CS 461 - Artificial Intelligence

Project Report

YAINDU

Members:

Umer Shamaan - 21701563,

Taha Khurram - 21701824,

Muhammad Ali Khaqan - 21701812,

Hassan Raza - 21701811

Table Of Contents:

1. Introduction

This program is written to solve the New York Times 'The Mini Crossword - By Joel Fagliano'. The program does so by downloading the puzzle and clues from the website. The puzzle, clues, and its answers are revealed by the program after which the program starts to solve the puzzle.

The program searches each clue on wikipedia[1], and in PyDictionary[2], obtaining relevant results from them. This data is stored and the invalid words, which do not meet the word length are filtered out. Afterward, we compare the current candidate words with the answers taken from the website to choose a definitive answer. This answer is then used to shorten the candidate list for other clues that are intersected by this answer to slowly calculate each word. In the case of our program not having an answer it will try to guess a word from its list of candidates to find a possible answer.

Finally, the answers found are revealed on the right-hand side of the GUI window where it can be easily compared to the answers revealed from the puzzle kept on the left hand side.

2. Implementation

The implementation of our project has two major components, the GUI and the backend implementation. GUI deals with how the puzzle is displayed and the hints are shown. For the backend, we have to do two tasks, one is to interact with GUI to download the content of the puzzle to use it and upload answers generated for the puzzle while the other task is to do the predictions for answers using AI techniques.

2.1 System Decomposition

Our System has to major system decompositions as said above:

- Implementation of the logical sub-system is done in classes of findWord.py, code.py, and fillWord.py.
- GUI of our system is made in the class named code.py.

2.2 Logical Subsystem

Description of methods used:

```
def possible_answers(clues, length):
    myarr = wikipedia_solution(clues)
    myarr = numpy.array(myarr)
    myarr = numpy.concatenate([myarr, findWords(clues)])
    myarr = numpy.concatenate([myarr, wikipedia_sentence_solution(clues)])
    finalResult = []
    returnResult = []
    for i in myarr:
    #if len(i) <= length:
       if i not in finalResult:
        finalResult.append(i.upper())
        if p.plural(i.upper()):
            finalResult.append(p.plural(i.upper()))

        if p.singular_noun(i.upper()):
            finalResult.append(p.singular_noun(i.upper()))

    for j in finalResult:
        #sorting based on length
        if len(j) == length:
            if j not in returnResult:
                returnResult.append(j)

    return returnResult
```

Figure 1 - possible_answers()

1- possible_answers():

This method receives the clue and the length of the answer. Then we call methods called wikipedia_solution(), wikipedia_sencter_solution() which search for the keywords for our clue on Wikipedia (The code snippet is provided below). Then we call the findWords() method and find the synonyms, antonyms, plural and singular forms for each word. We then filter out possible answers based on the length of the word expected and return the answers as an array to the callee.

```python
def wikipedia_solution(wikipedia_clues):
    WIKIPEDIA_API = "https://en.wikipedia.org/w/api.php?action=query&utf8=&format=json&list=search&srlimit=50&srsearch="
    stop = stopwords.words('english') + list(string.punctuation)
    #clue_mapping = dict()
    #print("wikipedia fetching word for " + wikipedia_clues)
    if '#' in wikipedia_clues:
        wikipedia_clues = wikipedia_clues.replace("#", "")


    for sentence in wikipedia_clues:
        req = requests.get(WIKIPEDIA_API + sentence)
        wiki_json = ast.literal_eval(req.text)
        #print(wiki_json["query"])
        solutions = list(set([word for word in [word for word in word_tokenize(info["title"].lower()) if word not in stop] for i
    return solutions

def wikipedia_sentence_solution(wikipedia_clues):
    WIKIPEDIA_API = "https://en.wikipedia.org/w/api.php?action=query&utf8=&format=json&list=search&srlimit=50&srsearch="
    stop = stopwords.words('english') + list(string.punctuation)
    #clue_mapping = dict()
    #print(">>> STARTING WIKI FETCH sentence.....")
    if '#' in wikipedia_clues:
        wikipedia_clues = wikipedia_clues.replace("#", "")
    req = requests.get(WIKIPEDIA_API + wikipedia_clues)
    wiki_json = ast.literal_eval(req.text)
    solutions = list(set([word for word in [word for word in word_tokenize(info["title"].lower()) if word not in stop] for info
    return solutions
```

Figure 2- wikipedia_solution() & wikipedia_sentece_solution()

```
dictionary = PyDictionary()
def findWords(stri):
    result = []
    data = re.split(', |_|-|!| ', stri)
    for words in data:
        blockPrint()
        synonym = dictionary.synonym(words)
        antonym = dictionary.antonym(words)
        enablePrint()
        if not antonym and not synonym:
            temp = []
        elif not synonym and antonym:
            temp = antonym
        elif not antonym and synonym:
            temp = synonym
        else:
            temp = numpy.concatenate([synonym, antonym])
        result = numpy.concatenate([result, temp])

    return result
```

Figure 3 - findWords()

```
def fillWords(results_array, sample_array2, list_sample, notfound_array, word_size):
    index = [-1, -1]
    for i in range(0, len(sample_array2)):
        try:
            index[1] = sample_array2[i].index(list_sample[0])
            #print("test", i)
            #print(sample_array2[i].index(list_sample[0]))
            index[0] = i
            #print(index[0])
        except:
            pass

    #print(index[0],index[1])

    if(list_sample[1] == "across"):
        word_index = -1
        for i in range(index[1], word_size + index[1]):
            word_index += 1
            if notfound_array[index[0]][i] != '*':
                temp_result = []
                for x in range(len(results_array)):
                    #print(results_array[x][word_index], notfound_array[index[0]][i], word_index)
                    if results_array[x][word_index] == notfound_array[index[0]][i]:
                        temp_result.append(results_array[x])
                results_array = temp_result
```

Figure 4- fillWords

2- fillWords():

This method maps each character found on our puzzle to shortlist possible solutions based on the character's position in the word. For example, A word 'MOM' has O on index 1. The program then traverses the results_array and short lists the words that have 'O' at index 1 as each word is of length 3. This results in us having shortlisted the possible solution and is done for each new word revealed/found on the puzzle.

```
def fillSolution(result, sample_array2, list_sample, notfound_array, word_size):
    index = [-1, -1]
    for i in range(0, len(sample_array2)):
        try:
            index[1] = sample_array2[i].index(list_sample[0])
            index[0] = i
        except:
            pass

    if(list_sample[1] == "across"):
        word_index = -1
        for i in range(index[1], word_size + index[1]):
            word_index += 1
            if notfound_array[index[0]][i] == '*':
                notfound_array[index[0]][i] = result[0][word_index]

    else:
        word_index = -1
        #down the table, R X C
        for i in range(index[0], word_size + index[0]):
            word_index += 1
            #print(notfound_array[i][index[1]], result[0][word_index], word_index)
            if notfound_array[i][index[1]] == '*':
                notfound_array[i][index[1]] = result[0][word_index]

    return notfound_array
```

Figure 5 - fillSolution()

3- fillSolution():

The method finds the beginning index of the clue on our 2D matrix, using these indices the program maps the solution onto its designated position in the 2D matrix which is returned to the callee who then uses the data received from this 2D solution matrix to map our answers onto the GUI.

```
# sample_array = [['*','*','*','*','*'],
#                 ['*','*','*','*','*'],
#                 ['*','*','*','*','*'],
#                 ['*','*','*','*','*'],
#                 ['*','*','*','*','*']]

#positioning
# sample_array2 = [[1, 2, 3, 4, 5],
#                  [6, 0, 0, 0, 0],
#                  [7, 0, 0, 0, 0],
#                  [8, 0, 0, 0, 0],
#                  [9, 0, 0, 0, -1]]

#                  #map_array

# list_sample = [6, "across"]

# sample_array = [['P','A','R','I','S'],
#                 ['O','*','*','*','*'],
#                 ['O','*','*','*','*'],
#                 ['C','*','*','*','*'],
#                 ['H','*','*','*','!']]
```

Figure 6 - 2D Matrix structure

In this method, Asterisk '*' means an empty place, and an exclamation mark '!' means a black box (sample 2D matrix structure is provided above. Moreover, another 2-D array is created for the positioning of the alphabets. In this method, this positioning is dependent on the clue number. '-1' in the positioning array means that we are not using that specific line because it is a black box. This 2-D array depends on the type of puzzle that is available each day.

