Fall 2020
CS 461 - Artificial Intelligence

Term Project Report

Group: LUMOS

Rahmiye Büşra Büyükgebiz
Ozan Aydın
Hüseyin Ata Atasoy
Mehmet Ali Altunsoy

Table of Contents

1. Introduction

This report takes a look at the final program which solves The New York Times 5x5 Mini Crossword Puzzle with the help of Artificial Intelligence learning. The New York Times Mini Crossword Puzzle is created by Joel Fagliano and updated daily. Every puzzle consists of 5 down and 5 across clues, except Saturday's puzzle, which is not in the scope of this program. The goal of this program is to solve the crossword using the given clues on The New York Times Mini Crossword webpage, while keeping the geometry and the solutions the same as the puzzle given on the The New York Times webpage.

2. Methods

As stated earlier, the goal of this project was to come up with answers for the given puzzle using its clues. In order to find methods and strategies to reach the goal, we consulted the lecture notes provided by Prof. Akman and researched other possible ways.

2.1. Finding Domains

The first thing we did was to utilize web scraping to create domains for every clue. In order to do this, we relied on Wikipedia, Merriam-Webster and Datamuse. Once we gathered enough words for a clue, we filter the initial domain in terms of word size and various other factors which will be explained later.

2.2. Solving the Puzzle

In order to solve the puzzle, we first use constraint filtering to eliminate the unnecessary words from the domains.The constraints used for this are found by examining the grid and finding the locational information, so the constraints tell us the shared letters for each domain. After applying the constraints, we gather the final domains. Usually, after the elimination, some of the domains only have one word left inside them, and we see a considerable reduction of words in others. After this, we assume that the domains that have only one word left inside them are the correct domains and place them on the grid, and solve the grid for the other domains.

3. Implementation Details

3.1. Web Scraping

To initialize the domains, we use web scraping, which is already commonly used in other crossword applications as well. While talking about their own crossword solver, Ernandes states that “The main innovation consists of having shown, for the first time, the effectiveness of searching the Web to crack cross- words.” (Angelini et al. 1417). In order to gather words, we relied on three resources which are the Datamuse API, Merriam Webster dictionary and Wikipedia. At first, we thought Google might be another resource for scraping, but when we

searched some of the clues on Google manually, we observed that there were many crossword solver websites in the search results. Therefore, we decided not to use the words gathered from Google search. In the Figure 1, you can see the search results for the clue “Show strength in poker”.

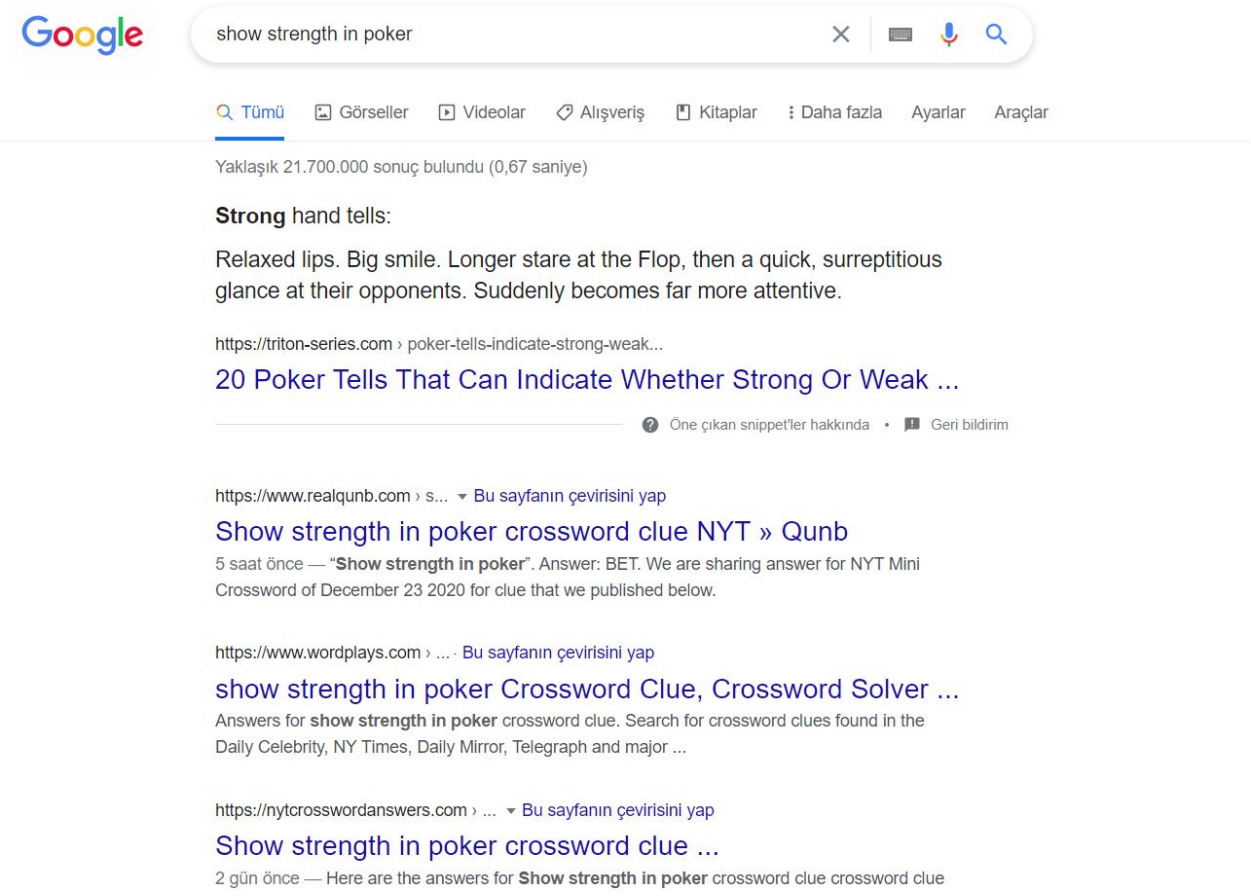

Figure 1: Google search results for the clue “Show strength in poker”

After testing these resources, it became clear that, searching the clues without any processing, resulted in bloated and inaccurate domains which were impossible to work with, therefore we implemented processing methods to improve the accuracy and shrink the size of the domains. We stripped each clue from its stop words and punctuations and created token pairs to be searched, in addition to each token of the clue (see Appendix A). For example, in the beginning for the clue “Subject of the famous photo “The Blue Marble”, we conducted a search for each element in the list [“subject”, “of”, “the”, “famous”, “photo”, “the”, “blue” “marble”]. The result of this search did not contain the answer “Earth”. With our clue processing, our search list became [“subject”, “famous”, “photo”, “blue”, “marble”, “subject famous”, “famous photo”, “blue marble”], which was sufficient to find and include “Earth” in the clue domain.

```
Wikipedia search for: ___ Arabia
Synonym search from Datamuse and Merriam-Webster for: ___ Arabia
Search list for "___ Arabia": ['arabia']
Wikipedia search for: Gentle prod
Synonym search from Datamuse and Merriam-Webster for: Gentle prod
Search list for "Gentle prod": ['gentle', 'prod', 'gentle prod']
Wikipedia search for: Rodentia or Carnivora
Synonym search from Datamuse and Merriam-Webster for: Rodentia or Carnivora
Search list for "Rodentia or Carnivora": ['rodentia', 'carnivora', 'rodentia carnivora']
Wikipedia search for: "No freaking ___!"
Synonym search from Datamuse and Merriam-Webster for: "No freaking ___!"
Search list for ""No freaking ___!"": ['freaking']
Wikipedia search for: Show strength in poker
Synonym search from Datamuse and Merriam-Webster for: Show strength in poker
Search list for "Show strength in poker": ['show', 'strength', 'poker', 'show strength', 'strength poker']
Wikipedia search for: The "white" in "White Christmas"
Synonym search from Datamuse and Merriam-Webster for: The "white" in "White Christmas"
Search list for "The "white" in "White Christmas"": ['white', 'white', 'christmas', 'white white', 'white
christmas']
Wikipedia search for: Mystical glow
Synonym search from Datamuse and Merriam-Webster for: Mystical glow
Search list for "Mystical glow": ['mystical', 'glow', 'mystical glow']
Wikipedia search for: The elf in "Elf"
Synonym search from Datamuse and Merriam-Webster for: The elf in "Elf"
Search list for "The elf in "Elf"": ['elf', 'elf', 'elf elf']
Wikipedia search for: Precipice
Synonym search from Datamuse and Merriam-Webster for: Precipice
Search list for "Precipice": ['precipice']
Wikipedia search for: Layer of a wedding cake
Synonym search from Datamuse and Merriam-Webster for: Layer of a wedding cake
Search list for "Layer of a wedding cake": ['layer', 'wedding', 'cake', 'layer wedding', 'wedding cake']
```

Figure 2: Console output from 23.12.2020, displaying the search list for each clue

We realized that by tokenizing words like this, even though we got the correct results more often, our domains were still too big, so we limited the number of words we can gather from each website. For example, we only got the first “summary” paragraph from the Wikipedia articles, as through trial and error we realized that most often the word that we were looking for was located there. In order to do this, we used Wikipedia API which finds relevant pages for the search query (see Appendix B). For each query, we access only the first page in the search results, thus limiting the size of the domains.

We used Merriam-Webster and Datamuse API to get synonyms and related words (see Appendix C). Datamuse is an API that collects semantic information from various websites that provide this information, including Wordnet. There are multiple services available in Datamuse, but only the semantic relations API is used, in which it returns various words that have a semantic similarity with the word that we have input.

3.2. Solving the Puzzle

3.2.1. Data Structures

Word Class

Word class has the following attributes:

- Word: String of the answer word
- Type: Whether the answer is “across” type or “down” type.
- rowColIndex: Index number of the column if it is a “down” type, else index of the row.
- clueIndex: Actual clue number from the NY Times Puzzle.
- Active: Whether the word is still in the domain or not.

Data Structure for Storing Cells

In order to reduce the complexity of the solver algorithm, a combination of Python List and Dictionary structures are used to store cells which is an attribute of solver class. Aim of this structure is to reach the words which contain the letters at the corresponding index while looping through cells. Hence the structure of the cells are designed as a 5x5 2-D list, which contains a dictionary structure with “across” and “down” keys. Each key value is another dictionary structure that the letters of the english alphabet are the keys. The values of the keys are lists that contain Word objects mentioned in the Word Class section. The storing of this variable is done with setCells() function before the solve function is called and it is done only once (see Appendix D).

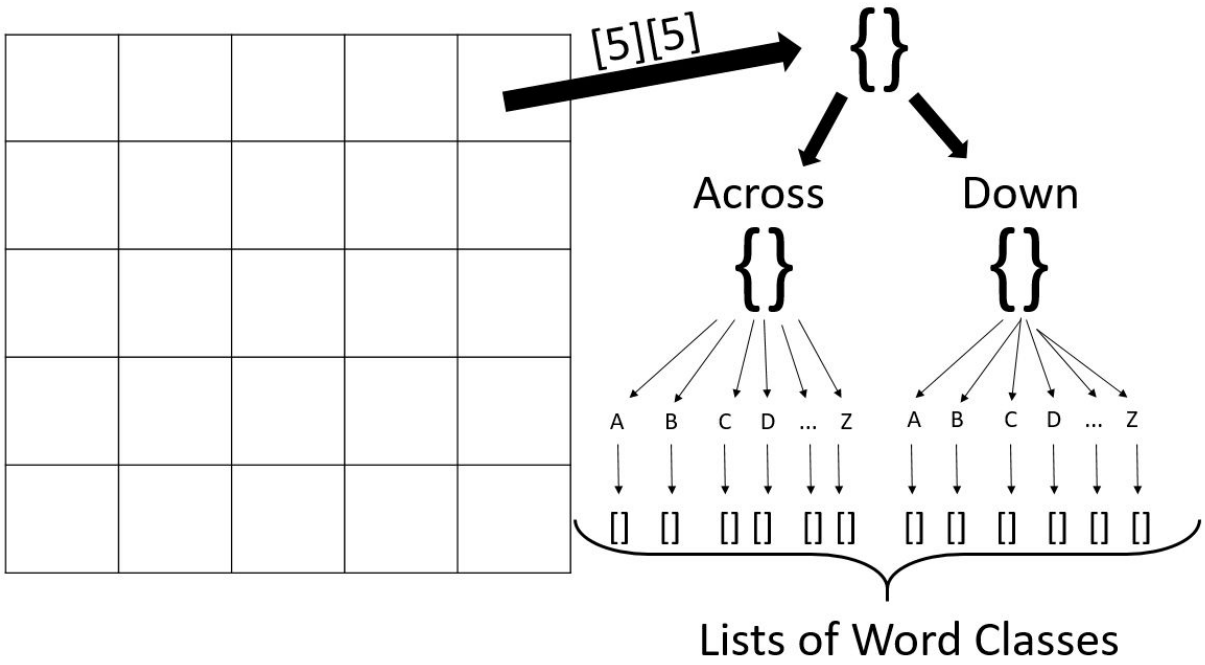

Figure 3: Visualisation of data structure that stores cells.
{ } : Python Dictionary [ ] : Python List

Example usage of the self.cells attribute:
self.cells[4][2][“across”][“k”]
This will return a list of across words that has a letter K in the given cells. The significance of this is mainly the runtime of the algorithm, as accessing the words like this instead of searching for the whole domain array every single time saves us a lot of time.

3.3. Domain Initialization

Aim of the domain initialization is filtering the words coming from the web scraping and initializing answer domains for corresponding clues. Size of the words that gathered from web scraping for each clue can be seen from Figure 3.

```
Across Clues
Size of the initial domain for the across clue 1 : 1809
Size of the initial domain for the across clue 5 : 1474
Size of the initial domain for the across clue 6 : 791
Size of the initial domain for the across clue 7 : 1100
Size of the initial domain for the across clue 8 : 2234
Down Clues
Size of the initial domain for the down clue 6 : 1040
Size of the initial domain for the down clue 1 : 2374
Size of the initial domain for the down clue 2 : 1042
Size of the initial domain for the down clue 3 : 487
Size of the initial domain for the down clue 4 : 3353
```

Figure 4: Size of the words that gathered from web scraping for each clue

The first step of domain initialization is calculating the answer's cell location in the puzzle grid and word lengths and storing them as Python dictionaries.

