE EXERCISE WITH: None
# WEB PAGES I USED: None
# NOTES: None
#####################################################################

import sys
from collections import defaultdict

#  GLOBAL CONSTANTS
# All the defined search directions accordingly to documentation
VALID_DIRECTIONS = {'u', 'd', 'r', 'l', 'w', 'x', 'y', 'z'}
# Dictionary matching to each direction a vector (tuple representing the change in row (x) and column (y)):
# (1 For down, -1 for up, 0 if not moving, 1 For right , -1 for left, 0 if not moving)
DIRECTION_VECTORS = {
    'r': (0, 1), 'l': (0, -1), 'u': (-1, 0), 'd': (1, 0),
    'w': (-1, 1), 'x': (-1, -1), 'y': (1, 1), 'z': (1, -1)
}


def read_wordlist(filename):
    """
    Read a list of words from a file, traversing each line, removing \n escape sequences and appending to a list (thus
    saving the words order).

       Args:
           filename (str): The name of the file containing the word list.

       Returns:
           list: A list of words.

       Exits:
           If the file is not found or cannot be opened, the program exits with an appropriate error message.
    """
    # Declare return container
    list_of_words = []
    # try to open the desired file
    try:
        # with for safe closing process.
        with open(filename, 'r') as f:
            line = f.readline()
            while line:
                list_of_words.append(line.strip())
                line = f.readline()
        return list_of_words
    # if we couldn't open file, also terminate the program with appropriate error type
    except PermissionError:
        print(f"Error while opening the file {filename}.")
        sys.exit(1)


```

```python
def read_matrix(filename):
    """
       Read a matrix of letters from a file.

       Args:
           filename (str): The name of the file containing the matrix.

       Returns:
           list: A list of lists representing the matrix.

       Exits:
           If the file is not found or cannot be opened, the program exits with an error message.
    """
    # Declare return container
    letter_matrix = []
    try:
        # with for safe closing process.
        with open(filename, 'r') as f:
            for line in f:
                # split all letters based on comma between them and add the line of letters as a row to letter_matrix
                matrix_row = line.strip().split(',')
                letter_matrix.append(matrix_row)

        return letter_matrix

    # if we couldn't open file, also terminate the program with appropriate error type
    except PermissionError:
        print(f"Error while opening the file {filename}.")
        sys.exit(1)


def validate_directions(directions):
    """
      Validate the provided directions (str) against the set of the valid ones.

      Args:
          directions (str): A string of directions to validate.

      Exits:
          If any invalid direction is found, the program exits with an error message.
    """
    # convert to set to ensure uniqueness
    provided_directions = set(directions)
    # If not subset of valid directions - there are one or more invalid characters in the input.
    if not provided_directions.issubset(VALID_DIRECTIONS):
        print(f"Invalid directions provided. Directions must be a combination of:", {VALID_DIRECTIONS})
        sys.exit(1)


def write_output(results, filename):
    """
       Write the search results to an output file.

       Args:
           results (list): A list of tuples  (words, their counts).
           filename (str): The name of the output file.
    """
    # With ensures safe proces closure. W mode to crete if not exists, and if exists - to override
    with open(filename, 'w') as f:
        for word, count in results:
            f.write(f"{word},{count}\n")


def is_valid_position(x, y, num_rows, num_cols):
    """
        Check if the position is valid within the matrix boundaries.

        Args:
```

```
            x (int): The row index.
            y (int): The column index.
            num_rows (int): The number of rows in the matrix.
            num_cols (int): The number of columns in the matrix.

        Returns:
            bool: True if the position is valid, False otherwise.
    """
    return 0 <= x < num_rows and 0 <= y < num_cols


def is_word_in_direction(word, matrix, start_x, start_y, delta_x, delta_y):
    """
    Check if the word can be found in the specific direction starting from the potential position (first letter of
    the word) which we mapped as dictionary values

    Args:
        word (str): The word to search for.
        matrix (list): The matrix of letters.
        start_x (int): The starting row index of the letter.
        start_y (int): The starting column index of the letter.
        delta_x (int): The change in the row index for the direction.
        delta_y (int): The change in the column index for the direction.