Solution Process:

Firstly, all the words which are revealed at the initialization of the program are stored in a dictionary which contains the clue number, clue itself, word orientation (across or down), and word size. For instance, for the first clue, after the puzzle has been revealed, we know that the answer is ‘Paris’.

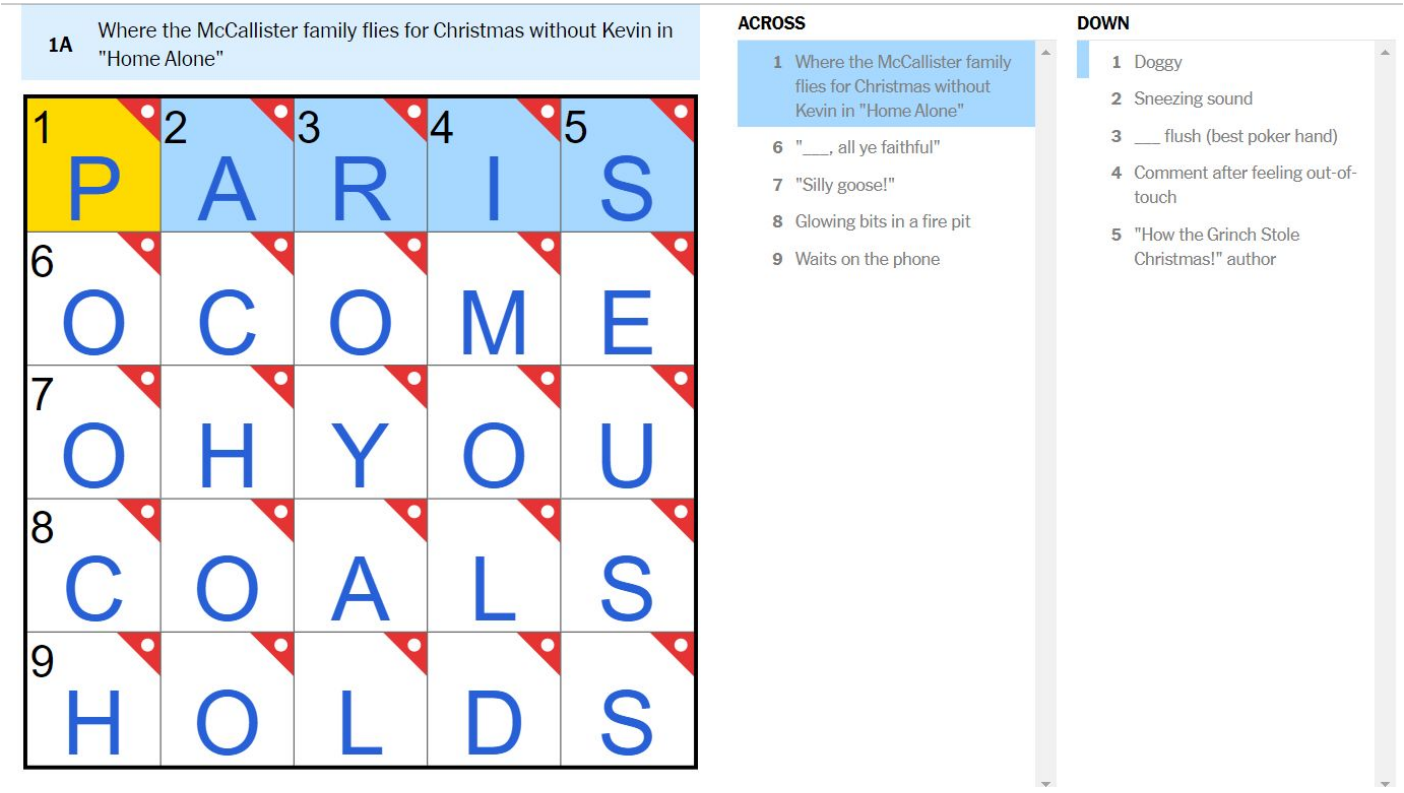

Figure 7 - Sample daily puzzle

This result will be stored as:
{"number": 1, "dir": "across", "clue": “Where the McCallister family flies for Christmas without Kevin in "Home Alone", "answer": PARIS, "length": 5}, This allows for easy access to each clue in our code and in case our program gets stuck at one clue the code can easily find the answer for that clue following this structure.

The following explains the process through which the program finds, stores, and eliminates candidate words and then calculates the answer for the puzzle:

1. Pick the entire phrase of the clue and check it word by word i.e., check each word on Wikipedia and find relevant key terms, then use PyDictionary for its synonyms, antonyms, plural and singular forms. The above process is repeated for each of the clues until we have arrays of synonyms, antonyms, plurals, and singulars for each of the words in each of the phrases of the clues.
2. Pick the entire clue and search for key terms regarding the clue on Wikipedia, for each key term found use PyDictionary for synonyms, antonyms, singular and plural forms.

Note: This process is repeated for the entire phrase to get the related answers for the entire clue phrase and stored in the same array for each of the given clues.

3. After the list of words has been stored as strings in arrays, we check the length of the stored words and try to find the words that match the upper limit of length of the puzzle, and remove the rest of the words from the given list. For instance, if the word length is 5, only words such as 'PARIS' that have a string length of 5 remain in the list and all the words with a string length either smaller or greater than 5 are removed from the given array to improve efficiency and increase the chances of finding the right answer to a given clue.

Result: This leaves us with all the possible candidate solutions for the given clues in a compact list of words. The procedure above is repeated for all the clues, whether across or down the puzzle, and the shortened list of words for each of the clues is stored in an array. This leaves us with a multitude of arrays consisting of a plethora of words relevant to the clues provided for the given crossword puzzle.

4. Since we already have the solution, we go to the list of words and try to find whether the solution exists in the given list or not for a specific clue. If the given answer does not exist in the list of words relevant to a particular clue, the given word is revealed on the puzzle.

5. A method named ‘Fillwords’ is created to find the words relevant to the clues from the list of words stored in the arrays by the searching algorithm explained above.

6. We have created a result array that, as explained before, stores the answers in an array depending on the word size that is specified by the puzzle length. If the required word is not found in the result array, this specific word is revealed in the ‘NOTFOUND’ array (See the appendix). This step is repeated for each of the given clues.

7. We know that a 2-D array has a Row and Column format. After we have filled the first puzzle row, we look at the first alphabet in the first row and try to find all the words in the given list that start with this alphabet. We try to limit the words in the list and try to find the specific answer that matches the clue. After this word has been found, it fills the first column of the puzzle. We will repeat the above procedure and continue the process until all of the puzzle space has been filled.

8. Once all the words have been filled, the resultant 2D matrix is used by the program to update the GUI where the puzzle on the right-hand side is filled with the program’s calculated solutions and can be compared visually to the actual puzzle on the left-hand side of the screen.