The second step is filtering the words from web scraping with respect to following criteria:

- Does the length of the word match with the corresponding answer’s word length.
- All the characters of the world are subsets of the letters of English Alphabet.
- Does the length of combination of two side by side words match with the corresponding answer’s word length.
- Does the word’s length is matched with the corresponding answer’s word length after removing the “s” plural suffix, if any.

The words that satisfy these criteria are added to corresponding answer domains of the related row or column without duplication as a Word Object explained in 3.2.1. It should be noted that combining of the words is done because an answer for a clue can sometimes be two words written as one word.

```
Across Clues
Size of the filtered domain for the across clue 1 :30
Size of the filtered domain for the across clue 5 :20
Size of the filtered domain for the across clue 6 :29
Size of the filtered domain for the across clue 7 :29
Size of the filtered domain for the across clue 8 :54

Down Clues
Size of the filtered domain for the down clue 6 :24
Size of the filtered domain for the down clue 1 :21
Size of the filtered domain for the down clue 2 :46
Size of the filtered domain for the down clue 3 :15
Size of the filtered domain for the down clue 4 :2
```

Figure 5: Size of the words that gathered from web scraping for each clue

After applying mentioned filtering procedures, the size of each domain has shrunk considerably which can be seen from the console output screenshot in Figure 4. In addition, the words in initial filtered domains are given in Figure 5 in order to compare the domains before and after domain reduction using constraints which will be explained in section 3.4.

```
==================
INITIAL FILTERED DOMAINS
==================
Across
1:  AREA ALSO ASHY BELL BING CANE CLAU CLAD DULL EGGS FILM GOOD GIFT GREY
GRAY JESU KAYE LILY MILD NOEL PALE PURE SONG SNOW SAFE THAT THIS USED XMAS YULE

5:  AURA BAKE BEAT BEAM BURN COOK CHAR DEEP DICE DARK FAZE FIRE GLOS GALO
HILL HALO LAMP MELT OMEN YOGI

6:  AFTER CLAUS COMES CAROL DWARF ELVES ELFIN ELVEN FAERY FAIRY GHOUL GNOME
ISFEB KNOWS NIXIE NORTH OUPHE PITTS PIXIE RHYME SHELF SMALL SCOUT SANTA STORY
 TELLS TROLL UNTIL WHICH

7:  ARID ALSO AREA ABYS BASE CRAG DOWN EDGE FACE FORM GULF HIGH HAVE LEAD
LINE MOST MANY PATH ROCK SUCH SOME SCAR SOIL THAT TYPE THEY TALU WITH

8:  AMAS BANK BAND COAT COST CLAS CLAN CASE CAKE CROW DOES DAUB DRES DUST
DUCK EVEN FACE FOOD FOLK FAKE GOWN GOOD HEAP HELD KNOT LUCK LOOP LUMP MASS
MORE MADE MEAL MEAN ONLY PILE PUFF PART PEEL PLAY RANK REAL RARE ROSE RACE
RIME SOUP SNAP SEAM SOME SAME THIS TUCK TIER THEY TIME VEIL WILL WERE WITH WIFE


Down
6:  ALL ACE ANY AIR ARE AND ACT BOB BET END FOR HAS INH JAB LAY NOT OFF ONE
OUT POT SEE TWO THE WHO WIN

1:  ARABS ANYON CZECH COAST HEJAZ JIZAN LUCIA NORTH OTHER OCEAN PARTS PLATE
PLAYS QATAR REDIN SOUTH STRIP SAUDI WHICH WORLD YEMEN YEARS

2:  ATTHE ASLOW ALLAY ACTOR BLOOD BLAND BALMY CLEAR CHUCK CARES CATER DRILL
DRIVE EGGON GRAND GREAT HUMOR HURRY IMPEL KNOCK KNEAD LOFTY LIGHT MUTED MOGUL
 NUDGE NOBLE NABOB PEEVE PRICK PUNCH PRESS QUIET ROYAL REGAL SUNNY SPOON STILL
 STICK SLEEK SHOVE SPOIL SLICK SWELL TONED TAMED URGES

3:  ALTAR BASED BALES CANIS CODES CANID EVERY FOUND GLIRE LUPUS LISTS MOUSE
ORDER PONGO STATE

4:  BUM CUZ
```

Figure 6: Console output for the puzzle dated 23.12.2020, initial domains after filtering

3.4. Domain Reduction with Constraints

The constraints that we use is explained by Jimbo et.al as: “If two words cross each other, they should have the same character at the crossing point. Violation of this constraint means that at least one of the two words is incorrect.” (Jimbo et al. 40). Since there could be other words that fit the puzzle grid which are not the correct solution, the term “possible solution” is used and they will be checked in section 4.3.

Constraints are used in order to reduce the size of the domains. Reducing the size is crucial as the initial domains contain 200 words on average per domain. Trying this many words to solve the puzzle is extremely time consuming and cannot be done in a reasonable amount of time. Function solve() iterates through cells in the puzzle and checks whether all the letters that may be placed to the cell both exist in the corresponding index of across and down word domains of

that row and column, or not (see Appendix E). For example, if a constraint states that the first letters of across_1 domain and down_1 domain should be the same, we check the first letters of all words in both domains and eliminate the words that start with a letter which is not present in both of the domains. If the letter does not exist in the both domains no changes needed. If it only exists in one domain, the words containing that letter at corresponding index are removed from the domain. Iterating through cells is continued in a while loop until there are no changes made in the loop. This means the domains are in the final form and they could not be reduced more.

Generally, the worst case while reducing the domains is the case where we have multiple missing correct answers in our domains.When this happens, when we use constraint reduction on the domains, since the correct words are not in the domains, all of the words get filtered out and the domain becomes empty. Through trial and error, it became clear that when this happens, we get no results. In order to counter this, we first tried to add more words to our domains so that the probability of them becoming empty will be lower, but after some reading we decided this was incorrect, as stated by Shazeer et al., "... we might be tempted to over-generate our candidate list. Of course, this has the new shortcoming that spurious solutions will result." (Shazeer et al.).

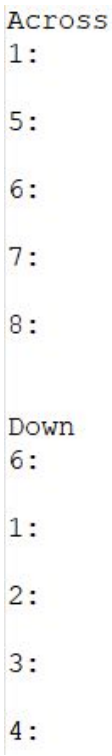

Figure 7: Console output of empty domains when we encounter the worst case

To cope with this case, the algorithm neglects some clues. Which means the constraints of the cells corresponding to the neglected clues do not apply. If we could not find two words and because of this our domains reduce to size 0, we apply this neglection algorithm so that we can avoid this and at least let some words remain in our domains. In each iteration another

combination is tried. This means 1024 (by 2^{10}) iterations of the while loop mentioned above. Thus, the complexity of the algorithm in the while loop is very significant. For example, if we look at the result from Figure X, it appears we could not find both 'WAY' and 'BUDDY' from our web scraping, and since they do share a constraint, when we use constraint reduction to reduce our domains, both of the domains would become empty, resulting in an empty solution set. In order to counter this, our algorithm starts neglecting row and column constraints in order not to over filter and leave a domain empty in case of us not finding multiple correct words. As seen in the below figure, when we neglect row 3 and column 5 constraints, it skips over that particular constraint and as a result, our domains can be filtered without being emptied. We still do not have 'WAY' and 'BUDDY' in our domain set, but our aim is to be able to keep our domains populated and at least come up with a solution, albeit not the correct one.

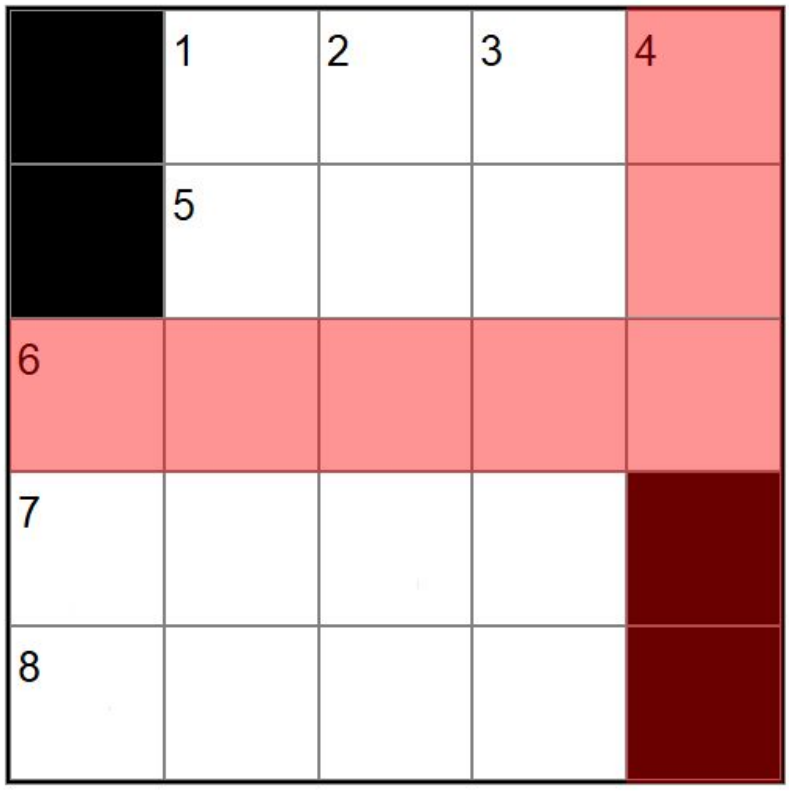

Figure 8: Visual representation of our neglection algorithm

3.4.1. Processing Final Domains

After gathering the final domains, our aim basically becomes to "find the best way of choosing a candidate for each clue, while respecting the constraints of the grid." (Keim et al.). After reducing domains for each iteration, isItTheBestSolution function is called (see Appendix F). This function counts the number of domains with size 1, which the possible answer for that clue is reduced to a single option. Afterward if the number of domains with size 1 are more than 5 (5 is selected as a border since the puzzles with at least 5 answers are sufficient enough),these domains are copied to the list of final solutions.

```
====================
FILTERING DOMAINS BY APPLYING CONSTRAINTS
====================
====================
BEST DOMAINS FROM FILTERING
====================
Across
1:  SNOW

5:  AURA

6:  AFTER CLAUS COMES CAROL DWARF ELVES ELFIN ELVEN FAERY FAIRY GHOUL
GNOME ISFEB KNOWS NIXIE NORTH OUPHE PITTS PIXIE RHYME SHELF SMALL
SCOUT SANTA STORY TELLS TROLL UNTIL
WHICH

7:  EDGE

8:  TIER


Down
6:  BET

1:  SAUDI

2:  NUDGE

3:  ORDER

4:  BUM CUZ
```

Figure 9: Console output for the puzzle dated 23.12.2020, final domains after filtering

3.4.2. Scoring Possible Solutions

Since the size of each domain would not shrink down to a single word, there was a possibility of the program coming up with answers that vastly differ from the ones provided on the NY Times website, but still fit the constraints of the problem. Therefore to obtain the best possible solution to the puzzle, we came up with a scoring algorithm. For each answer placed on the grid, we searched for its corresponding clue on Datamuse using Datamuse API. We used the scoring system provided by the Datamuse, as Datamuse also returns a score for each word, highest score being the most relevant word to our search.They use The Google Books Ngrams dataset and word2vec method to build the language model that scores candidate words by context, and also for some of the lexical relations. Scores of the relevant words of an example clue, “show approval”, can be seen from the figure below. As can be seen, “applaud” and “clap” have higher scores than “nod” and “ovation”.

```
[{"word":"applaud","score":49816,"tags":["syn","v"]},
{"word":"clap","score":49416,"tags":["syn","n","v"]},
{"word":"nod","score":48616,"tags":["syn","n","v"]},
{"word":"ovation","score":48100,"tags":["syn","n"]}]
```


Figure 10: Relevant words of "show approval" clue returned by Datamuse API

Initially, each possible grid has 0 score. If an answer has semantic relations with its corresponding clue, we add its corresponding score on Datamuse to the score of the grid. Sometimes a correct answer might not be in the relevant words list that returns from Datamuse. However, we thought this should not be an important problem since other grids with the incorrect answer are most likely to not be in that list also. We used these scores to compare the intermediate solution grids, based on the sum of the relevancy scores of each word placed on the grid (see Appendix G). Finally, we return the grid with the highest score as our final solution.

```
Calculating score for a possible grid:
['-', 'S', 'N', 'O', 'Y']
['-', 'A', 'U', 'R', 'A']
['B', 'U', 'D', 'D', '']
['E', 'D', 'G', 'E', '-']
['T', 'I', 'E', 'R', '-']

Score:  282833
Calculating score for a possible grid:
['-', 'W', 'C', 'A', 'Y']
['-', 'H', 'A', 'L', 'O']
['P', 'I', 'T', 'T', '']
['O', 'C', 'E', 'A', '-']
['T', 'H', 'R', 'R', '-']

Score:  10496
==================
SOLUTION
==================
['-', 'S', 'N', 'O', 'W']
['-', 'A', 'U', 'R', 'A']
['B', 'U', 'D', 'D', '']
['E', 'D', 'G', 'E', '-']
['T', 'I', 'E', 'R', '-']
```

Figure 11: Console output for the puzzle dated 23.12.2020, the solution is found after scoring the grids

4. Results

In this section, some of the previous results that we have obtained will be shown.

