CS 461
Artificial Intelligence

Spring 2021

Project Report

Team Members

Ahmet Feyzi Halaç	21703026
Aybars Altınışık	21601054
Ege Şahin	21702300
Göktuğ Gürbüztürk	21702383
Zeynep Cankara	21703381

Group Nick: RIDDLER

Section: 1

Instructor: Varol Akman

1. Introduction:

As the field of computer science evolves, the field of Artificial Intelligence keeps expanding its boundaries. *Artificial Intelligence* can be defined as the science and engineering of intelligent machines, specifically intelligent programs that can show some level of reasoning. Thus, the Artificial Intelligence methods and algorithms currently applied for solving various tasks in natural language understanding, computer vision, and automation are the use of Artificial Intelligence in a task-specific manner. In the term project, we focused on the crossword solving task, and we developed a program to solve the New York Times Mini puzzle. The task of solving a crossword puzzle can be modelled as a constraint satisfaction problem. What makes this problem especially challenging is constructing the search space for the possible list of answers, requiring knowledge of the language intrinsics and understanding to generate a set of answers in alignment with the puzzle clues. The project has the learning objective of solving the constraint satisfaction problem in crossword puzzles by using methods and algorithms used in Artificial Intelligence. The following sections of the report will be presenting the project description, algorithms and methods from Artificial Intelligence to solve the crossword puzzle problem and our implementation details together with the experimentation and results.

2. Project Description:

The project is about implementing a crossword puzzle solver for the New York Times Mini Puzzle, a 5x5 crossword puzzle first created by Joel Fagliano. The puzzle has five clues down and five clues across, as shown in an example screenshot (Figure 1.) We demonstrated the daily puzzle downloaded from the New York Times website on the left side of the program, where it is filled with the answers of the given day, which is given in the example screenshot (Figure 1.). There are various algorithms for solving crossword puzzles using AI where the majority of them handle the problem as a constraint satisfaction problem. Later we used the original answers scraped from the New York Times website to evaluate the performance of our crossword puzzle solver program. Our puzzle solver program's results from the puzzle UI presented on the right side of the screenshot, where the correct answers highlighted with the colour green (Figure 1.). In the appendix section, we presented the screenshots of the sample 13 New York Times Mini puzzles when we run our puzzle solver program, the screenshots obtained from different days chosen randomly from the archive and excluding our demo day.

3. Literature Research

In the literature, there exist various approaches for solving crossword puzzles with Artificial Intelligence methods and algorithms. Solving crossword puzzles requires an understanding of semantic information present within the crossword puzzle in the form of clues and the use of orthogonal patterns that make up the puzzle constraints (Thanasuan & Mueller, 2014, 1). Thus, in solving the crossword puzzle, it is common to construct the set of possible answers by fetching information from the web by forming queries using the semantic information present within the puzzle clues. Furthermore, constructing a candidate answer list requires a Semantic-based search which is the natural language processing component of the puzzle-solving process. The quality of the candidate answer list depends on the source of the data fetched from the web. The shared resources used to generate candidate answer lists are clue databases, dictionaries, Wikipedia titles, online thesaurus, and using the databases previously being used for crossword puzzles (Thomas & Sangeetha, 2020, 2311).

We checked out the literature on the optimisation methods for solving the constraint satisfaction problem in crossword puzzle solving with AI methods. We found out Dr Fill, a program designed to solve American-style crossword puzzles, using optimisation techniques such as specific variable and value selection heuristics to improve the performance of the answer search significantly (Ginsberg, 2011, 852). Furthermore, Dr Fill uses post-processing the possible candidate answer list again to increase the search performance (Ginsberg, 2011, 852). The Dr Fill program models the crossword puzzle program as a constraint satisfaction problem that consists of variables and set domains for each variable. The variable values make up the possible list of answer candidates and set of constraints (Ginsberg, 2011, 852). The constraints used for checking which values are allowed for specific variables in the puzzle-solving process. The Dr Fill puzzle solver makes use of various heuristics such as assigning weights to the constraints, using discrepancy search and makes use of a value selection heuristic which rely on the projected cost of assigning a score both to the selected candidate variable and to all possible set variables that are sharing typical constraints (Ginsberg, 2011, 883).