3. Test Results

Brief explanation of panels:

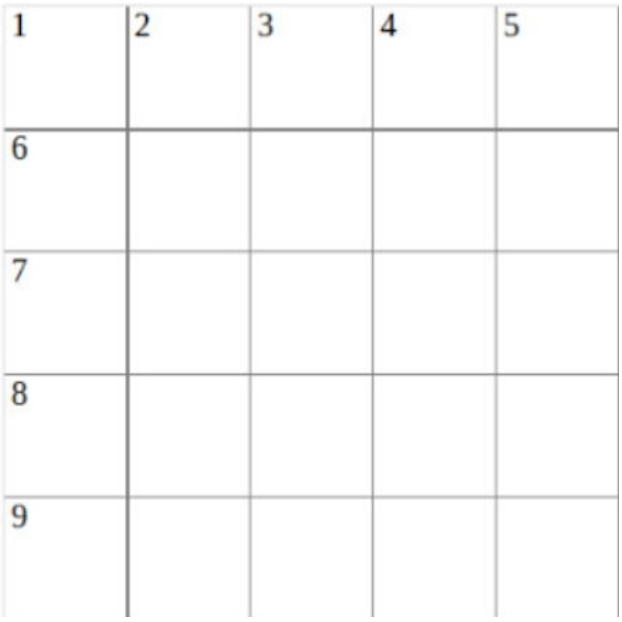

Figure 8 - Empty box before initializing

Figure 9 - Puzzle containing answers (New York Mini puzzle)

Across

1: The "white" in "White Christmas"

5: Mystical glow

6: The elf in "Elf"

7: Precipice

8: Layer of a wedding cake

Down

1: ___ Arabia

2: Gentle prod

3: Rodentia or Carnivora

4: "No freaking ___!"

6: Show strength in poker

Figure 10 - Clues

	1 P	2 O	3 A	
4 B	O	Z	N	5 N
6 T	R	A	N	B
7 S	T	R	E	A
	8 S	K	S	

Figure 11 - Program's solution to the puzzle

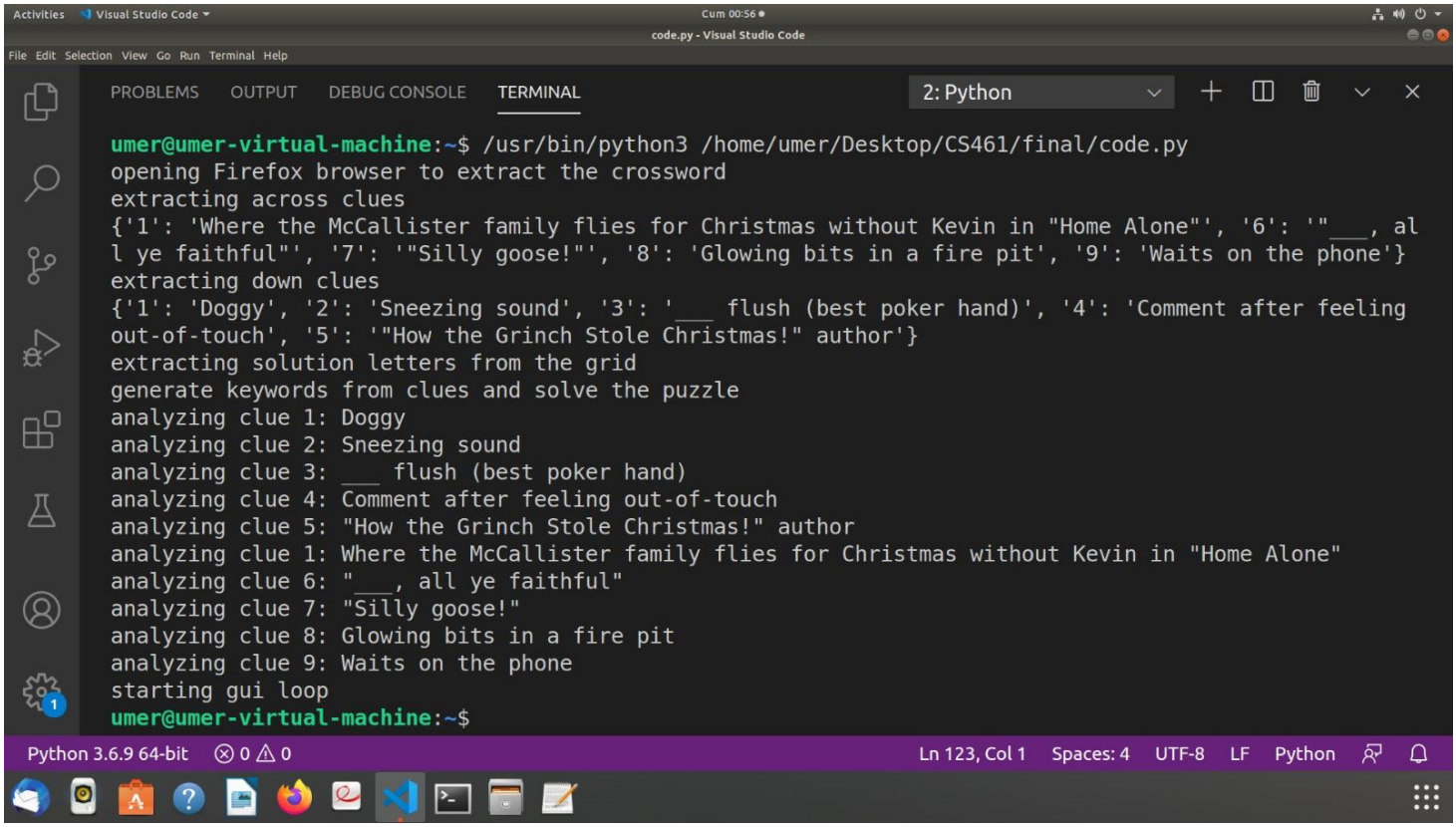

Figure 12 - Console output

Below are the results for some of the puzzles that we found.
The Grid on the left shows actual answers while the grid on the right shows the answers predicted by our program.

18th December 2020:

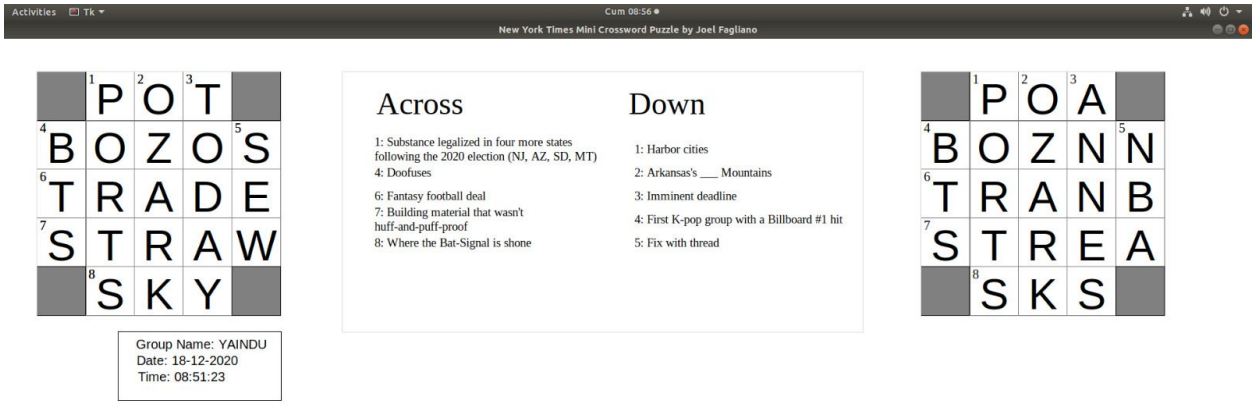

Figure 13 - Puzzle 18-12-2020

This was the puzzle for Friday which is supposed to be the most difficult. We were able to get three correct answers. For others, we were not able to predict correctly because we could not get the correct words using similar words to the hints.