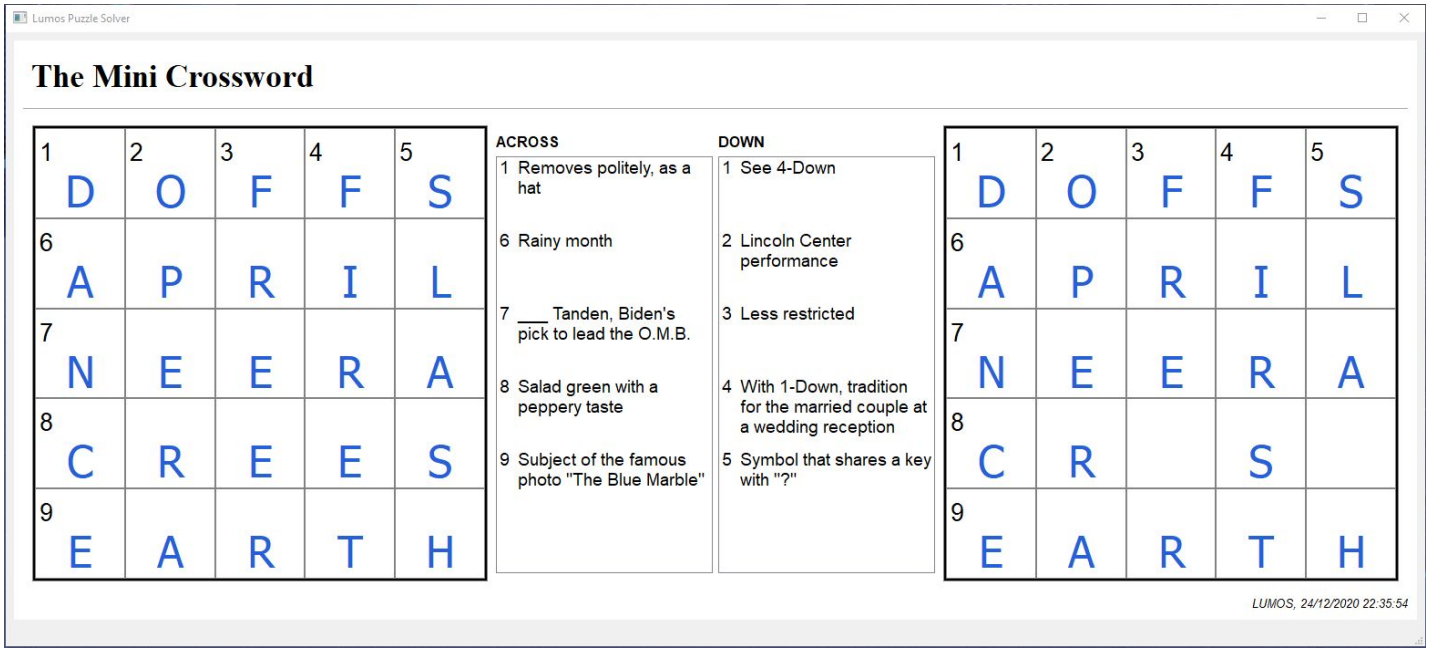

Figure 12: Screenshot for the puzzle dated 16.12.2020

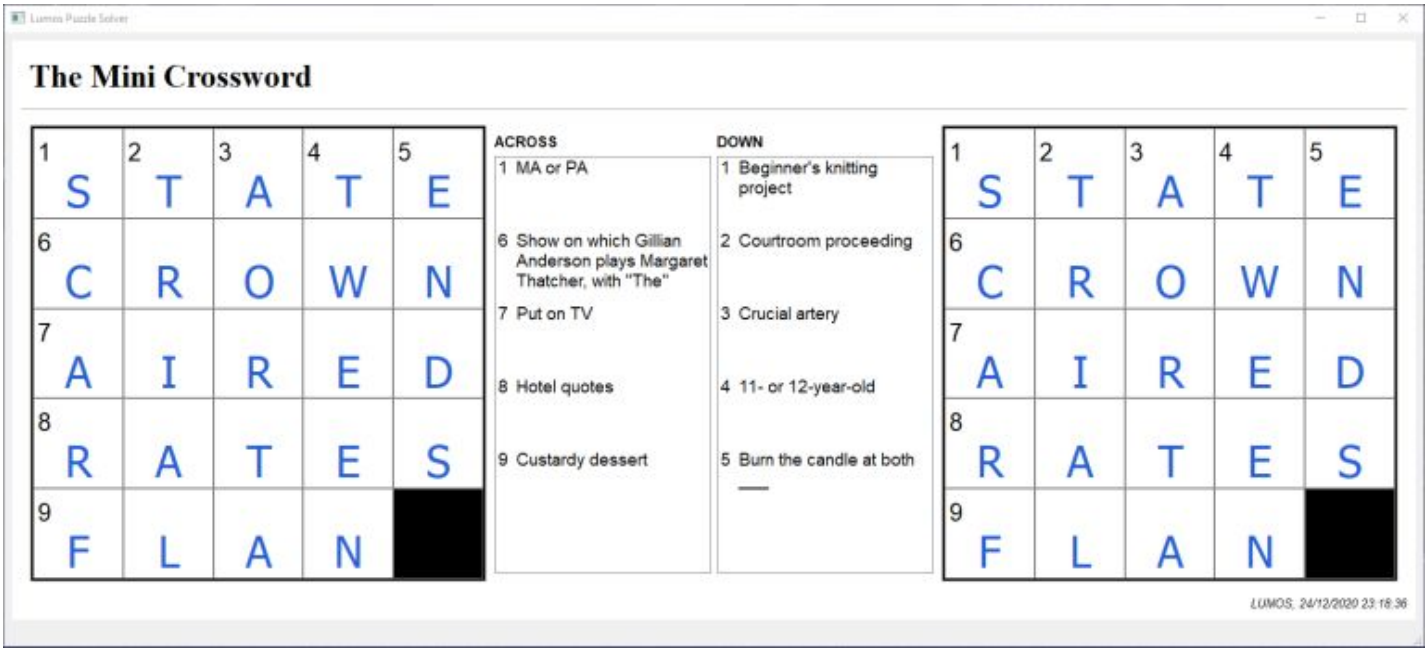

Figure 13: Screenshot for the puzzle dated 24.11.2020

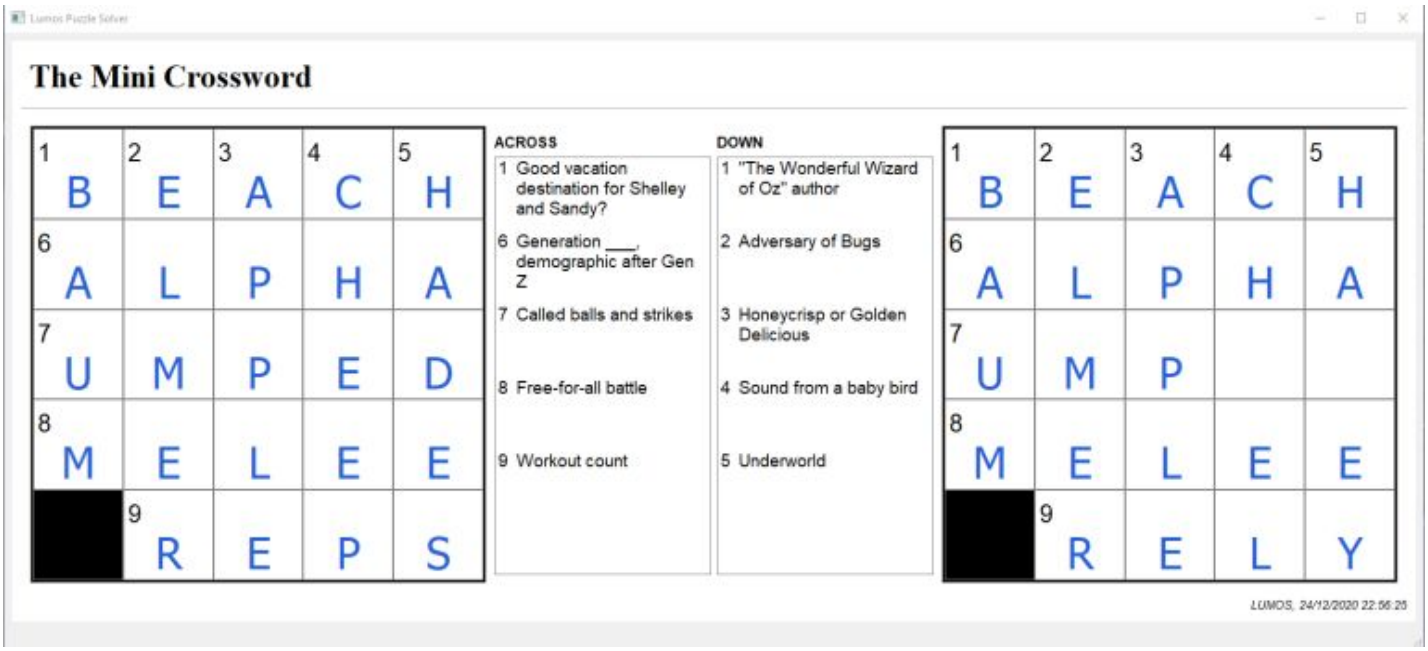

Figure 14: Screenshot for the puzzle dated 1.12.2020

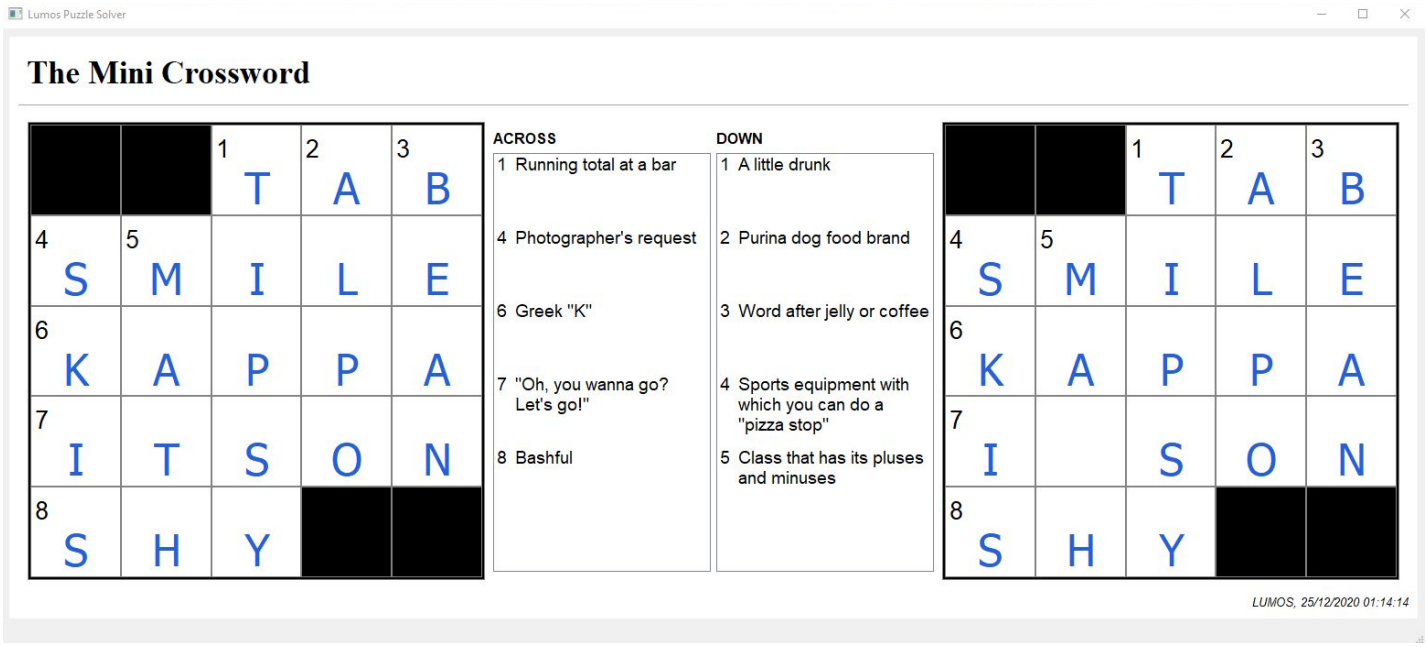

Figure 15: Screenshot for the puzzle dated 3.12.2020

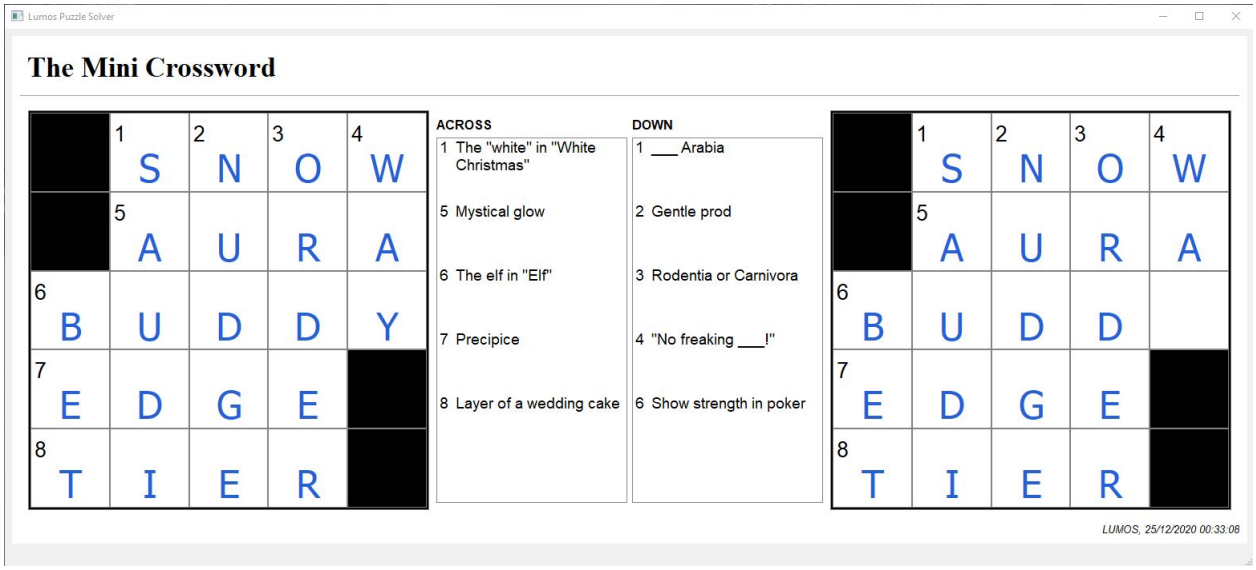

Figure 16: Screenshot for the puzzle dated 23.12.2020

5. Conclusion

The aim of this program is to solve the 5x5 New York Times puzzle by Joel Fagliano. In order to achieve this, we constructed a program that creates domains for each clue by searching the Internet, reduces these domains using constraint reduction, and then uses the final,smaller domains to come up with a small amount of alternative solutions. We then assign points to these solutions by semantically comparing each word on the puzzle grid with its clue and the puzzle with the highest score becomes the solution to our puzzle. While designing our algorithm, we drew on a lot of artificial intelligence concepts such as creating initial words pools, constraint reduction and heuristics.

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 LUMOS.

Word Count: 2497

6. References

Ernandes, Marco, Giovanni Angelini, and Marco Gori. "Webcrow: A web-based system for crossword solving." *AAAI*. 2005, https://www.aaai.org/Papers/AAAI/2005/AAAI05-224.pdf.