4. Data Resources Used

We have used the data sources Wikipedia, Merriam Webster, WordNet to fetch information about a set of possible answers given the puzzle's clues. We pre-processed the clues by dividing them into separate words and searching indicated resources for every part.

For the data retrieval from Merriam Webster Website, we have used its request API to get candidate answers about clues. We searched each word of the clue in the dictionary and took every word in the definitions of them as a candidate answer. On the other hand, from Thesaurus, we searched synonyms and the antonyms of the tokens, namely words of that clue.

For the data retrieval from Wikipedia, we have used the Wikipedia Python library. First, we tokenized the clue and got tokens by looking at Wikipedia via the API specific search method. Also, to increase the variety, we found the matching pages in Wikipedia using the library's searching functionality. Then we get the content of the matching pages if the page exists.

For the data retrieval from WordNet, we have used Python nltk package corpus for the WordNet to get tokens from the clue. WordNet corpus provides synsets, nouns, verbs, adjectives, and adverbs bundled together into sets of cognitive synonyms. Then we iterated over lemmas of the synsets obtained from the clue. We followed the similar constraint checking and post-processing of tokens that we used earlier in Merriam Webster and Wikipedia.

In order to improve the quality of the data obtained from the web, we removed the characters and strings that are not a valid alphanumeric character which includes removing the punctuation marks, tabs and newlines. Also, we removed stopwords like 'the, a, of' from the possible tokens to search these resources with unnecessary tokens. Moreover, we removed answers that do not satisfy the length constraint of the clue.

Finally, we concatenated all candidate answers we obtained from the following three sources, which became our search space for answers.

5. Implementation of the AI Method

As mentioned in the earlier sections, we modelled the crossword puzzle solving as a constraint satisfaction problem. Any state of the puzzle contains information about current constraints and domains. For the initial state, domains contain possible answers for each clue fetched from the resources we mentioned above. However, there is a difference between initially fetched domains and initial state's domains. After fetching, we are shrinking domains for constraints of the crossword. This process allows the search algorithm to try fewer combinations. Our search algorithm is similar to the depth-first search algorithm. The only difference is about selecting and sorting possible next states of the state popped from the queue. The algorithm pops a state from the queue and checks whether it is the goal state or not. If it is a goal state, it ends the search.

On the other hand, it ignores that state and continues to pop another state from the queue if it is stuck. This process allows backtracking, namely, deleting the last answers until another path found. If the current state is neither goal nor stuck state, it gets all the possible next states, append them to the current path and pushes newly generated paths to the beginning of the queue. This process continues until the queue becomes empty.

5.1. Getting next states of a state

Since the goal of our search algorithm is filling the puzzle from a set of domains, shrinking domains in each iteration will reduce the remaining combinations, which means our algorithm will work more efficiently. So, it determines an unfilled clue with the smallest domain when it needs to get the next states. The reason is that there will be fewer possibilities for that clue. After determining which clue will be filled next, it sorts all answers in the domain for the total reduction they cause in all domains. Total reduction means how many answers will be eliminated from other domains by inserting the specified answer to this clue. Again, reducing domains' sizes will decrease the remaining clues' answer combinations. Therefore, it sorts answers in the specified clues so that the state with the most reduction in domains will be selected first from the queue. To conclude, our search algorithm always selects the state with these two approaches, which reduces the domains most, which decreases time to find the goal state.

Also, for every next state, it shrinks domains according to constraints of crossword because a new answer is inserted into the puzzle, and conflicting possible answers must be eliminated from other domains.

5.2. Handling clues whose correct answer could not found in resources

After implementing the initial algorithm, we realized that if a clue does not have a corresponding correct answer in its domain, it will eventually destroy all domains. The reason is that clue with a correct answer cannot find another correct answer in the intersecting clue, which will result in eliminating that correct answer also according to the constraints of the crossword.