21st December 2020:

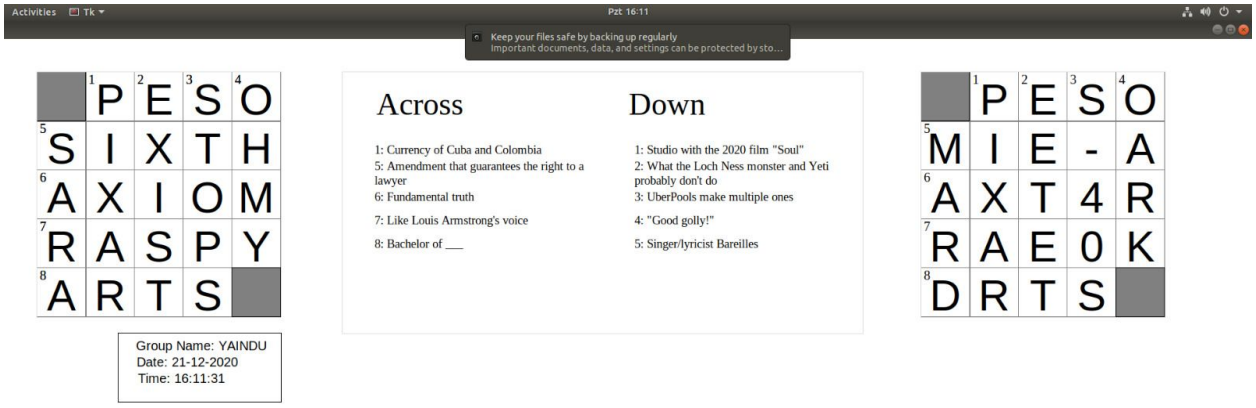

Figure 14 - Puzzle 21-12-2020

This puzzle was for Monday which is supposed to be the easiest, however, our number of correct predictions was very few for this day as we predicted only one word correctly.

This is as a result of Monday being an easy puzzle which is mostly answering easy clues, as our program does not deal with language processing it has trouble with such puzzles.

22nd December 2020:

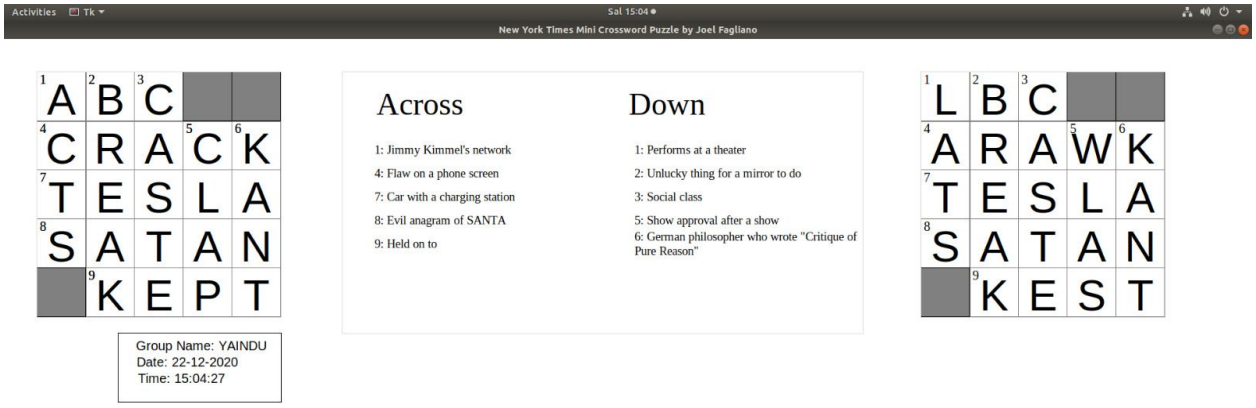

Figure 15 - Puzzle 22-12-2020

This puzzle is for Tuesday, as difficulty increases every next day of the week, so this was supposed to be more difficult than Monday. However we were able to get four correct answers for this puzzle.

23rd December 2020:

Attempt 1:

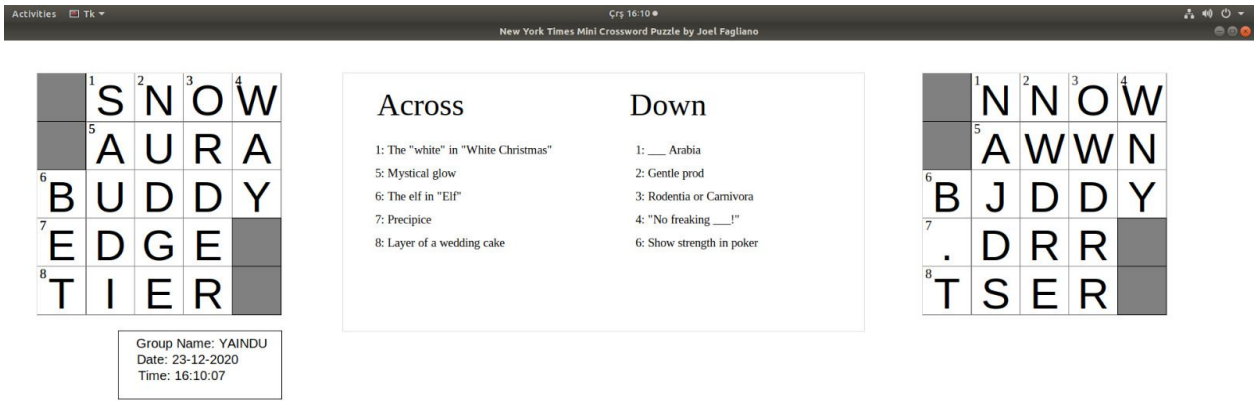

Figure 16 - Puzzle 23-12-2020 Attempt 1

Attempt 2:

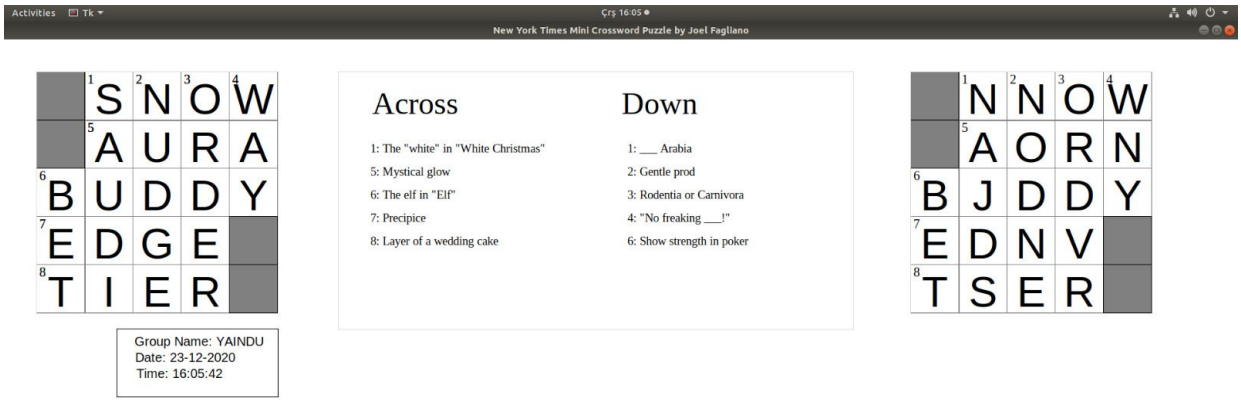

Figure 17 - Puzzle 23-12-2020 Attempt 2

This puzzle is for Wednesday, we made two attempts this day. It shows that the number of correct answers changes in both attempts. This is because sometimes there is more than one solution in our array of possible solutions for a given hint and all of them fit the puzzle when we check them with respect to the answers that are disclosed. In such cases, any of these solutions can be taken thus creating a difference in the solutions.