Ginsberg, Matthew L. "Dr. fill: Crosswords and an implemented solver for singly weighted csps." *Journal of Artificial Intelligence Research* 42 (2011): 851-886. https://www.jair.org/index.php/jair/article/view/10741/25656

Jimbo, Kazuki, et al. “Solving Crossword Puzzles Using Extended Potts Model.” *New Frontiers in Artificial Intelligence Lecture Notes in Computer Science*, 2009, pp. 39–47., doi:10.1007/978-3-642-00609-8_5.

Keim, Greg A., et al. “Proverb: the Probabilistic Cruciverbalist.” *AAAI-99 Proceedings*, 1 July 1999, dl.acm.org/doi/10.5555/315149.315425.

Shazeer, Noam M., Michael L. Littman, and Greg A. Keim. "Solving Crossword Puzzles as Probabilistic Constraint Satisfaction." *AAAI/IAAI*. 1999, https://www.aaai.org/Papers/AAAI/1999/AAAI99-023.pdf.

Appendices

Appendix A: Source Code of Tokenizing Clues

```
import nltk
import string
from nltk.corpus import stopwords

stop_words = set(stopwords.words('english'))

# Processes the clue to generate token pairs to be later used in scraping
# Returns all the possible pairs of tokens for a clue
def combine_tokens(clue, acrossClues, downClues):
    # Checks if the clue is in the form of "See X-Down" or "See X-Across"
    if "See" in clue:
        clue = getPointedClue(clue, acrossClues, downClues)

    clue = clue.lower()
    #clue = clue.replace("-", " ")
    clue_without_punctuation = clue.translate(
        str.maketrans('', '', string.punctuation))
    tokens = nltk.word_tokenize(clue_without_punctuation)
    tokens = [w for w in tokens if not w in stop_words]
    number_of_tokens = len(tokens)
    for i in range(number_of_tokens):
        if i != number_of_tokens - 1:
            tokens.append(tokens[i] + " " + tokens[i + 1])
    # print(tokens)
    return tokens

# Used to find the clues that are in the form of "See X-Down" or "See X-Across"
def getPointedClue(clue, acrossClues, downClues):
    pointedClue = ""
    if "Down" in clue:
        pointedClue = downClues[clue[4]]
    else:
        pointedClue = acrossClues[clue[4]]
    return pointedClue
```

Appendix B: Source Code of Wikipedia Search

```
def searchWikipedia(clue):
    """
    This function takes a clue and searches for the clue in wikipedia.
    Takes the first result found in wikipedia and considers only the summary
    of that page.
    Returns a set containing words it found on a wikipedia page.
    """
    text = set()
    clue = clue.translate(str.maketrans('', '', string.punctuation))
    search_results = wikipedia.search(clue, results=1)
    for result in search_results:
        try:
            content = wikipedia.page(result + ".").summary.translate(str.maketrans('',
'', string.punctuation))
            for w in content.split():
                text.add(w.lower())
        except:
            continue

    return text
```

Appendix C: Source Code of Synonym Search

```
def searchSynonyms(clue, acrossClues, downClues):
    """
    This function takes a clue, list of across clues and list of down clues.
    It tokenizes the clue first to subwords and searches for each word on Merriam
Webster and Datamuse.
    """
    print('Search list for "' + clue + '":', end=' ')
    tokens = combine_tokens(clue, acrossClues, downClues)
    print(tokens)
    words = set()
    for word in tokens:
        if(contains_multiple_words(word)):
            # Merriam webster
            r_mw = requests.get("https://www.merriam-webster.com/thesaurus/" +
word.split()[0] + "%20" + word.split()[1])

            # Datamuse
            r_dm =
requests.get('https://api.datamuse.com/words?rel_syn={}'.format(word))
            r_dm2 = requests.get('https://api.datamuse.com/words?ml={}'.format(word))

        else:
            # Merriam webster
            r_mw = requests.get("https://www.merriam-webster.com/thesaurus/" + word)

            # Datamuse
            r_dm =
requests.get('https://api.datamuse.com/words?rel_syn={}'.format(word))
            r_dm2 = requests.get('https://api.datamuse.com/words?ml={}'.format(word))

        soup = BeautifulSoup(r_mw.text, 'lxml')

        # Search for synonym list on Merriam Webster
        for i in soup.select('.thes-list.syn-list a'):
            words.add(str(i.string).lower())

        # Search for related list on Merriam Webster
        for i in soup.select('.thes-list.rel-list a'):
            words.add(str(i.string).lower())

         # Search for similarity list on Merriam Webster
        for i in soup.select('.thes-list.sim-list a'):
            words.add(str(i.string).lower())

        count = 0
```

```
        results = r_dm.json()
        for result in results:
            words.add(result['word'])
            count += 1
            if count > 20: # 20 is an enough value to get synonyms from one website
                break

        results = r_dm2.json()
        count = 0
        for result in results:
            words.add(result['word'])
            count += 1
            if count > 20:
                break
    return words
```

Appendix D: Source Code of Cell Setting

```
def setCells(self):
        """This function stores the possible letters in each cell in order to fasten
the constraint reduction process
        """
        for row in range(0, 5):
            for col in range(0, 5):
                for letter in string.ascii_uppercase:
                    self.cells[row][col]["across"][letter] = []
                    self.cells[row][col]["down"][letter] = []

        for row in range(0, 5):
            for col in range(0, 5):
                for letter in string.ascii_uppercase:
                    if self.getTheRelatedDomainOfThisCell(row, col, "") != {}:
                        for word in self.getTheRelatedDomainOfThisCell(row, col,
"")["across"]["domain"]:
                            if letter ==
word.word[self.getTheRelatedDomainOfThisCell(row, col, "")["across"]["index"]]:
                                self.cells[row][col]["across"][letter].append(
                                    word)
                        for word in self.getTheRelatedDomainOfThisCell(row, col,
"")["down"]["domain"]:
                            if letter ==
word.word[self.getTheRelatedDomainOfThisCell(row, col, "")["down"]["index"]]:
                                self.cells[row][col]["down"][letter].append(
                                    word)
```

Appendix E: Source Code of Puzzle Solver

```
def solver(self):
        """This function reduces the domains with constraint check. After the domains
are reduced in size,the solver tries combinations of domains in order to find
combinations of words that can fit the puzzle while still holding the constraints.
Since some answers may not be available to us in the domains, it neglects an answer
and its constraints and searches for a possible solution again. Then the outcome will
be sent to the validators and if it is solved enough the best solutions are kept."""
        puzzleNotSolved = True
        # while puzzle is solved try more jokers
        while puzzleNotSolved:
            puzzleNotSolved = False
            self.reset()
            if self.changeNeglected():
                changeMade = True
                # Constraints checks
                while changeMade:
                    changeMade = False
                    for row in range(0, 5):
                        # --------------------
                        if row not in self.neglectedWords["row"]:
                            for col in range(0, 5):
                                # --------------------
                                if col not in self.neglectedWords["col"]:
                                    for letter in string.ascii_uppercase:
                                        # check whether the letters in the cells are
matched. If not, reduce the domain.
                                        if self.getTrueFalse(row, col, "across",
letter) and not self.getTrueFalse(row, col, "down", letter):
                                            for word in
self.cells[row][col]["across"][letter]:
                                                word.active = False
                                            changeMade = True

                                        elif not self.getTrueFalse(row, col, "across",
letter) and self.getTrueFalse(row, col, "down", letter):
                                            for word in
self.cells[row][col]["down"][letter]:
                                                word.active = False
                                            changeMade = True
                self.isItTheBestSolution()  # test for best solution

                # if the puzzle is solved. Stop loop
                puzzleNotSolved = not self.isPuzzleSolved()
            else:
                return False
```

Appendix F: Source Code of Finding Best Solutions

```
def isItTheBestSolution(self):
        """This function is looking for the best solution.
        If a domain of the clue is reduced to a single word, it recieves +1 score
        the most scored solution is determined as the best solution
        """
        count = 0
        for row in range(0, 5):
            wordCount = 0
            for word in self.domains["across"][row]:
                if word.active:
                    wordCount += 1
            if wordCount == 1:
                count += 1
        for col in range(0, 5):
            wordCount = 0
            for word in self.domains["down"][col]:
                if word.active:
                    wordCount += 1
            if wordCount == 1:
                count += 1

        if count > self.bestSolution["find"]:
            for row in range(0, 5):
                self.bestSolution["across"][row] = copy.deepcopy(
                    self.domains["across"][row])
            for col in range(0, 5):
                self.bestSolution["down"][col] = copy.deepcopy(
                    self.domains["down"][col])
            self.bestSolution["find"] = count

        # final consideration
        if count > 5:
            tempSolution = {"across": [[], [], [],
                                       [], []], "down": [[], [], [], [], []]}
            for row in range(0, 5):
                tempSolution["across"][row] = copy.deepcopy(
                    self.domains["across"][row])
            for col in range(0, 5):
                tempSolution["down"][col] = copy.deepcopy(
                    self.domains["down"][col])
            self.finalSolutions.append(tempSolution)
```

Appendix G: Source Code of Scoring Grids

```
def scoreGrid(self):
        """
        This function scores grid by looking at each answers relation
        to the according clue. If an answer is related to the clue,
        it gets score given in the datamuse data.
        """
        downDict = {}
        acrossDict = {}
        for i in self.downClues:
            downDict[i[0]] = i[1]
        for i in self.acrossClues:
            acrossDict[i[0]] = i[1]

        self.helperScoreGrid(self.acrossAnswers,  acrossDict)
        self.helperScoreGrid(self.downAnswers, downDict)

    def helperScoreGrid(self, answers, clues):
        for answer in answers:
            word = clues[answer]
            r = requests.get('https://api.datamuse.com/words?ml={}'.format(word))
            if r != None:
                results = r.json()
                count = 0
                for result in results:
                    count += 1
                    if result['word'] ==
answers[answer].lower():#answers[indices[index]].lower():
                        self.score += result['score']
                        break
                    if count > 50:
                        break
```

Appendix H: Entire Source Code

Connector.py

```
from selenium import webdriver
from selenium.webdriver.common.keys import Keys
import time

class Connector:
    def __init__(self, PATH):
        self.cluesAcross = []
        self.cluesDown = []
        self.PATH = PATH
        self.cellNumberArray = []
        self.cellBlockArray = []
        self.cellAnswerArray = []

    def connectToPuzzle(self):
        separtor = "============================\n"
        print(separtor , "CONNECTING TO THE WEBSITE\n"+ separtor)
        options = webdriver.ChromeOptions()
        options.add_experimental_option("excludeSwitches", ["enable-logging"])
        driver = webdriver.Chrome(self.PATH, options=options)
        driver.get("https://www.nytimes.com/crosswords/game/mini")

        """
        Skip first web message
        """
        time.sleep(1)
        okButton  =
driver.find_element_by_xpath("//*[@id=\"root\"]/div/div/div[4]/div/main/div[2]/div/div
[2]/div[3]/div/article/div[2]/button")
        driver.execute_script("arguments[0].click();", okButton)


        """
        REVEAL ANSWERS
        """
        revealButton =
driver.find_element_by_xpath("//*[@id=\"root\"]/div/div/div[4]/div/main/div[2]/div/div
/ul/div[2]/li[2]/button")
        driver.execute_script("arguments[0].click();", revealButton)
        #revealButton.click()

        revealPuzzleButton =
driver.find_element_by_xpath("//*[@id=\"root\"]/div/div/div[4]/div/main/div[2]/div/div
[1]/ul/div[2]/li[2]/ul/li[3]/a")
        driver.execute_script("arguments[0].click();", revealPuzzleButton)
```

```
        #revealPuzzleButton.click()

        yesIAmSureButton  =
driver.find_element_by_xpath("//*[@id=\"root\"]/div/div[2]/div[2]/article/div[2]/butto
n[2]")
        driver.execute_script("arguments[0].click();", yesIAmSureButton)

        closePopUp  =
driver.find_element_by_xpath("//*[@id=\"root\"]/div/div[2]/div[2]/span")
        driver.execute_script("arguments[0].click();", closePopUp)

        """
        Get clue and cell data
        """
        print(separtor , "RECEIVING CLUES\n"+ separtor)
        clue = driver.find_element_by_class_name("Layout-clueLists--10_Xl")
        childs = clue.find_elements_by_class_name("Clue-li--1JoPu")
        cells = driver.find_elements_by_tag_name("g")

        """
        Manage Clues
        """
        for i in range(0,5):
            clueNum = childs[i].find_elements_by_css_selector("*")[0].text
            clueText = childs[i].find_elements_by_css_selector("*")[1].text
            self.cluesAcross.append([clueNum, clueText])
        for i in range(5,10):
            clueNum = childs[i].find_elements_by_css_selector("*")[0].text
            clueText = childs[i].find_elements_by_css_selector("*")[1].text
            self.cluesDown.append([clueNum, clueText])

        """
        Print clues
        """
        print("***ACROSS CLUES***")
        for i in self.cluesAcross:
            print(i)
        print("\n***DOWN CLUES***")
        for i in self.cluesDown:
            print(i)

        """
        Manage Grid Cells
        """
        print(separtor , "RECEIVING CELL DATA\n"+ separtor)
        #print("G size: ", len(cells))
```

```
        tempNumber = []
        tempBlock = []
        tempAnswer = []

        for g in range(0,25):
            if g%5 == 0:
                tempNumber = []
                tempBlock = []
                tempAnswer = []
            #print(g+1, cells[g+4].text)
            cellMiniNumber = ""
            cellAnswer = ""

            cellText = cells[g+4].text
            for s in cellText:
                if s.isdigit():
                    cellMiniNumber = s
                if s.isupper():
                    cellAnswer = s

            className =
cells[g+4].find_elements_by_css_selector("rect")[0].get_attribute("class")
            if "block" in className:
                className = "1"
            else:
                className = "0"

            if cellMiniNumber == '':
                cellMiniNumber = "-"
            if cellAnswer == '':
                cellAnswer = "-"

            tempNumber.append(cellMiniNumber)
            tempBlock.append(className)
            tempAnswer.append(cellAnswer)
            if g%5 == 4:
                self.cellNumberArray.append(tempNumber)
                self.cellBlockArray.append(tempBlock)
                self.cellAnswerArray.append(tempAnswer)

        """
        Print Cell Data
        """
        print("*** CELL INDEX NUMBERS ***")
        for r in self.cellNumberArray:
            print(r)
        print("\n*** BLOCK CELL MAP ***")
```

```
        for r in self.cellBlockArray:
            print(r)
        print("\n*** CELLS WITH SOLUTIONS ***")
        for r in self.cellAnswerArray:
            print(r)

        """
        Quit browser
        """
        print(separtor , "CLOSING THE BROWSER\n"+ separtor)
        driver.quit()


    def print_clues(self):
        """
        Print Clues
        """
        print("Across Clues")
        for clue in self.cluesAcross:
            print(clue)

        print("Down Clues")
        for clue in self.cluesDown:
            print(clue)

    def print_grid_cells(self):
        """
        Print Grid Cells
        """
        print("\nGrid Number")
        for row in self.cellNumberArray:
            print(row)
        print("Grid Blocks")
        for row in self.cellBlockArray:
            print(row)
        print("Grid Answer")
        for row in self.cellAnswerArray:
            print(row)
```