    Returns:
        bool: True if the word is found, False otherwise.
    """
    word_len = len(word)
    for k in range(word_len):
        # Calculate the new position based on the current position and the direction (in which we're
        # currently traversing the matrix)
        new_x, new_y = start_x + k * delta_x, start_y + k * delta_y
        # Fast check if the new (next) position is valid and matches the next character in the word
        if not is_valid_position(new_x, new_y, len(matrix), len(matrix[0])) or matrix[new_x][new_y] != word[k]:
            return False
    # We found the word from the starting position! (The current word can be fitted within the boundaries and each
    # next character is part of the word)
    return True


def count_word_matches(word, matrix, positions, directions):
    """
       Count times word from the list is presented in matrix for each mapped position and for each chosen direction.

       Args:
           word (str): The word to search for.
           matrix (list): The matrix of letters.
           positions (list): Dictionary values contain a list of suitable (row, col) positions to start the search from.
           directions (str): The directions to search in.

       Returns:
           int: The count of occurrences of the word.
    """
    count = 0
    # Traverse through all suitable positions
    for start_x, start_y in positions:
        # In all directions for each position
        for direction in directions:
            # For critical case when constant valid directions will not correspond to constant vectors (preventing
            # unexpected result)
            if direction in DIRECTION_VECTORS:
                delta_x, delta_y = DIRECTION_VECTORS[direction]
                # for each position and vector, if found match update counter
                if is_word_in_direction(word, matrix, start_x, start_y, delta_x, delta_y):
                    count += 1
    return count
```

```python
def find_words(word_list, matrix, directions):
    """
    Collective function which utilizes previous blocks to find all words in the word list within the matrix based on
    the provided directions. In this func we will convert a matrix of letters into a dictionary. Key - letter,
    value - position in matrix [row_num, num_in_row] - in this way we will bind all the locations to the desired
    character The points for this task are also given for efficiency, so I tried to find out creative solution to not
    traverse again and again blindly to find the beginning letter of each word. In O(1) we will point for the first
    letter for each word, then perform validations for the possible next letter from the current position and only
    then continue traversing (We already studied dictionaries).

        Args:
            word_list (list): A list of words to search for.
            matrix (list): The matrix of letters.
            directions (str): The directions to search in.

        Returns:
            list: A list of tuples containing words and their counts.
    """
    # Convert the matrix to a dictionary of character positions
    char_positions = defaultdict(list)
    # Iterate over each row in the matrix
    for row_index, row in enumerate(matrix):
        # Iterate over each character in the row
        for col_index, char in enumerate(row):
            # Append the position (row, col) to the list of positions for the character
            char_positions[char].append((row_index, col_index))

    # Declare container for the results (tuples word, count)
    results = []

    for word in word_list:
        # Fast check to decide whether word exists in matrix (by first char).
        if word[0] in char_positions:
            # Point from whole matrix some positions to check from it to all the necessary directions
            start_positions = char_positions[word[0]]
            # Count number of times the word appeared in the matrix
            word_count = count_word_matches(word, matrix, start_positions, directions)
            # Only if word appeared, at to results
            if word_count > 0:
                results.append((word, word_count))
    # If no word from the word_list appeared in matrix, return blank list (so later on we will return blank file
    # accordingly to requirements)
    return results


def main():
    """
        Main function to execute the word search program.

        This function archives the final result by performing the following algorithm:
        1. Validates the number of command-line arguments.
        2. Reads and validates the word list and matrix from the provided files.
        3. Validates the search directions.
        4. Finds the occurrences of each word from the list in the matrix based on the specified directions.
        5. Writes the results to the output file.

        Command-line arguments:
            sys.argv[1]: The filename containing the list of words.
            sys.argv[2]: The filename containing the matrix of letters.
            sys.argv[3]: The filename for the output results.
            sys.argv[4]: A string representing the search directions.

        Exits:
            If the number of arguments is not 4, the function prints an error message and exits.
            If any file operation fails the function prints an error message and exits.
            If the directions provided are invalid, the function prints an error message and exits.
    """
    # If the number of parameters isn't 4, message user and terminate program
```

```
    if len(sys.argv) != 5:
        print("Incorrect number of parameters.")
        sys.exit(1)
    # Assigning variables with parameters with values from command line input
    word_file = sys.argv[1]
    matrix_file = sys.argv[2]
    output_file = sys.argv[3]
    directions = sys.argv[4]

    # Attempting to pass to serving function str parameters from command line, if one of them is invalid we
    # will terminate the program immediately on him and print appropriate message
    word_list = read_wordlist(word_file)
    matrix = read_matrix(matrix_file)
    validate_directions(directions)

    # Main function
    results = find_words(word_list, matrix, directions)
    # Writing results to a file
    write_output(results, output_file)


if __name__ == '__main__':
    main()
```