24th December 2020:

Attempt 1:

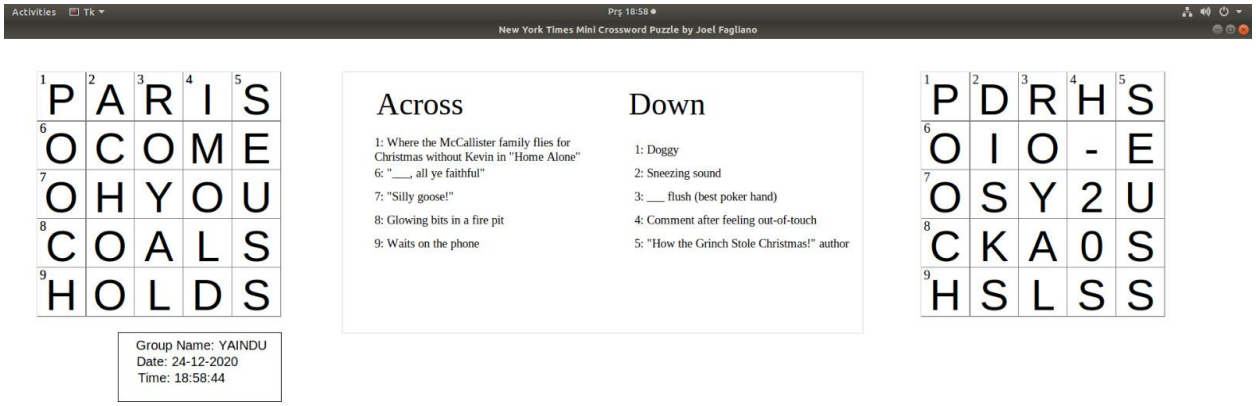

Figure 18 - Puzzle 24-12-2020 Attempt 1

Attempt 2:

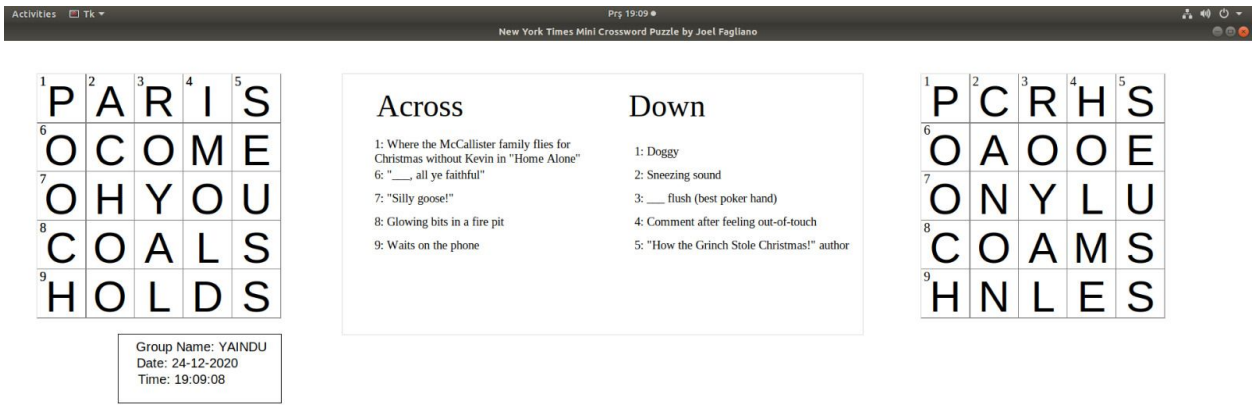

Figure 19 - Puzzle 24-12-2020 Attempt 2

This was the puzzle for Thursday. That day we again had two attempts. In both attempts, we predicted three correct answers.

4. Research Papers

1- Using clue-word associations, the semantic route looks for potential responses. The orthographic route, on the contrary, obtains candidate answers using letter-word associations and combinations. The findings of a research paper that suggests how the two routes can be applied separately indicate that experts rely on techniques similar to the computer-based model for rapid memory search and retrieval [3].

We implemented a similar method to solve this problem; not only the letter word associations are employed to specify the word search but also clue-word associations are heavily looked upon to narrow down the search to potential correct answers.

2- Another research paper discusses the methods for improving the efficiency of automatic extraction of candidate responses for an extremely difficult task: the automatic resolution of Italian language crossword puzzles. The approach to solving this puzzle consists of querying the database (DB) with a search engine and translating its output into a probability score that incorporates both the search engine score and statistical similarity features in a single scoring model, i.e., a logistic regression model. The resolution accuracy of crossword puzzles is significantly influenced by this enhanced retrieval model [4]. However, this method turned out to be far more complicated to implement as an algorithm, so we decided not to entirely implement it but use the specific parts of the method that we liked such as using only the Wikipedia search engine to look for keywords.

3- Two natural optimization problems are defined in a research paper that is essentially useful to us: maximizing the likelihood of the right solution and maximizing the number of correct words in the solution (variable values) [5]. Therefore, this paper explains how we can apply an appropriate iterative approximation similar to turbo decoding and present findings on a set of real and artificial crossword puzzles. A method analogous to this as explained above helped to speed the optimization process, thereby providing fast and accurate results to the puzzle.

4- Another research paper that is of particular interest to us, studies the automatic resolution of crossword puzzles. According to this paper, automated solvers are based on two solution retrieval modules - a web search engine, e.g. Google, Bing, etc., and a database (DB) system to access crossword puzzles previously resolved. This research paper shows that both modules above can be improved by learning to rank models based on relational syntactic structures identified between the clues and the response, particularly improving the resolution accuracy by 15% [6]. We implemented a similar scheme to solve the given crossword puzzle, where we not only rely on the search engine to find candidate answers but also look for solutions from the solved puzzle to help us reach the correct answer in a particular row and column.

5- This research paper proposes a semantic search method for generating candidate response lists by using the lexical relationships encoded in WordNet, the lexical database for English, for concept type clues. Automatic crossword puzzle resolution is an open task in natural language that involves the filling of the puzzle grid with candidate responses, thus meeting the limitations of the grid. Besides, it has a significant effect on the efficiency of the automated crossword resolution challenge to include a correct list of response candidates [7]. We took the idea of finding an accurate list of candidate solutions from this paper.