combine_tokens.py

```
import nltk
import string
```

```
from nltk.corpus import stopwords

stop_words = set(stopwords.words('english'))

# Processes the clue to generate token pairs to be later used in scraping
# Returns all the possible pairs of tokens for a clue
def combine_tokens(clue, acrossClues, downClues):
    # Checks if the clue is in the form of "See X-Down" or "See X-Across"
    if "See" in clue:
        clue = getPointedClue(clue, acrossClues, downClues)

    clue = clue.lower()
    #clue = clue.replace("-", " ")
    clue_without_punctuation = clue.translate(
        str.maketrans('', '', string.punctuation))
    tokens = nltk.word_tokenize(clue_without_punctuation)
    tokens = [w for w in tokens if not w in stop_words]
    number_of_tokens = len(tokens)
    for i in range(number_of_tokens):
        if i != number_of_tokens - 1:
            tokens.append(tokens[i] + " " + tokens[i + 1])
    # print(tokens)
    return tokens

# Used to find the clues that are in the form of "See X-Down" or "See X-Across"
def getPointedClue(clue, acrossClues, downClues):
    pointedClue = ""
    if "Down" in clue:
        pointedClue = downClues[clue[4]]
    else:
        pointedClue = acrossClues[clue[4]]
    return pointedClue
```

searchWikipedia.py

```
import wikipedia
import string
from combineTokens import combine_tokens

def searchWikipedia(clue):
    """
    This function takes clue and search for the clue in wikipedia.
    Takes the first result found in wikipedia and considers only the summary
    of that page.
    Returns a set containing words it found on a wikipedia page.
    """
    text = set()
```

```
    clue = clue.translate(str.maketrans('', '', string.punctuation))
    search_results = wikipedia.search(clue, results=1)
    for result in search_results:
        try:
            content = wikipedia.page(result + ".").summary.translate(str.maketrans('',
'', string.punctuation))
            for w in content.split():
                text.add(w.lower())
        except:
            continue

    return text
```

synonyms.py

```
import requests
from time import sleep
from bs4 import BeautifulSoup
import string
from combineTokens import combine_tokens
import cProfile
import lxml
import cchardet

def searchSynonyms(clue, acrossClues, downClues):
    """
    This function takes clue, list of across clues and list of down clues.
    It tokenizes the clue first to subwords and search for each word on Merriam
Webster
    and Datamuse.
    """
    print('Search list for "' + clue + '":', end=' ')
    tokens = combine_tokens(clue, acrossClues, downClues)
    print(tokens)
    words = set()
    for word in tokens:
        if(contains_multiple_words(word)):
            # Merriam webster
            r_mw = requests.get("https://www.merriam-webster.com/thesaurus/" +
word.split()[0] + "%20" + word.split()[1])

            # Datamuse
            r_dm =
requests.get('https://api.datamuse.com/words?rel_syn={}'.format(word))
            r_dm2 = requests.get('https://api.datamuse.com/words?ml={}'.format(word))

        else:
            # Merriam webster
```

```
            r_mw = requests.get("https://www.merriam-webster.com/thesaurus/" + word)

            # Datamuse
            r_dm =
requests.get('https://api.datamuse.com/words?rel_syn={}'.format(word))
            r_dm2 = requests.get('https://api.datamuse.com/words?ml={}'.format(word))

        soup = BeautifulSoup(r_mw.text, 'lxml')

        # Search for synonym list on Merriam Webster
        for i in soup.select('.thes-list.syn-list a'):
            words.add(str(i.string).lower())

        # Search for related list on Merriam Webster
        for i in soup.select('.thes-list.rel-list a'):
            words.add(str(i.string).lower())

         # Search for similarity list on Merriam Webster
        for i in soup.select('.thes-list.sim-list a'):
            words.add(str(i.string).lower())

        count = 0
        results = r_dm.json()
        for result in results:
            words.add(result['word'])
            count += 1
            if count > 20: # 20 is an enough value to get synonyms from one website
                break

        results = r_dm2.json()
        count = 0
        for result in results:
            words.add(result['word'])
            count += 1
            if count > 20:
                break
    return words


def contains_multiple_words(s):
  return len(s.split()) > 1
```

scraping.py

```
from searchWikipedia import searchWikipedia
from synonyms import searchSynonyms
```

```
class Scraping:
    def __init__(self, clues, answers, gridIndex):
        self.clues = clues
        self.domains = {"across": {}, "down":{}}
        self.answers = answers
        self.gridIndex = gridIndex

    def setDomains(self):
        for down in self.clues["down"]:
            self.domains["down"][down] = self.search(self.clues["down"][down])
        for across in self.clues["across"]:
            self.domains["across"][across] = self.search(self.clues["across"][across])

    def search(self, clue):
        """
        This function searches clue on wikipedia, merriam webster and datamuse.
        Calls two functions, getWiki and getSynonyms. At the end of the function,
        it gets union of the returning sets from the functions so that duplicates
        are eliminated.
        Returns a text including all words that are found on web.
        """
        domain = set()
        wiki_set = set()
        synonym_set = set()
        toSearch = clue

        print("Wikipedia search for:", toSearch)
        try:
            wiki_set = wiki_set | self.getWiki(toSearch)
        except:
            print("An exception occurred")

        print("Synonym search from Datamuse and Merriam-Webster for:", toSearch)
        try:
            synonym_set = synonym_set | self.getSynonyms(toSearch)
        except:
            print("An exception occurred")

        domain = domain.union(wiki_set, synonym_set)
        return ' '.join(str(e) for e in domain)

    def getWiki(self, toSearch):
        return searchWikipedia(toSearch)

    def getSynonyms(self, toSearch):
        return searchSynonyms(toSearch, self.clues["across"], self.clues["down"])
```

```
    def cheat(self):
        for across in self.clues["across"]:
            for row in range(0,5):
                for col in range(0,5):
                    if self.gridIndex[row][col] == across:
                        answer = ""
                        for colIn in range(0,5):
                            if self.answers[row][colIn] != "-":
                                answer = answer + self.answers[row][colIn]
                        self.domains["across"][across] =
self.domains["across"][across] + " " + answer

        for down in self.clues["down"]:
            for row in range(0,5):
                for col in range(0,5):
                    if self.gridIndex[row][col] == down:
                        answer = ""
                        for rowIn in range(0,5):
                            if self.answers[rowIn][col] != "-":
                                answer = answer + self.answers[rowIn][col]
                        self.domains["down"][down] = self.domains["down"][down] + " "
+ answer
```