So, when all answers in the domain of a clue are tried, and still the goal state cannot be found, it means that the correct answer for that clue couldn't be fetched. This case removes all constraints of that clue and continues to search the algorithm with the remaining clues. With this approach, our search algorithm can find partially filled goal states even if it could not fetch all correct answers from the resources.

5.3 Determining whether a state is the goal or stuck

A state is a goal state if the current answer of every clue is either the correct answer or an empty answer. By empty answer, what is meant is explained in the section above. If all answers in the domain of a clue are tried, and the goal state could not be found, the algorithm inserts an empty answer to that clue.

For the stuck state, if there is any unfilled clue whose domain is empty, it means that that clue cannot be filled in any case. So, the state is stuck, and backtracking required.

6. Conclusion

In conclusion, we were able to implement a crossword puzzle solver for New York Times Mini Crossword. After conducting literature research on crossword puzzles, we modelled the problem as a constraint satisfaction problem. We choose Python programming language due to the accessibility of libraries for Natural Language Processing, especially the NLTK package and ease of data scraping. We have used three resources to construct our set of possible answers using the clue information of the crossword puzzle. Those resources are in alignment with the term project specifications and are Wikipedia, Merriam Webster, WordNet. We performed domain shrinking when we were selecting which clue to fill in first. Domain shrinking helped us remove possible answers that will not satisfy constraints as we continue filling the puzzle. We prioritise the domains with the least possible answer candidates present when filling the crossword puzzle. This optimization narrowed down our search space significantly since it decreased the number of branches in the backtracking routine. We used a depth-first search routine to fill in the answers and backtrack once we stuck in a particular puzzle configuration where we cannot continue matching clues with the possible list of answers within the domain. After we match all clues with a possible answer, we reach a goal state. Afterwards, we evaluated the performance of our AI algorithms with the original puzzle answers by checking how many clues our crossword puzzle solver got correct. Even though the performance of the crossword solver program of the group "RIDDLER" was changing daily due to the complexity of puzzles changing throughout the week, we were pleased with the results we got from the crossword solver. Overall, we found the term project entertaining and educative since it allowed us to practice the AI methods we learnt in the class for constraint satisfaction problems. Furthermore, we enjoyed experimenting with various heuristics of effective search to improve the performance of the algorithm.