Appendix

Code.py

```
import tkinter as tk #gui library
from selenium import webdriver  #for web scrapping
from selenium.webdriver.common.keys import Keys
import time
from time import gmtime, strftime
from datetime import datetime, timedelta
from findWords import possible_answers
from fillWords import fillWords, fillSolution

"""
This program creates a mini crossword puzzle similar to the one at
www.nytimes.com/crosswords/game/mini.
It extracts the puzzle data from the same websites. It uses the Selenium library in order to extract
relevant infomration from the website. Furthermore, the Tkinter framework is used to build the gui
components. After that it tries to solve the puzzle. It extracts some key words like synonyms, antonyms
and other related words from the clues given from APIs. It uses those to generate matching words.


@authors:   Umer Shamaan
        Taha Khurram
        Hassan Raza Warraich
        Muhammad Ali Khaqan
"""



cellSize = 75  #default size of a grid cell


clueList = []  #store the clues and their information

num_arr = [[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]]


letter_arr = [['!', '!', '!', '!', '!'],
        ['!', '!', '!', '!', '!'],
```

```
          ['!', '!', '!', '!', '!'],
          ['!', '!', '!', '!', '!'],
          ['!', '!', '!', '!', '!']]

to_be_solved_arr = [['!', '!', '!', '!', '!'],
          ['!', '!', '!', '!', '!'],
          ['!', '!', '!', '!', '!'],
          ['!', '!', '!', '!', '!'],
          ['!', '!', '!', '!', '!']]


final_sol_arr = [['!', '!', '!', '!', '!'],
          ['!', '!', '!', '!', '!'],
          ['!', '!', '!', '!', '!'],
          ['!', '!', '!', '!', '!'],
          ['!', '!', '!', '!', '!']]


def createGrid(grid):
    wid = 0
    while wid < (5 * cellSize)+1:
        lent = 0
        while lent < (5 * cellSize)+1:
            grid.create_rectangle(wid, lent, cellSize, cellSize, outline = 'grey')
            lent = lent + cellSize
        wid = wid + cellSize
    return


#to fill the wordless blocks
def fillGrey(grid, i, j):
    grid.create_rectangle(i * cellSize, j * cellSize, i * cellSize + cellSize, j * cellSize + cellSize, fill='grey')
    return


# filling grid cells with the solution letters
lSize = 50 #the size of the letter
def fillWord(grid, lent, wid, letter):
    x = cellSize * wid + 38
    y = cellSize * lent + 43
    wid=grid.create_text(x, y, text=letter, font=("Arial", lSize))
    r=grid.create_rectangle(wid * cellSize, lent * cellSize, wid * cellSize + cellSize, lent * cellSize +
cellSize,  fill="green")
    grid.tag_lower(r,wid)
```

```
    return


# Printing the clues for accross
tWidth = 350 #max width of text section
def across():
    clues.create_text(120,50, font=('Times',36),text='Across')
    size = 0
    for i in acrossClues:
        clues.create_text(50, 120 + size,font=('Times',14), anchor='w', text= i + ": " + acrossClues[i],
width=tWidth)
        #clues.create_text(50, 120 + size,font=('Times',14), anchor='w', text= i + ": " + str(acrossClues[i]),
width=tWidth)
        size = size + 36
    return


#filling grid numbers
nSize = 15
def markN(grid, lent, wid, num):
    x = cellSize * wid + 10
    y = cellSize * lent + 12
    grid.create_text(x, y, font=("Times", nSize), text = num)
    return


# Printing the clues for down
def down():
    #clues.create_text(450,50,font=('Times',36),text='Down')
    clues.create_text(500,50,font=('Times',36),text='Down')
    size = 0
    for i in downClues:
        clues.create_text(450, 120 + size,font=('Times',14), anchor='w', text= i + ": " + downClues[i],
width=tWidth)
        size = size+36
    return


print("opening Firefox browser to extract the crossword")
# obtaining data from newyork times puzzle throught webscrapping
driver = webdriver.Firefox() #open firefox browser
#driver.minimize_window()
driver.set_page_load_timeout(50)
driver.get("https://www.nytimes.com/crosswords/game/mini")                # navigate to website
```

```
driver.find_element_by_xpath("//button[@aria-label='OK']").send_keys(Keys.ENTER)       # press enter
driver.find_element_by_xpath("//button[@aria-label='reveal']").click()              # click on the reveal link
driver.find_element_by_xpath('/html/body/div[1]/div/div/div[4]/div/main/div[2]/div/div/ul/div[2]/li[2]/ul/
li[3]/a').click()    #click reveal
driver.find_element_by_xpath("//button[@aria-label='Reveal']").send_keys(Keys.ENTER)        #press
enter to exit popup
driver.find_element_by_xpath("/html/body/div[1]/div/div[2]/div[2]/span").send_keys(Keys.ENTER)
#press enter to exit popup
#driver.find_element_by_css_selector('minimodal-congrats-message').send_keys(Keys.ENTER)
#CongratsModal-subscriptionUpsell--2tbB2
element = driver.find_element_by_xpath("/html/body")         #extract the body tag from html
time.sleep(1)
content = element.get_attribute("innerHTML")    #put the extracted html data to variable
time.sleep(1)
driver.quit()      #exit browser



# creating the window
window = tk.Tk()
window.configure(background = 'white')
window.title("New York Times Mini Crossword Puzzle by Joel Fagliano")
window.geometry("1800x900") #setting window size

# creating puzzle and answer grid
grid = tk.Canvas(window,width=cellSize * 5,height=cellSize * 5, bg='white')
grid.place(x=50,y=50)
#createGrid()
createGrid(grid)


ans_grid = tk.Canvas(window,width=cellSize * 5,height=cellSize * 5, bg='white')
ans_grid.place(x=1410,y=50)
#createGrid()
createGrid(ans_grid)




#############################################################################################
#
#printing number labels of cells to main grid
temp = 0
```

```
i = 0
j = 0
while temp < 25:
    indexLetter = 'id="cell-id-' + str(temp)
    indexLetter = indexLetter + ""
    si = len(indexLetter)
    letter = content[content.find(indexLetter) + (si + 8):content.find(indexLetter) + (si + 18)]

    if letter == 'Cell-block':
        lent = int(temp / 5)
        wid = temp % 5
        fillGrey(grid, wid, lent)
        #temp=temp+1
        num_arr[i][j] = -1
        #continue

    else:
        indexStart = content.find('text-anchor="start"', content.find(indexLetter),
content.find('text-anchor="middle"', content.find(indexLetter)))
        if indexStart != (-1):
            markN(grid, int(temp / 5), (temp % 5), content[content.find("</text>", indexStart) + 7])
            #num_arr[i][j] = content[content.find("</text>", indexStart) + 7]
#####################################################################
            num_arr[i][j] = int(content[content.find("</text>", indexStart) + 7])

    temp+=1
    j+=1
    if(j >4):
        j = 0
        i+=1



#########################################################################################################
#########

#########################################################################################################
#
#printing number labels of cells to solution grid
temp = 0
while temp < 25:
    indexLetter = 'id="cell-id-' + str(temp)
    indexLetter = indexLetter + ""
    si = len(indexLetter)
```

```
    letter = content[content.find(indexLetter) + (si + 8):content.find(indexLetter) + (si + 18)]

    if letter == 'Cell-block':
        lent = int(temp / 5)
        wid = temp % 5
        fillGrey(ans_grid, wid, lent)
        #temp=temp+1
        #continue

    else:
        indexStart = content.find('text-anchor="start"', content.find(indexLetter),
content.find('text-anchor="middle"', content.find(indexLetter)))
        if indexStart != (-1):
            markN(ans_grid, int(temp / 5), (temp % 5), content[content.find("</text>", indexStart) + 7])

    temp+=1

##########################################################################################
#########


print("extracting across clues")
# create clues
clues = tk.Canvas(window,height = 400,width=800, bg='white')
#clues.place(x=600,y=40)
clues.place(x=520,y=50)
downClues = dict()
acrossClues = dict()


index = content.find("</span>", content.find("Across"))
initial = 0
fina = 5
while initial < fina:
    initial =initial+1
    end = content.find("<", content.find(">", index + 8))
    acrossClues[content[index-1]] = content[(content.find(">", index + 8))+1:end]
#############################################################
    #acrossClues[int(content[index-1])] = content[(content.find(">", index + 8))+1:end]

    #clue = dict(pos = content[index-1], content = content[(content.find(">", index + 8))+1:end], length =
0)
    #cluseList.append(clue)
```

```
    index = content.find("</span>", end + 10)

print(acrossClues)
across()


print("extracting down clues")
index = content.find("</span>", content.find("Down", end))
initial = 0
while initial < fina:
    initial =initial+1
    end = content.find("<", content.find(">", index + 8))
    downClues[content[index-1]] = content[(content.find(">", index + 8))+1:end]
############################################################
    #downClues[int(content[index-1])] = content[(content.find(">", index + 8))+1:end]

    #clue = dict(pos = content[index-1], content = content[(content.find(">", index + 8))+1:end], length =
0)
    #cluseList.append(clue)

    index = content.find("</span>", end + 10)

print(downClues)
down()


print("extracting solution letters from the grid")
#filling the main grid with the solution letters
temp = 0
i = 0
j = 0
while temp < 25:
    indexLetter = 'id="cell-id-' + str(temp)
    indexLetter = indexLetter + ""
    si = len(indexLetter)
    letter = content[content.find(indexLetter) + (si + 8):content.find(indexLetter) + (si + 18)]


    if letter == 'Cell-block':
        lent = int(temp / 5)
        wid = temp % 5
        fillGrey(grid, wid, lent)
        #temp=temp+1
        #continue
```

```
    else:
        indexStart = content.find('text-anchor="start"', content.find(indexLetter),
content.find('text-anchor="middle"', content.find(indexLetter)))
        if indexStart != (-1):
            markN(grid, int(temp / 5), (temp % 5), content[content.find("</text>", indexStart) + 7])
        fillWord(grid, int(temp / 5), (temp % 5),
content[content.find('</text>',content.find('text-anchor="middle"', content.find(indexLetter)))-1])
        letter_arr[i][j] = content[content.find('</text>',content.find('text-anchor="middle"',
content.find(indexLetter)))-1]
        to_be_solved_arr[i][j] = '*'
        final_sol_arr[i][j] = '*'

    temp=temp+1
    j+=1
    if j > 4:
        j = 0
        i+=1

#final_sol_arr = to_be_solved_arr.copy()


#generating dictionaries to associate clues with their answers
#genrating for down
for j in range(5):
    word = ""
    i = 0
    first =  True
    num = 0
    while num_arr[i][j] == -1:
        i+=1

    if i < 5:
        while num_arr[i][j] != -1 and i < 5:
            if(first):
                num = num_arr[i][j]
                first = False
            word += letter_arr[i][j]
            i+=1
            if i > 4:
                break
```

```
        #thisdict = {"number": num, "dir": "down", "clue": downClues[num], "answer": word, "length": len(word)}################################
        thisdict = {"number": num, "dir": "down", "clue": downClues[str(num)], "answer": word, "length": len(word)}


        #cluseList.append(dict("number"= num, "dir"= "down", "clue"= downClues[num], "answer"= word, "length"= len(word)))
        clueList.append(thisdict)


#genrating for across
for i in range(5):
    word = ""
    j = 0
    first =  True
    num = 0
    while num_arr[i][j] == -1:
        j+=1

    if j < 5:
        while num_arr[i][j] != -1 and j < 5:
            if(first):
                num = num_arr[i][j]
                first = False
            word += letter_arr[i][j]
            j+=1
            if j > 4:
                break

        #thisdict = {"number": num, "dir": "across", "clue": acrossClues[num], "answer": word, "length": len(word)}################################
        thisdict = {"number": num, "dir": "across", "clue": acrossClues[str(num)], "answer": word, "length": len(word)}


        #cluseList.append(dict("number"= num, "dir"= "across", "clue"= acrossClues[num], "answer"= word, "length"= len(word)))
        clueList.append(thisdict)


#HARD CODED ONLY FOR NOW!!
```

```
print("generate keywords from clues and solve the puzzle")
line_clue_list = []
for i in range(len(clueList)):
    print("analyzing clue " + str(i))
    pos_answers = possible_answers(clueList[i]["clue"], clueList[i]["length"])
    #line_clue_list.append(possible_answers(clueList[i]["clue"], clueList[i]["length"]))
    line_clue_list.append(pos_answers)


not_to_include = [] #represents the index of those words not included in the sol array (impossible to compute) above so they will not be considered
to_include = []
index_clue_list = []
index_clue_list_2 = []
for i in range(len(line_clue_list)):
    if clueList[i]["answer"] not in line_clue_list[i]:
        not_to_include.append({"number": clueList[i]["number"], "dir": clueList[i]["dir"],  "ans": clueList[i]["answer"], "length": clueList[i]["length"]})
        index_clue_list_2.append(line_clue_list[i])
    else:
        to_include.append({"number": clueList[i]["number"], "dir": clueList[i]["dir"],  "ans": clueList[i]["answer"], "length": clueList[i]["length"]})
        index_clue_list.append(line_clue_list[i])

# for i in range(len(num_arr)):
#     print(num_arr[i])

# for i in range(len(to_be_solved_arr)):
#     print(to_be_solved_arr[i])

#filling the puzzle grid with the words for which the solution could not be determined
for i in range(len(not_to_include)):
    cur = not_to_include[i]
    a = 0
    b = 0
    direction = ""
    ans = ""
    length = 0
    done = False
    for x in range(5):
        for y in range(5):
            if num_arr[x][y] == cur["number"]:
```

```
                a = x
                b = y
                direction = cur["dir"]
                length = cur["length"]
                ans = cur["ans"]
                done = True
                break

        if done:
            break

    if direction == "down":
        index = 0
        for p in range(a, length):
            to_be_solved_arr[p][b] = ans[index]
            index+=1
    else:
        index = 0
        for p in range(b, length):
            to_be_solved_arr[a][p] = ans[index]
            index+=1

# print("----------------------------------------------------------------------")

# for i in range(len(to_be_solved_arr)):
#     print(to_be_solved_arr[i])

# print("final_sol_arr")
# for j in range(len(final_sol_arr)):
#     print(final_sol_arr[j])
# print("----------------------------------------------------------------------")


#generating words
for i in range(len(to_include)):
    resultant_words = fillWords(index_clue_list[i], num_arr, [to_include[i]["number"],
to_include[i]["dir"]], to_be_solved_arr, to_include[i]["length"])
    #to_be_solved_arr = fillSolution(resultant_words, num_arr, [to_include[i]["number"],
to_include[i]["dir"]], to_be_solved_arr, to_include[i]["length"])
    final_sol_arr = fillSolution(resultant_words, num_arr, [to_include[i]["number"], to_include[i]["dir"]],
final_sol_arr, to_include[i]["length"])
    #print(resultant_words)
#     for j in range(len(final_sol_arr)):
#         print(final_sol_arr[j])
```

```
# print("-----------------------------------------------------------------")

for i in range(len(not_to_include)):
   #resultant_words = fillWords(index_clue_list_2[i], num_arr, [not_to_include[i]["number"],
not_to_include[i]["dir"]], to_be_solved_arr, to_include[i]["length"])
   #to_be_solved_arr = fillSolution(resultant_words, num_arr, [to_include[i]["number"],
to_include[i]["dir"]], to_be_solved_arr, to_include[i]["length"])
   final_sol_arr = fillSolution(index_clue_list_2[i], num_arr, [not_to_include[i]["number"],
not_to_include[i]["dir"]], final_sol_arr, not_to_include[i]["length"])
   # print("")
   # for j in range(len(final_sol_arr)):
   #     print(final_sol_arr[j])



#filling the solution grid with the solution letters
temp = 0
i = 0
j = 0
while temp < 25:
   #indexLetter = 'id="cell-id-' + str(temp)
   #indexLetter = indexLetter + ""
   #si = len(indexLetter)
   #letter = content[content.find(indexLetter) + (si + 8):content.find(indexLetter) + (si + 18)]
   letter = final_sol_arr[i][j]


   if letter == '!':
      lent = int(temp / 5)
      wid = temp % 5
      fillGrey(ans_grid, wid, lent)
      #temp=temp+1
      #continue

   else:
      indexStart = content.find('text-anchor="start"', content.find(indexLetter),
content.find('text-anchor="middle"', content.find(indexLetter)))
      if indexStart != (-1):
         markN(ans_grid, int(temp / 5), (temp % 5), content[content.find("</text>", indexStart) + 7])
      fillWord(ans_grid, int(temp / 5), (temp % 5), letter)
      #letter_arr[i][j] = content[content.find('</text>',content.find('text-anchor="middle"',
content.find(indexLetter)))-1]
```

```
        #to_be_solved_arr[i][j] = '*'
        #final_sol_arr[i][j] = '*'

    temp=temp+1
    j+=1
    if j > 4:
        j = 0
        i+=1


#creating group name and current date and time
description = tk.Canvas(window,height = 105,width=250,highlightbackground='black', bg='white')
description.place(x=175,y=450)
description.create_text(128,20, font=('Arial',15),text='Group Name: YAINDU')
date = datetime.today().strftime('%d-%m-%Y')
description.create_text(107,45, font=('Arial',15),text='Date: '+str(date))

curTime = datetime.now() + timedelta(hours=0)
description.create_text(98,70, font=('Arial',15),text='Time: ' + curTime.strftime("%H:%M:%S"))

#loop to run rin the window
window.mainloop()
```