solver.py

```
import itertools
import copy
import string
import json
from ScorePuzzle import ScorePuzzle


class Word:
    """This Class hold a single word in the domain, we used an object because we had
to access its active features in a constant time.
    """
    def __init__(self, word, type, rowColIndex, clueIndex, active):
        self.word = word
        self.type = type
        self.rowColIndex = rowColIndex
        self.clueIndex = clueIndex
        self.active = active


class NewSolver:
    def __init__(self, grid, numbers, downClues, acrossClues, domains):
        self.grid = grid
        self.numbers = numbers
```

```
        self.downClues = downClues
        self.acrossClues = acrossClues
        self.initialDomains = copy.deepcopy(domains)  # INITIAL

        with open('data.json', 'w') as fp:
            json.dump(domains, fp,  indent=4)

        self.lengthOfDownClues = {}
        self.lengthOfAcrossClues = {}
        self.locationOfDownClues = {}
        self.locationOfAcrossClues = {}
        self.filteredDomains = {"down": {}, "across": {}}  # FILTERED
        self.neglectedWords = {"row": [], "col": []}
        self.neglectedWordsArray = []
        self.count = 0
        self.bestSolution = {"across": [[], [], [], [], []], "down": [
            [], [], [], [], []], "find": 0}
        self.finalSolutions = []
        self.busGrids = []
        self.finalSolution = []
        self.solvedPuzzle = []
        self.domains = {"across": [[], [], [],
                                   [], []], "down": [[], [], [], [], []]}
        self.cells = [[{"across": {}, "down": {}}, {"across": {}, "down": {}},
{"across": {
        }, "down": {}}, {"across": {}, "down": {}}, {"across": {}, "down": {}}] for r
in range(5)]

        self.setup()
        print("====================\nINITIAL FILTERED DOMAINS\n====================")
        self.printDomainss()
        # with open('filteredDomains.json', 'w') as fp:
       #   json.dump(self.domains, fp,  indent=4)

        # self.tempDomains = copy.deepcopy(self.domains) #TEMP
        #self.tempCells = copy.deepcopy(self.cells)
        print("====================\nFILTERING DOMAINS BY APPLYING
CONSTRAINTS\n====================")
        self.solver()
        print("====================\nBEST DOMAINS FROM FILTERING\n====================")
        self.printBestDomains()
        # self.getAnswerGrid()
        print(
            "====================\nCOMPARING MULTIPLE OPTIMAL
SOLUTIONS\n====================")
        print("\n")
        print("====================\nOPTIMAL GRIDS TO COMPARE\n====================")
```

```
        self.findFinalPuzzle()
        self.solvedPuzzle = self.finalSolution

        # print(self.bestSolution["find"])

    def setup(self):
        self.wordLengthCalculator()
        self.filterDomains()
        self.deleteDups()
        self.setDomains()
        self.setCells()

    def setDomains(self):
        """This function sets the location and length of the domains of the clues.
        """
        for i in range(0, 5):
            for across in self.filteredDomains["across"]:
                if self.locationOfAcrossClues[across]["start"]["row"] == i:
                    for word in self.filteredDomains["across"][across]:
                        self.domains["across"][i].append(word)
        for i in range(0, 5):
            for down in self.filteredDomains["down"]:
                if self.locationOfDownClues[down]["start"]["col"] == i:
                    for word in self.filteredDomains["down"][down]:
                        self.domains["down"][i].append(word)

    def reset(self):
        """This function resets the domain from constraint reduction, so the domain
becomes the initial domain.
        """
        for index in range(0, 5):
            for word in self.domains["across"][index]:
                word.active = True
            for word in self.domains["down"][index]:
                word.active = True

    def setCells(self):
        """This function stores the possible letters in each cell in order to fasten
the constraint reduction process
        """
        for row in range(0, 5):
            for col in range(0, 5):
                for letter in string.ascii_uppercase:
                    self.cells[row][col]["across"][letter] = []
                    self.cells[row][col]["down"][letter] = []

        for row in range(0, 5):
```

```
            for col in range(0, 5):
                for letter in string.ascii_uppercase:
                    if self.getTheRelatedDomainOfThisCell(row, col, "") != {}:
                        for word in self.getTheRelatedDomainOfThisCell(row, col,
"")["across"]["domain"]:
                            if letter ==
word.word[self.getTheRelatedDomainOfThisCell(row, col, "")["across"]["index"]]:
                                self.cells[row][col]["across"][letter].append(
                                    word)
                        for word in self.getTheRelatedDomainOfThisCell(row, col,
"")["down"]["domain"]:
                            if letter ==
word.word[self.getTheRelatedDomainOfThisCell(row, col, "")["down"]["index"]]:
                                self.cells[row][col]["down"][letter].append(
                                    word)

    def solver(self):
        """This function reduces the domains with constraint check.
        After the domains are reduced in size,
        the solver tries combinations of domains in order to find combinations of
        words that can fit the puzzle while still holding the constraints.
        Since some answers may not be available to us in the domains,
        it neglects an answer and its constraints and searches for a possible solution
again.
        Then the outcome will send to the validators and if it is solved enough the
best solutions are keept
        """
        puzzleNotSolved = True
        # while puzzle is solved try more jokers
        while puzzleNotSolved:
            puzzleNotSolved = False
            self.reset()
            if self.changeNeglected():
                changeMade = True
                # Constraints checks
                while changeMade:
                    changeMade = False
                    for row in range(0, 5):
                        # --------------------
                        if row not in self.neglectedWords["row"]:
                            for col in range(0, 5):
                                # --------------------
                                if col not in self.neglectedWords["col"]:
                                    for letter in string.ascii_uppercase:
                                        # check wheter the letters in the cells are
matchad. If not reduce domain.
```

```
                                    if self.getTrueFalse(row, col, "across",
letter) and not self.getTrueFalse(row, col, "down", letter):
                                        for word in
self.cells[row][col]["across"][letter]:
                                            word.active = False
                                        changeMade = True

                                    elif not self.getTrueFalse(row, col, "across",
letter) and self.getTrueFalse(row, col, "down", letter):
                                        for word in
self.cells[row][col]["down"][letter]:
                                            word.active = False
                                        changeMade = True
                self.isItTheBestSolution()  # test for best solution

                # if the puzzle is solved. Stop loop
                puzzleNotSolved = not self.isPuzzleSolved()
            else:
                return False

    def getTrueFalse(self, row, col, acrossDown, letter):
        """This function returns if the letter given as a parameter is present or not
in the desired cell.
        """
        for word in self.cells[row][col][acrossDown][letter]:
            if word.active:
                return True
        return False

    def getCurrentWords(self, row, col, acrossDown, letter):
        """This function returns the presenet words in the desired cell for specific
letter.
        """
        words = []
        for word in self.cells[row][col][acrossDown][letter]:
            if word.active:
                words.append(word)
        return words

    def printDomainss(self):
        """This function prints the current domains.
        """
        print("Across")
        for col in range(0, 5):
            print(str(col) + ": ", end=" ")
            for word in self.domains["across"][col]:
                if word.active:
```

```
                print(word.word, end=" ")
        print("\n")

    print("\nDown")
    for row in range(0, 5):
        print(str(row) + ": ", end=" ")
        for word in self.domains["down"][row]:
            if word.active:
                print(word.word, end=" ")
        print("\n")

def printDomainLen(self):
    """This function prints the size of the domains.
    """
    print("Across")
    for col in range(0, 5):
        count = 0
        for word in self.domains["across"][col]:
            if word.active:
                count += 1
        print(str(col) + ": " + str(count))

    print("\nDown")
    for row in range(0, 5):
        count = 0
        for word in self.domains["down"][row]:
            if word.active:
                count += 1
        print(str(row) + ": " + str(count))

def printCells(self):
    """This function prints the content of the cells
    """
    for row in range(0, 5):
        for col in range(0, 5):
            print(str(row) + ", " + str(col) + ": ")
            print("Across: ", end="")
            for letter in self.cells[row][col]["across"]:
                if len(self.cells[row][col]["down"][letter]) > 0:
                    print(letter, end=": ")
                    for word in self.cells[row][col]["across"][letter]:
                        print(word.word, end=", ")
                    print("//", end="")
            print()
            print("Down: ", end="")
            for letter in self.cells[row][col]["down"]:
                if len(self.cells[row][col]["down"][letter]) > 0:
```

```
                        print(letter, end=": ")
                        for word in self.cells[row][col]["down"][letter]:
                            print(word.word, end=", ")
                        print("//", end="")
                print()

    def wordLengthCalculator(self):
        """For every clue, calculates the row and column points where the answer for
that clue starts and finishes in the grid. The values are stored inside
'locationOfDownClues' and 'locationOfAcrossClues' in the form of a dictionary.
        """
        # downClues
        for clue in self.downClues:
            clueNumber = clue[0]
            rowIndex = -1
            colIndex = -1
            counter = 0
            for row in self.numbers:
                if clueNumber in row:
                    rowIndex = counter
                counter = counter + 1
            colIndex = self.numbers[rowIndex].index(clueNumber)

            # count spaces
            wordLength = 0
            for row in self.grid[rowIndex:]:
                if row[colIndex] == "0":
                    wordLength = wordLength + 1
            self.lengthOfDownClues[clueNumber] = wordLength
            self.locationOfDownClues[clueNumber] = {"start": {
                "row": rowIndex, "col": colIndex}, "end": {"row":
rowIndex+wordLength-1, "col": colIndex}}

            # acrossClues
        for clue in self.acrossClues:
            clueNumber = clue[0]
            rowIndex = -1
            colIndex = -1
            counter = 0
            for row in self.numbers:
                if clueNumber in row:
                    rowIndex = counter
                counter = counter + 1
            colIndex = self.numbers[rowIndex].index(clueNumber)

            # count spaces
            wordLength = 0
```

```
            for cell in self.grid[rowIndex][colIndex:]:
                if cell == "0":
                    wordLength = wordLength + 1
            self.lengthOfAcrossClues[clueNumber] = wordLength
            self.locationOfAcrossClues[clueNumber] = {"start": {
                "row": rowIndex, "col": colIndex}, "end": {"row": rowIndex, "col":
colIndex+wordLength-1}}

    def filterDomains(self):
        """Filters the words that comes from the web scraping in order to place them
into domains that belong to each clue.
            It checks for length, for example for an across clue with 4 available
spaces in the grid, only words with 4 letters pass.
            It checks for spaces around the words and duplicate words and remove them
from the domain.
            Also for the two word combinations, if the domain includes two words that
do not exceed the length constraint when combined, it combines them into a new word.
        """
        # downClues
        newDomain = set()
        for clue in self.initialDomains["down"]:
            previousWord = ""
            for word in self.initialDomains["down"][clue].split():
                word = self.filterHelper(word)
                if word[len(word)-1] == "." or word[len(word)-1] == "," or
word[len(word)-1] == ":" or word[len(word)-1] == ";" or word[len(word)-1] == "+" or
word[len(word)-1] == "?" or word[len(word)-1] == "!" or word[len(word)-1] == ")" or
word[len(word)-1] == "}" or word[len(word)-1] == "]":
                    word = word[0:len(word)-1]
                # if word lenght is valid
                if (len(word) == self.lengthOfDownClues[clue]) and not
self.checkDuplicates(clue, "down", word.upper()):
                    newDomain.add(Word(word.upper(), "down", -1, clue, True))
                prevPlusCur = previousWord + word
                # if two words next to each others total length is valid
                if (len(prevPlusCur) == self.lengthOfDownClues[clue]) and (prevPlusCur
!= word) and not self.checkDuplicates(clue, "down", prevPlusCur.upper()):
                    newDomain.add(
                        Word(prevPlusCur.upper(), "down", -1, clue, True))
                previousWord = word
                # if the word ends with "s"
                if (len(word) == self.lengthOfDownClues[clue]+1) and
(word[self.lengthOfDownClues[clue]] == "s") and not self.checkDuplicates(clue, "down",
word[0:self.lengthOfDownClues[clue]].upper()):
                    newDomain.add(
                        Word(word[0:self.lengthOfDownClues[clue]].upper(), "down", -1,
clue, True))
```

```
                if ("'" in word) and (len(word[0:word.index("'")]) ==
self.lengthOfDownClues[clue]) and not self.checkDuplicates(clue, "down",
word[0:word.index("'")].upper()):
                    newDomain.add(
                        Word(word[0:word.index("'")].upper(), "down", -1, clue, True))
            for word in self.initialDomains["down"][clue].split("'"):
                # if word lenght is valid
                if (len(word) == self.lengthOfDownClues[clue]) and not
self.checkDuplicates(clue, "down", word.upper()):
                    newDomain.add(Word(word.upper(), "down", -1, clue, True))
            newDomain = list(newDomain)
            for word in reversed(newDomain):
                deleted = False
                for letter in word.word:
                    if not letter in list(string.ascii_uppercase) and not deleted:
                        newDomain.remove(word)
                        deleted = True

            def lexical(word):
                return word.word[0]
            newDomain.sort(key=lexical)
            #self.downClueDomains[clue] = newDomain.copy()
            self.filteredDomains["down"][clue] = copy.deepcopy(newDomain)
            newDomain = set()

        # acrossClues
        newDomain = set()
        for clue in self.lengthOfAcrossClues:
            previousWord = ""
            for word in self.initialDomains["across"][clue].split():
                word = self.filterHelper(word)
                if word[len(word)-1] == "." or word[len(word)-1] == "," or
word[len(word)-1] == ":" or word[len(word)-1] == ";" or word[len(word)-1] == "+" or
word[len(word)-1] == "?" or word[len(word)-1] == "!" or word[len(word)-1] == ")" or
word[len(word)-1] == "}" or word[len(word)-1] == "]":
                    word = word[0:len(word)-1]
                # if word lenght is valid
                if (len(word) == self.lengthOfAcrossClues[clue]) and not
self.checkDuplicates(clue, "across", word.upper()):
                    newDomain.add(Word(word.upper(), "across", -1, clue, True))
                prevPlusCur = previousWord + word
                # if two words next to each others total length is valid
                if (len(prevPlusCur) == self.lengthOfAcrossClues[clue]) and
(prevPlusCur != word) and not self.checkDuplicates(clue, "across",
prevPlusCur.upper()):
                    newDomain.add(Word(prevPlusCur.upper(),
                                       "across", -1, clue, True))
```

```
                previousWord = word
                # if the word ends with "s"
                if (len(word) == self.lengthOfAcrossClues[clue]+1) and
(word[self.lengthOfAcrossClues[clue]] == "s") and not self.checkDuplicates(clue,
"across", word[0:self.lengthOfAcrossClues[clue]].upper()):
                    newDomain.add(
                        Word(word[0:self.lengthOfAcrossClues[clue]].upper(), "across",
-1, clue, True))
                if ("'" in word) and (len(word[0:word.index("'")]) ==
self.lengthOfAcrossClues[clue]) and not self.checkDuplicates(clue, "across",
word[0:word.index("'")].upper()):
                    newDomain.add(
                        Word(word[0:word.index("'")].upper(), "across", -1, clue,
True))
            for word in self.initialDomains["across"][clue].split("'"):
                # if word lenght is valid
                if (len(word) == self.lengthOfAcrossClues[clue]) and not
self.checkDuplicates(clue, "across", word.upper()):
                    newDomain.add(Word(word.upper(), "across", -1, clue, True))
            newDomain = list(newDomain)
            for word in reversed(newDomain):
                deleted = False
                for letter in word.word:
                    if not letter in list(string.ascii_uppercase) and not deleted:
                        newDomain.remove(word)
                        deleted = True

            def lexical(word):
                return word.word[0]
            newDomain.sort(key=lexical)
            #self.acrossClueDomains[clue] = newDomain
            self.filteredDomains["across"][clue] = copy.deepcopy(newDomain)
            newDomain = set()

    def checkDuplicates(self, clueIndex, acrossDown, wordToCompare):
        """Check for duplicates inside the domain in order to eliminate them.
        """
        if clueIndex in self.filteredDomains[acrossDown]:
            for word in self.filteredDomains[acrossDown][clueIndex]:
                if wordToCompare == word.word:
                    return True
        return False

    def deleteDups(self):
        """Deletes duplicate words from the domains.
        """
        for key in self.filteredDomains["across"]:
```

```
            for word in self.filteredDomains["across"][key]:
                wordToCheck = word.word
                wordCount = 0
                for wordIn in reversed(self.filteredDomains["across"][key]):
                    if wordToCheck == wordIn.word:
                        wordCount += 1
                        if wordCount > 1:
                            self.filteredDomains["across"][key].remove(wordIn)

        for key in self.filteredDomains["down"]:
            for word in self.filteredDomains["down"][key]:
                wordToCheck = word.word
                wordCount = 0
                for wordIn in reversed(self.filteredDomains["down"][key]):
                    if wordToCheck == wordIn.word:
                        wordCount += 1
                        if wordCount > 1:
                            self.filteredDomains["down"][key].remove(wordIn)

    def filterHelper(self, input):
        """Converts any numerical value into string form.

        Args:
            input (char): A number inside a string which wil be converted into string
form.
        """
        if input == "0":
            return "ZERO"
        if input == "1":
            return "ONE"
        if input == "2":
            return "TWO"
        if input == "3":
            return "THREE"
        if input == "4":
            return "FOUR"
        if input == "5":
            return "FIVE"
        if input == "6":
            return "SIX"
        if input == "7":
            return "SEVEN"
        if input == "8":
            return "EIGHT"
        if input == "9":
            return "NINE"
        if input == "10":
```

```
            return "TEN"
        if input == "-":
            return "MINUS"
        if input == "+":
            return "PLUS"
        return input

    def getTheRelatedDomainOfThisCell(self, row, col, option):
        """For a given cell in the grid, find the clue answers that pass through that
cell and return the domains of those clue answers.

        Args:
            row (int): row index of given cell
            col (int): col index of given cell

        Returns:
            Returns the domains (all possible words) for the clue answers that pass
through the given cell.
        """
        domains = {}
        # down
        for location in self.locationOfDownClues:
            wordIndex = -1
            tempCol = self.locationOfDownClues[location]["start"]["col"]
            rowStart = self.locationOfDownClues[location]["start"]["row"]
            rowEnd = self.locationOfDownClues[location]["end"]["row"]
            if (row <= int(rowEnd)) and (row >= int(rowStart)) and (col ==
int(tempCol)):
                wordIndex = row-rowStart
                if option == "best":
                    domains["down"] = {"index": wordIndex, "domain":
self.getCurrentDomainWord(
                        "down", tempCol), "loc": self.locationOfDownClues[location]}
                else:
                    domains["down"] = {"index": wordIndex, "domain":
self.filteredDomains["down"]
                                       [location], "loc":
self.locationOfDownClues[location]}
        # across
        for location in self.locationOfAcrossClues:
            wordIndex = -1
            tempRow = self.locationOfAcrossClues[location]["start"]["row"]
            colStart = self.locationOfAcrossClues[location]["start"]["col"]
            colEnd = self.locationOfAcrossClues[location]["end"]["col"]
            if (col <= int(colEnd)) and (col >= int(colStart)) and (row ==
int(tempRow)):
                wordIndex = col-colStart
```

```
                if option == "best":
                    domains["across"] = {"index": wordIndex, "domain":
self.getCurrentDomainWord(
                        "across", tempRow), "loc":
self.locationOfAcrossClues[location]}
                else:
                    domains["across"] = {"index": wordIndex, "domain":
self.filteredDomains["across"]
                                         [location], "loc":
self.locationOfAcrossClues[location]}
        return domains

    def getCurrentDomainWord(self, acrossDown, index):
        """This function returns the words after the domain is filtered and becomes
the final domain.
        """
        words = []
        for word in self.bestSolution[acrossDown][index]:
            if word.active:
                words.append(word)
        return words

    def getAnswerGrid(self):
        """This function prints the best solutions grid. With considering across and
down solutions seperately,
        and then prints the matched cells otherwise leaves empty
        """
        if self.bestSolution["find"] != 0:
            answerGrid = [["", "", "", "", ""], ["", "", "", "", ""], [
                "", "", "", "", ""], ["", "", "", "", ""], ["", "", "", "", ""]]
            for row in range(0, 5):
                for col in range(0, 5):
                    if answerGrid[row][col] == "":
                        domains = self.getTheRelatedDomainOfThisCell(
                            row, col, "best")
                        if domains == {}:
                            answerGrid[row][col] = "-"
                        else:
                            if len(self.getCurrentDomainWord("across", row)) == 1:
                                for colIndex in
range(domains["across"]["loc"]["start"]["col"],
domains["across"]["loc"]["start"]["col"] + len(domains["across"]["domain"][0].word)):
                                    answerGrid[row][colIndex] =
domains["across"]["domain"][0].word[colIndex -

domains["across"]["loc"]["start"]["col"]]
                            if len(self.getCurrentDomainWord("down", col)) == 1:
```

```
                                for rowIndex in
range(domains["down"]["loc"]["start"]["row"], domains["down"]["loc"]["start"]["row"] +
len(domains["down"]["domain"][0].word)):
                                    answerGrid[rowIndex][col] =
domains["down"]["domain"][0].word[rowIndex -

domains["down"]["loc"]["start"]["row"]]

            for row in range(0, 5):
                for col in range(0, 5):
                    if answerGrid[row][col] == "":
                        answerGrid[row][col] = "*"

            for row in answerGrid:
                print(row)
            self.solvedPuzzle = answerGrid
        else:  # If there is no best solution program failes
            print(" CORT ")

    def isPuzzleSolved(self):
        """This function checks if the current domains are enough
        for filling all the cells of the puzzle with one possible option
        """
        answerGrid = [["", "", "", "", ""], ["", "", "", "", ""], [
            "", "", "", "", ""], ["", "", "", "", ""], ["", "", "", "", ""]]
        puzzleSolved = True
        for row in range(0, 5):
            for col in range(0, 5):
                if answerGrid[row][col] == "":
                    domains = self.getTheRelatedDomainOfThisCell(row, col, "")
                    if domains == {}:
                        answerGrid[row][col] = "-"
                    else:
                        if len(self.getCurrentDomainWord("across", row)) == 1:
                            for colIndex in
range(domains["across"]["loc"]["start"]["col"],
domains["across"]["loc"]["start"]["col"] + len(domains["across"]["domain"][0].word)):
                                answerGrid[row][colIndex] =
domains["across"]["domain"][0].word[colIndex -

domains["across"]["loc"]["start"]["col"]]
                        if len(self.getCurrentDomainWord("down", col)) == 1:
                            for rowIndex in
range(domains["down"]["loc"]["start"]["row"], domains["down"]["loc"]["start"]["row"] +
len(domains["down"]["domain"][0].word)):
                                answerGrid[rowIndex][col] =
domains["down"]["domain"][0].word[rowIndex -
```

```
domains["down"]["loc"]["start"]["row"]]
        for row in range(0, 5):
            for col in range(0, 5):
                if answerGrid[row][col] == "":
                    puzzleSolved = False
        return puzzleSolved

    def printBestDomains(self):
        """This function prints the domains of best solution found
        """
        print("Across")
        for col in range(0, 5):
            print(str(col) + ": ", end=" ")
            for word in self.bestSolution["across"][col]:
                if word.active:
                    print(word.word, end=" ")
            print("\n")

        print("\nDown")
        for row in range(0, 5):
            print(str(row) + ": ", end=" ")
            for word in self.bestSolution["down"][row]:
                if word.active:
                    print(word.word, end=" ")
            print("\n")

    def changeNeglected(self):
        """This program creates all combinations of neglegted clues and
        iterates them when called
        """
        if self.count == 0:
            rows = [0, 1, 2, 3, 4]
            for r in range(0, len(rows)+1):
                for rsubset in itertools.combinations(rows, r):
                    cols = [0, 1, 2, 3, 4]
                    for c in range(0, len(cols)+1):
                        for csubset in itertools.combinations(cols, c):
                            self.neglectedWordsArray.append(
                                {"row": rsubset, "col": csubset})
            self.neglectedWords = self.neglectedWordsArray[self.count]
            self.count = self.count + 1
            return True
        else:
            if self.count == 1024:
                return False
            self.neglectedWords = self.neglectedWordsArray[self.count]
```

```
            self.count = self.count + 1
            return True

    def isItTheBestSolution(self):
        """This function is look for the best solution.
        If a domain of the clue is reduced to a single word, it recieves +1 score
        the most scored solution is determined as the best solution
        """
        count = 0
        for row in range(0, 5):
            wordCount = 0
            for word in self.domains["across"][row]:
                if word.active:
                    wordCount += 1
            if wordCount == 1:
                count += 1
        for col in range(0, 5):
            wordCount = 0
            for word in self.domains["down"][col]:
                if word.active:
                    wordCount += 1
            if wordCount == 1:
                count += 1

        if count > self.bestSolution["find"]:
            for row in range(0, 5):
                self.bestSolution["across"][row] = copy.deepcopy(
                    self.domains["across"][row])
            for col in range(0, 5):
                self.bestSolution["down"][col] = copy.deepcopy(
                    self.domains["down"][col])
            self.bestSolution["find"] = count

        # final consideration
        if count > 5:
            tempSolution = {"across": [[], [], [],
                                       [], []], "down": [[], [], [], [], []]}
            for row in range(0, 5):
                tempSolution["across"][row] = copy.deepcopy(
                    self.domains["across"][row])
            for col in range(0, 5):
                tempSolution["down"][col] = copy.deepcopy(
                    self.domains["down"][col])
            self.finalSolutions.append(tempSolution)

    def findFinalPuzzle(self):
```

```
        """This function tests the solutions which are closely solved. Generally there
are 5-10 solutions.
        And crosscheck the answers placed on the grid with their clues in order to
decide if they are relevant.
        If we have placed an irrelevant word just because it fits in the constraints,
it will eliminate it.
        It gives a ratinf to the solution by checking every word on it and giving it a
relevancy score.
        The solution with most relevant words on it is the best solution.
        """
        if len(self.finalSolution) == 1:
            self.finalSolution = self.gridMaker(self.finalSolution[0])
        else:
            maxPoint = 0
            for solution in self.finalSolutions:
                grid = self.gridMaker(solution)
                print("Calculating score for a possible grid:")
                for row in grid:
                    print(row)
                print("")
                score = ScorePuzzle(grid, self.locationOfAcrossClues,
                                    self.locationOfDownClues, self.acrossClues,
self.downClues).score
                print("Score: ", score)
                if score > maxPoint:
                    maxPoint = score
                    self.finalSolution = grid

    def gridMaker(self, solution):
        """This function draws the grid of the solutions that we have found to be
complete.
        """
        answerGrid = [["", "", "", "", ""], ["", "", "", "", ""], [
            "", "", "", "", ""], ["", "", "", "", ""], ["", "", "", "", ""]]
        for row in range(0, 5):
            for col in range(0, 5):
                if answerGrid[row][col] == "":
                    domains = self.gridMakerHelper(row, col, solution)
                    if domains == {}:
                        answerGrid[row][col] = "-"
                    else:
                        if len(self.getCurrentDomainWord("across", row)) == 1:
                            for colIndex in
range(domains["across"]["loc"]["start"]["col"],
domains["across"]["loc"]["start"]["col"] + len(domains["across"]["domain"][0].word)):
                                answerGrid[row][colIndex] =
domains["across"]["domain"][0].word[colIndex -
```

```
domains["across"]["loc"]["start"]["col"]]
                        if len(self.getCurrentDomainWord("down", col)) == 1:
                            for rowIndex in
range(domains["down"]["loc"]["start"]["row"], domains["down"]["loc"]["start"]["row"] +
len(domains["down"]["domain"][0].word)):
                                answerGrid[rowIndex][col] =
domains["down"]["domain"][0].word[rowIndex -

domains["down"]["loc"]["start"]["row"]]

        return answerGrid

    def gridMakerHelper(self, row, col, curDomain):
        """This solution returns the cell domains of the final trials
        """
        domains = {}
        # down
        for location in self.locationOfDownClues:
            wordIndex = -1
            tempCol = self.locationOfDownClues[location]["start"]["col"]
            rowStart = self.locationOfDownClues[location]["start"]["row"]
            rowEnd = self.locationOfDownClues[location]["end"]["row"]
            if (row <= int(rowEnd)) and (row >= int(rowStart)) and (col ==
int(tempCol)):
                wordIndex = row-rowStart
                domains["down"] = {"index": wordIndex, "domain":
self.getCurrentDomainWords(
                    "down", tempCol, curDomain), "loc":
self.locationOfDownClues[location]}
        # across
        for location in self.locationOfAcrossClues:
            wordIndex = -1
            tempRow = self.locationOfAcrossClues[location]["start"]["row"]
            colStart = self.locationOfAcrossClues[location]["start"]["col"]
            colEnd = self.locationOfAcrossClues[location]["end"]["col"]
            if (col <= int(colEnd)) and (col >= int(colStart)) and (row ==
int(tempRow)):
                wordIndex = col-colStart
                domains["across"] = {"index": wordIndex, "domain":
self.getCurrentDomainWords(
                    "across", tempRow, curDomain), "loc":
self.locationOfAcrossClues[location]}
        return domains

    def getCurrentDomainWords(self, acrossDown, index, domain):
        """This function returns words of the cell
```

```
        """
        words = []
        for word in domain[acrossDown][index]:
            if word.active:
                words.append(word)
        return words
```

crossword_gui.py

```
from PyQt5 import QtCore, QtGui, QtWidgets, Qt
from datetime import date
from datetime import datetime

class Ui_MainWindow(object):
    """This is the main window class that contains every widget,grid and answer that
we display to the screen. It was generated using PyQt UI interface, so every change
made in the interface is displayed here in code.
    """
    def setupUi(self, MainWindow, cellNumberArray, cellBlockArray, cluesAcross,
cluesDown, cellAnswerArray, solved):
        MainWindow.setObjectName("LUMOS Puzzle Solver")
        MainWindow.resize(1127, 743)
        MainWindow.adjustSize()
        self.cellNumberArray = cellNumberArray
        self.cellBlockArray = cellBlockArray
        self.cluesAcross = cluesAcross
        self.cluesDown = cluesDown
        self.cellAnswerArray = cellAnswerArray
        self.solved = solved
        self.centralwidget = QtWidgets.QWidget(MainWindow)
        self.centralwidget.setObjectName("centralwidget")
        self.gridLayout = QtWidgets.QGridLayout(self.centralwidget)
        self.gridLayout.setObjectName("gridLayout")
        self.frame = QtWidgets.QFrame(self.centralwidget)
        self.frame.setStyleSheet("background-color: rgb(255, 255, 255);")
        self.frame.setFrameShape(QtWidgets.QFrame.StyledPanel)
        self.frame.setFrameShadow(QtWidgets.QFrame.Raised)
        self.frame.setObjectName("frame")
        self.verticalLayout = QtWidgets.QVBoxLayout(self.frame)
        self.verticalLayout.setObjectName("verticalLayout")
        self.widget_2 = QtWidgets.QWidget(self.frame)
        self.widget_2.setObjectName("widget_2")
        self.horizontalLayout_2 = QtWidgets.QHBoxLayout(self.widget_2)
        self.horizontalLayout_2.setObjectName("horizontalLayout_2")
        self.label = QtWidgets.QLabel(self.widget_2)
        font = QtGui.QFont()
        font.setFamily("Times New Roman")
        font.setPointSize(26)
```

```
        font.setBold(True)
        font.setWeight(75)
        self.label.setFont(font)
        self.label.setObjectName("label")
        self.horizontalLayout_2.addWidget(self.label)
        self.verticalLayout.addWidget(self.widget_2)
        self.line = QtWidgets.QFrame(self.frame)
        self.line.setFrameShape(QtWidgets.QFrame.HLine)
        self.line.setFrameShadow(QtWidgets.QFrame.Sunken)
        self.line.setObjectName("line")
        self.verticalLayout.addWidget(self.line)
        self.frame_2 = QtWidgets.QFrame(self.frame)
        self.frame_2.setFrameShape(QtWidgets.QFrame.StyledPanel)
        self.frame_2.setFrameShadow(QtWidgets.QFrame.Raised)
        self.frame_2.setObjectName("frame_2")
        self.horizontalLayout = QtWidgets.QHBoxLayout(self.frame_2)
        self.horizontalLayout.setSpacing(0)

        self.horizontalLayout.setObjectName("horizontalLayout")
        self.widget = QtWidgets.QWidget(self.frame_2)
        self.widget.setMinimumSize(QtCore.QSize(500, 500))
        self.widget.setStyleSheet("gridline-color: rgb(0, 0, 0);\n"
"background-color: rgb(0, 0, 0);\n"
"border-color: rgb(131, 131, 131);\n"
"border-width: 1;\n"
"border-style: solid;")
        self.widget.setObjectName("widget")
        sizePolicy = QtWidgets.QSizePolicy(QtWidgets.QSizePolicy.Expanding,
QtWidgets.QSizePolicy.Expanding)
        sizePolicy.setHorizontalStretch(0)
        sizePolicy.setVerticalStretch(0)
        sizePolicy.setHeightForWidth(self.widget.sizePolicy().hasHeightForWidth())
        self.widget.setSizePolicy(sizePolicy)
        self.gridLayout_2 = QtWidgets.QGridLayout(self.widget)
        self.gridLayout_2.setContentsMargins(3, 3, 3, 3)
        self.gridLayout_2.setSpacing(0)
        self.gridLayout_2.setObjectName("gridLayout_2")
        self.labels = {}
        self.gridList_1 = {}

        self.generateInitialGrid(MainWindow, self.cellBlockArray, self.widget,
self.gridLayout_2, self.labels, self.gridList_1)
        self.revealAnswers(self.cellAnswerArray, self.labels)

        self.widget_right = QtWidgets.QWidget(self.frame_2)
        self.widget_right.setMinimumSize(QtCore.QSize(500, 500))
        self.widget_right.setStyleSheet("gridline-color: rgb(0, 0, 0);\n"
```

```
"background-color: rgb(0, 0, 0);\n"
"border-color: rgb(131, 131, 131);\n"
"border-width: 1;\n"
"border-style: solid;")
        self.widget_right.setObjectName("widget_right")
        self.widget_right.setSizePolicy(sizePolicy)

        self.gridLayout_right = QtWidgets.QGridLayout(self.widget_right)
        self.gridLayout_right.setContentsMargins(3, 3, 3, 3)
        self.gridLayout_right.setSpacing(0)
        self.gridLayout_right.setObjectName("gridLayout_right")
        self.labels_2 = {}
        self.gridList_2 = {}

        self.generateInitialGrid(MainWindow, self.cellBlockArray, self.widget_right,
self.gridLayout_right, self.labels_2, self.gridList_2)
        self.revealAnswers(self.solved, self.labels_2)

        self.horizontalLayout.addWidget(self.widget)
        self.widget_3 = QtWidgets.QWidget(self.frame_2)
        self.widget_3.setStyleSheet("background-color: rgb(255, 255, 255);")
        self.widget_3.setMinimumSize(500,500)
        self.widget_3.setObjectName("widget_3")
        self.gridLayout_3 = QtWidgets.QGridLayout(self.widget_3)
        self.gridLayout_3.setObjectName("gridLayout_3")
        self.tableWidget_2 = QtWidgets.QTableWidget(self.widget_3)
        self.tableWidget_2.setGeometry(QtCore.QRect(10, 40, 241, 451))
        self.tableWidget_2.setAutoFillBackground(False)
        self.tableWidget_2.setTextElideMode(QtCore.Qt.ElideRight)
        self.tableWidget_2.setShowGrid(False)
        self.tableWidget_2.setGridStyle(QtCore.Qt.NoPen)
        self.tableWidget_2.setCornerButtonEnabled(True)
        self.tableWidget_2.setObjectName("tableWidget")
        self.tableWidget_2.setColumnCount(2)
        self.tableWidget_2.setRowCount(5)
        self.tableWidget_2.horizontalHeader().setVisible(False)
        self.tableWidget_2.horizontalHeader().setMinimumSectionSize(0)
        self.tableWidget_2.horizontalHeader().setDefaultSectionSize(20)
        self.tableWidget_2.horizontalHeader().setStretchLastSection(True)
        self.tableWidget_2.verticalHeader().setVisible(False)
        self.tableWidget_2.verticalHeader().setCascadingSectionResizes(True)
        self.tableWidget_2.verticalHeader().setStretchLastSection(False)
        self.tableWidget_2.verticalHeader().setDefaultSectionSize(60)
        self.tableWidget_2.setWordWrap(True)
        self.gridLayout_3.addWidget(self.tableWidget_2, 1, 0, 1, 1)
        self.gridLayout_3.addWidget(self.tableWidget_2, 1, 0, 1, 1)
        self.tableWidget = QtWidgets.QTableWidget(self.widget_3)
```

```
self.tableWidget.setGeometry(QtCore.QRect(10, 40, 241, 451))
self.tableWidget.setAutoFillBackground(False)
self.tableWidget.setTextElideMode(QtCore.Qt.ElideRight)
self.tableWidget.setShowGrid(False)
self.tableWidget.setGridStyle(QtCore.Qt.NoPen)
self.tableWidget.setCornerButtonEnabled(True)
self.tableWidget.setObjectName("tableWidget")
self.tableWidget.setColumnCount(2)
self.tableWidget.setRowCount(5)
self.tableWidget.setWordWrap(True)
self.tableWidget.verticalHeader().setDefaultSectionSize(60)
self.tableWidget.horizontalHeader().setVisible(False)
self.tableWidget.horizontalHeader().setMinimumSectionSize(0)
self.tableWidget.horizontalHeader().setDefaultSectionSize(20)
self.tableWidget.horizontalHeader().setStretchLastSection(True)
self.tableWidget.verticalHeader().setVisible(False)
self.tableWidget.verticalHeader().setCascadingSectionResizes(True)
self.tableWidget.verticalHeader().setStretchLastSection(False)
self.gridLayout_3.addWidget(self.tableWidget, 1, 1, 1, 1)
self.label_54 = QtWidgets.QLabel(self.widget_3)
font = QtGui.QFont()
font.setFamily("Arial")
font.setPointSize(12)
font.setBold(True)
font.setWeight(75)
self.label_54.setFont(font)
self.label_54.setObjectName("label_54")
self.gridLayout_3.addWidget(self.label_54, 0, 0, 1, 1)
self.label_55 = QtWidgets.QLabel(self.widget_3)
font = QtGui.QFont()
font.setFamily("Arial")
font.setPointSize(12)
font.setBold(True)
font.setWeight(75)
self.label_55.setFont(font)
self.label_55.setObjectName("label_55")
self.gridLayout_3.addWidget(self.label_55, 0, 1, 1, 1)
self.horizontalLayout.addWidget(self.widget_3)
self.horizontalLayout.addWidget(self.widget_right)
self.verticalLayout.addWidget(self.frame_2)
self.label_56 = QtWidgets.QLabel(self.frame)
font = QtGui.QFont()
font.setFamily("Arial")
font.setPointSize(10)
font.setItalic(True)
self.label_56.setFont(font)
```

```
self.label_56.setAlignment(QtCore.Qt.AlignRight|QtCore.Qt.AlignTrailing|QtCore.Qt.Alig
nVCenter)
        self.label_56.setWordWrap(False)
        self.label_56.setObjectName("label_56")
        self.verticalLayout.addWidget(self.label_56)
        self.gridLayout.addWidget(self.frame, 0, 0, 1, 1)
        MainWindow.setCentralWidget(self.centralwidget)
        self.menubar = QtWidgets.QMenuBar(MainWindow)
        self.menubar.setGeometry(QtCore.QRect(0, 0, 1127, 26))
        self.menubar.setObjectName("menubar")
        MainWindow.setMenuBar(self.menubar)
        self.statusbar = QtWidgets.QStatusBar(MainWindow)
        self.statusbar.setObjectName("statusbar")
        MainWindow.setStatusBar(self.statusbar)
        self.retranslateUi(MainWindow)
        QtCore.QMetaObject.connectSlotsByName(MainWindow)


    def retranslateUi(self, MainWindow):
        """This method is also automatically generated in order to encode the text in
the labels that we use, hence it takes a translate function. This function is called
after the initial grids are generated.
        """
        _translate = QtCore.QCoreApplication.translate
        MainWindow.setWindowTitle(_translate("MainWindow", "Lumos Puzzle Solver"))
        self.label.setText(_translate("MainWindow", "The Mini Crossword"))
        self.generateLabelNumbers(MainWindow, self.cellNumberArray, _translate,
self.labels) #Generate numbers that will be displayed on the grid for the first grid.
        self.generateLabelNumbers(MainWindow, self.cellNumberArray, _translate,
self.labels_2) #Generate numbers that will be displayed on the grid for the second
grid.
        __sortingEnabled = self.tableWidget.isSortingEnabled()
        self.tableWidget.setSortingEnabled(False)
        self.tableWidget.setSortingEnabled(__sortingEnabled) #Enable sorting for the
clues so that the first clue starting with 1 is always on top.
        self.label_54.setText(_translate("MainWindow", "ACROSS"))
        self.label_55.setText(_translate("MainWindow", "DOWN"))

        self.generateClues(MainWindow, self.cluesAcross, self.cluesDown, _translate)
        self.label_56.setText(_translate("MainWindow", "LUMOS, {}
{}".format(date.today().strftime("%d/%m/%Y"), datetime.now().strftime("%H:%M:%S"))))

    def generateInitialGrid(self, MainWindow, blackGrids, widgetName, input_grid,
labels, gridList):
        """Generates the initial grid, creates frames and labels that we will fill
later.
```

```
        Args:
            MainWindow ([type]): MainWindow that we will show the grid in.
            blackGrids (List): Information from the backend regarding which grids are
black in todays puzzle.
            widgetName (Widget): Widget that the grid is bound to in the frame.
            input_grid (Dictionary): Grid dictionary that holds every label and every
frame that made up the cells.
            labels (Dictionary): Label dictionary that holds the numbers and the
colors of the grid cells.
        """
        for row in range(5):
            for col in range(5):
                # Establish grid and label names
                gridName = "frame_{}_{}".format(str(row), str(col))
                # Create a grid element
                gridList[gridName] = QtWidgets.QFrame(widgetName)
                # Set the tile color
                if blackGrids[row][col] == '1':
                    gridList[gridName].setStyleSheet("background-color: rgb(0, 0,
0);")
                else:
                    gridList[gridName].setStyleSheet("background-color:
rgb(255,255,255);")

                # Set the frame shape
                gridList[gridName].setFrameShape(QtWidgets.QFrame.StyledPanel)
                # Set the frame shadow
                gridList[gridName].setFrameShadow(QtWidgets.QFrame.Raised)
                # Set object name
                gridList[gridName].setObjectName(gridName)

                # Create the labels for the tile, the first label in the tuple is the
                labels[gridName] = (QtWidgets.QLabel(gridList[gridName]),
QtWidgets.QLabel(gridList[gridName]))

                # Set the styles of the labels
                labels[gridName][0].setGeometry(QtCore.QRect(20, 40, 60, 55))
                labels[gridName][1].setGeometry(QtCore.QRect(5, 10, 31, 31))
                font = QtGui.QFont()

                font.setFamily("Helvetica")
                font.setPointSize(36)
                font.setWeight(40)
                labels[gridName][0].setFont(font)

                font2 = QtGui.QFont()
```

```
                font2.setFamily("Arial")
                font2.setPointSize(20)
                font2.setWeight(30)
                labels[gridName][1].setFont(font2)

                labels[gridName][0].setStyleSheet("border-width: 0; color: rgb(40, 96,
216)")
                labels[gridName][1].setStyleSheet("border-width: 0")
                labels[gridName][0].setAlignment(QtCore.Qt.AlignCenter)
                input_grid.addWidget(gridList[gridName], row, col, 1, 1)

    def generateLabelNumbers(self, MainWindow, gridNumbers, translate, labels):
        """Numbers the grid cells according to today's puzzle. The number information
comes from the backend and it is in list form.
        """
        for row in range(5):
            for col in range(5):
                if gridNumbers[row][col] != '-':
                    gridName = "frame_{}_{}".format(str(row), str(col))
                    labels[gridName][1].setText(translate(str(MainWindow),
gridNumbers[row][col]))

    def generateClues(self, MainWindow, cluesAcross, cluesDown, translate):
        """Places today's clues inside a list in order to display it onto the page.
Clues information come from backend.
        """
        no_row = 0
        clue_row = 0
        no_col = 0
        clue_col = 1
        for across in cluesAcross:
            item_no = QtWidgets.QTableWidgetItem()
            item_no.setTextAlignment(QtCore.Qt.AlignTop)
            self.tableWidget_2.setItem(no_row, no_col, item_no)
            no_row += 1
            item_clue = QtWidgets.QTableWidgetItem()
            item_clue.setTextAlignment(QtCore.Qt.AlignTop)
            self.tableWidget_2.setItem(clue_row, clue_col, item_clue)
            clue_row += 1
            no = across[0]
            clue = across[1]
            item_no.setText(translate(str(MainWindow), no))
            item_clue.setText(translate(str(MainWindow),clue))
        no_row = 0
        clue_row = 0
        no_col = 0
        clue_col = 1
```

```
        for down in cluesDown:
            item_no = QtWidgets.QTableWidgetItem()
            item_no.setTextAlignment(QtCore.Qt.AlignTop)
            self.tableWidget.setItem(no_row, no_col, item_no)
            no_row += 1
            item_clue = QtWidgets.QTableWidgetItem()
            item_clue.setTextAlignment(QtCore.Qt.AlignTop)
            self.tableWidget.setItem(clue_row, clue_col, item_clue)
            clue_row += 1
            no =down[0]
            clue = down[1]
            item_no.setText(translate(str(MainWindow), no))
            item_clue.setText(translate(str(MainWindow),clue))

    def revealAnswers(self, cellAnswerArray, labels):
        """ Displays the letters that corresponds to the answers in the cells.
        """
        for row in range(5):
            for col in range(5):
                if cellAnswerArray[row][col] != '-':
                    gridName = "frame_{}_{}".format(str(row), str(col))
                    labels[gridName][0].setText(cellAnswerArray[row][col])
```