findWords.py

```
#!pip install PyDictionary
import inflect
p = inflect.engine()
from PyDictionary import PyDictionary
import re
import numpy
import nltk

#nltk.download('stopwords')
from nltk.corpus import stopwords
import string
import requests
import ast
from nltk.tokenize import word_tokenize
#nltk.download('punkt')
import sys
import os

# Disable
def blockPrint():
```

```
    sys.stdout = open(os.devnull, 'w')

# Restore
def enablePrint():
    sys.stdout = sys.__stdout__


dictionary = PyDictionary()
def findWords(stri):
    result = []
    data = re.split(', |_|-|!| ', stri)
    for words in data:
        blockPrint()
        synonym = dictionary.synonym(words)
        antonym = dictionary.antonym(words)
        enablePrint()
        if not antonym and not synonym:
            temp = []
        elif not synonym and antonym:
            temp = antonym
        elif not antonym and synonym:
            temp = synonym
        else:
            temp = numpy.concatenate([synonym, antonym])
        result = numpy.concatenate([result, temp])

    return result

def wikipedia_solution(wikipedia_clues):
    WIKIPEDIA_API =
"https://en.wikipedia.org/w/api.php?action=query&utf8=&format=json&list=search&srlimit=50&srsearc
h="
    stop = stopwords.words('english') + list(string.punctuation)
    #clue_mapping = dict()
    #print("wikipedia fetching word for " + wikipedia_clues)
    if '#' in wikipedia_clues:
        wikipedia_clues = wikipedia_clues.replace("#", "")


    for sentence in wikipedia_clues:
        req = requests.get(WIKIPEDIA_API + sentence)
        wiki_json = ast.literal_eval(req.text)
        #print(wiki_json["query"])
```

```
    solutions = list(set([word for word in [[word for word in word_tokenize(info["title"].lower()) if word
not in stop] for info in wiki_json["query"]["search"]] for word in word]))
  return solutions

def wikipedia_sentence_solution(wikipedia_clues):
  WIKIPEDIA_API =
"https://en.wikipedia.org/w/api.php?action=query&utf8=&format=json&list=search&srlimit=50&srsearc
h="
  stop = stopwords.words('english') + list(string.punctuation)
  #clue_mapping = dict()
  #print(">>> STARTING WIKI FETCH sentence.....")
  if '#' in wikipedia_clues:
    wikipedia_clues = wikipedia_clues.replace("#", "")
  req = requests.get(WIKIPEDIA_API + wikipedia_clues)
  wiki_json = ast.literal_eval(req.text)
  solutions = list(set([word for word in [[word for word in word_tokenize(info["title"].lower()) if word
not in stop] for info in wiki_json["query"]["search"]] for word in word]))
  return solutions


def possible_answers(clues, length):
  myarr = wikipedia_solution(clues)
  myarr = numpy.array(myarr)
  myarr = numpy.concatenate([myarr, findWords(clues)])
  myarr = numpy.concatenate([myarr, wikipedia_sentence_solution(clues)])
  finalResult = []
  returnResult = []
  for i in myarr:
  #if len(i) <= length:
    if i not in finalResult:
     finalResult.append(i.upper())
     if p.plural(i.upper()):
       finalResult.append(p.plural(i.upper()))

     if p.singular_noun(i.upper()):
       finalResult.append(p.singular_noun(i.upper()))

  for j in finalResult:
    #sorting based on length
    if len(j) == length:
      if j not in returnResult:
        returnResult.append(j)

  return returnResult
```

fillWords.py

```
def fillWords(results_array, sample_array2, list_sample, notfound_array, word_size):
    index = [-1, -1]
    for i in range(0, len(sample_array2)):
        try:
            index[1] = sample_array2[i].index(list_sample[0])
            #print("test", i)
            #print(sample_array2[i].index(list_sample[0]))
            index[0] = i
            #print(index[0])
        except:
            pass

    #print(index[0],index[1])

    if(list_sample[1] == "across"):
        word_index = -1
        for i in range(index[1], word_size + index[1]):
            word_index += 1
            if notfound_array[index[0]][i] != '*':
                temp_result = []
                for x in range(len(results_array)):
                    #print(results_array[x][word_index], notfound_array[index[0]][i], word_index)
                    if results_array[x][word_index] == notfound_array[index[0]][i]:
                        temp_result.append(results_array[x])
                results_array = temp_result

    else:
        word_index = -1
        #down the table, R X C
        for i in range(index[0], word_size + index[0]):
            word_index += 1
            if notfound_array[i][index[1]] != '*':
                temp_result = []
                for x in range(len(results_array)):
                    #print(results_array[x][word_index], notfound_array[i][index[1]], word_index)
                    if results_array[x][word_index] == notfound_array[i][index[1]]:
                        temp_result.append(results_array[x])
                results_array = temp_result

    return results_array

def fillSolution(result, sample_array2, list_sample, notfound_array, word_size):
```

```
index = [-1, -1]
for i in range(0, len(sample_array2)):
    try:
        index[1] = sample_array2[i].index(list_sample[0])
        index[0] = i
    except:
        pass

if(list_sample[1] == "across"):
    word_index = -1
    for i in range(index[1], word_size + index[1]):
        word_index += 1
        if notfound_array[index[0]][i] == '*':
            notfound_array[index[0]][i] = result[0][word_index]

else:
    word_index = -1
    #down the table, R X C
    for i in range(index[0], word_size + index[0]):
        word_index += 1
        #print(notfound_array[i][index[1]], result[0][word_index], word_index)
        if notfound_array[i][index[1]] == '*':
            notfound_array[i][index[1]] = result[0][word_index]

return notfound_array
```

References

[1] Pwtempuser, “Wikipedia,” ProgrammableWeb, 05-Jun-2020. [Online]. Available: https://www.programmableweb.com/api/wikipedia. [Accessed: 24-Dec-2020].

[2] “PyDictionary,” PyPI. [Online]. Available: https://pypi.org/project/PyDictionary/. [Accessed: 24-Dec-2020].

[3] Thanasuan, Kejkaew, and Shane T. Mueller. “Crossword Expertise as Recognitional Decision Making: An Artificial Intelligence Approach.” Frontiers in Psychology, vol. 5, 11 Sept. 2014, 10.3389/fpsyg.2014.01018.

[4] Roberto, Basili, and Alessandro Lenci. "The First Italian Conference on Computational Linguistics CLiC-it 2004." Pisa University Press, vol. 1, pp. 44-45, 10 Dec. 2014.

[5] Shazeer, Noam M., Michael L. Littman, and Greg A. Keim. "Solving Crossword Puzzles as Probabilistic Constraint Satisfaction." AAAI/IAAI. 2 August. 1999.

[6] Barlacchi, Gianni, Massimo Nicosia, and Alessandro Moschitti. "Learning to rank answer candidates for automatic resolution of crossword puzzles." Proceedings of the Eighteenth Conference on Computational Natural Language Learning, pp. 39-48. June. 2014.

[7] A. Thomas and S. S., “Towards a Semantic Approach for Candidate Answer Generation in Solving Crossword Puzzles,” *Procedia Computer Science*, 04-Jun-2020. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1877050920312424. [Accessed: 24-Dec-2020].

This project reports work done in partial fulfillment of the requirements for CS 461 -- Artificial Intelligence. The software is, to a large extent, original (with borrowed code clearly identified) and was written solely by members of <YAINDU>.

Word Count: 2211 words