main.py

```
from requests.api import delete
from scraping import Scraping
from crosswordSolver import CrosswordSolver
from Connector import Connector
from newSolver import NewSolver
from PyQt5 import QtCore, QtGui, QtWidgets, Qt
from crossword_gui import Ui_MainWindow
import json


class LUMOSCrosswordSolver:
    def __init__(self):
        self.cellNumberArray = []
        self.cellBlockArray = []
        self.cluesAcross = []
        self.cluesDown = []
        self.cellAnswerArray = []

        self.clues = {"across": {}, "down": {}}
        self.domains = {"across": {}, "down": {}}

    def run(self, demo):
        if demo:
```

```
            # GET CROSSWORD PUZZLE

            nyTimesConnector = Connector(
                "C:\Program Files (x86)/chromedriver.exe")
            nyTimesConnector.connectToPuzzle()
            self.cellNumberArray = nyTimesConnector.cellNumberArray
            self.cellBlockArray = nyTimesConnector.cellBlockArray
            self.cluesAcross = nyTimesConnector.cluesAcross
            self.cluesDown = nyTimesConnector.cluesDown
            self.cellAnswerArray = nyTimesConnector.cellAnswerArray
            self.setClues()
            print("====================\nWEB SCRAPING\n====================")
            webScrapper = Scraping(
                self.clues, self.cellAnswerArray, self.cellNumberArray)
            webScrapper.setDomains()
            print("====================\nSOLVING THE PUZZLE\n====================")
            puzzleSolver = NewSolver(self.cellBlockArray, self.cellNumberArray,
                                     self.cluesDown, self.cluesAcross,
webScrapper.domains)
        else:

            #with open('data.json', 'r') as fp:
            #    data = json.load(fp)
            with open('cellBlockArray.json', 'r') as fp:
                self.cellBlockArray = json.load(fp)
            with open('cellNumberArray.json', 'r') as fp:
                self.cellNumberArray = json.load(fp)
            with open('clueAcross.json', 'r') as fp:
                self.cluesAcross = json.load(fp)
            with open('cluesDown.json', 'r') as fp:
                self.cluesDown = json.load(fp)
            with open('answers.json', 'r') as fp:
                self.cellAnswerArray = json.load(fp)
            print("====================\nWEB SCRAPING\n====================")
            self.setClues()
            webScrapper = Scraping(self.clues, self.cellAnswerArray,
self.cellNumberArray)
            webScrapper.setDomains()
            print("====================\nSOLVING THE PUZZLE\n====================")
            puzzleSolver = NewSolver(self.cellBlockArray,
self.cellNumberArray,self.cluesDown, self.cluesAcross, webScrapper.domains)

        # SAVE
        """
        with open('cellBlockArray.json', 'w') as fp:
            json.dump(self.cellBlockArray, fp,  indent=4)
        with open('cellNumberArray.json', 'w') as fp:
```

```
            json.dump(self.cellNumberArray, fp,  indent=4)
        with open('clueAcross.json', 'w') as fp:
            json.dump(self.cluesAcross, fp,  indent=4)
        with open('cluesDown.json', 'w') as fp:
            json.dump(self.cluesDown, fp,  indent=4)
        with open('data.json', 'w') as fp:
            json.dump(webScrapper.domains, fp,  indent=4)
        """
        # puzzleSolver = CrosswordSolver(self.cellBlockArray,
self.cellNumberArray,self.cluesDown, self.cluesAcross, data)#webScrapper.domains)
        #puzzleSolver = newSolver(cellBlockArray, cellNumberArray,cluesDown,
cluesAcross, webScrapper.domains)

        print("====================\nSOLUTION\n====================")
        for i in puzzleSolver.solvedPuzzle:
            print(i)
        # DRAW GUI
        import sys
        app = QtWidgets.QApplication(sys.argv)
        MainWindow = QtWidgets.QMainWindow()
        ui = Ui_MainWindow()
        ui.setupUi(MainWindow, self.cellNumberArray, self.cellBlockArray,
self.cluesAcross,
                   self.cluesDown, self.cellAnswerArray, puzzleSolver.solvedPuzzle)
        MainWindow.show()
        sys.exit(app.exec_())

    def setClues(self):
        for across in self.cluesAcross:
            self.clues["across"][across[0]] = across[1]
        for down in self.cluesDown:
            self.clues["down"][down[0]] = down[1]


lumos = LUMOSCrosswordSolver()
lumos.run(True)
```