4. Appendix

Code

Following are the AI-related source code files from the term project.

```
"""
@Date: 04/05/2021 ~ Version: 1.4
@GroupNick: RIDDLER
@Author: Ahmet Feyzi Halaç
@Author: Aybars Altınışık
@Author: Ege Şahin
@Author: Göktuğ Gürbüztürk
@Author: Zeynep Cankara


@Description: New York Times Mini Crossword Puzzle Solver Term Project

"""

# ------- State.py -------------

from parsePuzzle import parsePuzzle
from Constraints import Constraints
from findAnswer import calculateInitialDomains
from collections import OrderedDict
import copy
from utils import log, getClueFromShortVersion

class State(object):
    # Make puzzleInformation and constraints static variable, since they don't change
for a single puzzle (in every State, this information will be same)
    puzzleInformation = parsePuzzle()

    def __init__(self, domains = False, filledDomains = OrderedDict()):
        if not domains: # Initial state, so initialize domains and shrink it with constraints
            domains = calculateInitialDomains(self.puzzleInformation)
            self.domains = {}
            self.found = {}
            for k, v in domains.items():
                self.domains[k] = v['domain']
```

```
            self.found[k] = v['isTrue']
        self.constraints = Constraints(self.puzzleInformation)

        #Shrink domains with constraints
        self.constraints.shrinkInitialDomains(self.domains,
self.puzzleInformation['answers'])
        for shortVersion, domain in self.domains.items():
            log(getClueFromShortVersion(shortVersion, self.puzzleInformation) + ' -> '
+ ', '.join(domain), newLine=False)
        print()
      else:
        self.domains = domains
      self.lastAnswer = ()
      self.filledDomains = filledDomains

   def __eq__(self, other):
      # For 'in' operation to work correctly, we must implement a custom equality
operator
      return self.domains == other.domains and self.filledDomains ==
other.filledDomains

   def __repr__(self):
      # In the debug panel, states can be visualized better if we represent them by
their filledDomains
      return str(self.filledDomains)

   def __hash__(self):
      # To add a State object to a set, we must implement a hash function
      return hash(str((self.domains, self.filledDomains)))

   def fillDomain(self, clue, answer):
      # This function fills the cells corresponding to this clue and update domains
according to the answer
      self.filledDomains[clue] = answer
      if answer == '':
        self.constraints.removeConstraintsForClue(clue)
      else:
        self.constraints.reduceDomainsWithAnswer(clue, answer, self.domains)

      # State.constraints.reduceDomainsWithAnswer(clue, answer, self.domains)
      self.lastAnswer = (clue, answer)

   def isStuck(self):
```

```
        if self.lastAnswer != () and self.lastAnswer[1] == '' and
self.found[self.lastAnswer[0]]:
            return True

        if self.isGoal(): # If it is goal state, return False immediately
            return False

        for clue, answer in State.puzzleInformation['answers'].items():
            if clue not in self.filledDomains.keys(): # There is a clue which is not
answered yet, so state is not stucked
                return False

        # All clues are answered since all of them are present in filledDomains.
        # However, it is not goal state because of the initial check in this function
        # So, this state is definitely stuck
        return True

    def isGoal(self):
        isGoal = True
        for clue, answer in State.puzzleInformation['answers'].items():
            if clue not in self.filledDomains.keys() or (self.filledDomains[clue] != '' and
self.filledDomains[clue] != answer):
                isGoal = False
                break
        return isGoal

    def getNewState(self, clueAnswerPair):
        # This function creates a deep copy from the current state and fills a clue with
the specified answer in new state. Then return this state
        clue = clueAnswerPair['clue']
        answer = clueAnswerPair['answer']

        state = copy.deepcopy(self)

        # Fill domain with this answer
        state.fillDomain(clue, answer)
        return state

    def getNextStates(self):
        # This function first gets clue with smallest possible answers in its domain and
sorts the answers according to the reduction they provide

        # Get all unfilled clues
```

```
        unfilledClues = list(filter(lambda x: x not in self.filledDomains.keys(),
self.domains.keys()))

        # Get the clue with minimum domain
        clue = min(unfilledClues, key = lambda x: len(self.domains[x]))

        clueAnswerPairs = []
        # For each answer, calculate total reduction
        for answer in self.domains[clue]:
            clueAnswerPairs.append({
                'clue': clue,
                'answer': answer,
                'possibleDomainReduction':
self.constraints.getTotalReductionForAnswer(clue, answer, self.domains,
self.filledDomains)
            })

        # Eliminate impossible clue answer pairs (which will eliminate all possible
answers for another domain)
        clueAnswerPairs = list(filter(lambda x: x['possibleDomainReduction'] !=
-1,clueAnswerPairs))

        # Sort the array with respect to total reduction
        clueAnswerPairs.sort(key= lambda x: x['possibleDomainReduction'])

        # Insert dummy answer, if this is inserted, it means that all answers in this
domain is tried
        clueAnswerPairs.insert(0, {
                'clue': clue,
                'answer': ''
        })

        #For each clue answer pair, get a new state and return the states list
        return list(map(self.getNewState, clueAnswerPairs))


# ------- utils.py -------------

def log(log, newLine=True):
    if type(log) == dict:
        # It is an operation
        parsedClue = log['clue'].replace('d', ' Down').replace('a', ' Across')
        if log['type'] == 'insert':
            if len(log['domain']) == 0:
```

```
                print('There is no candidate remaining for', log['longClue'], '-> removing
constraints for this clue')
            else:
                print('Candidates for ' + log['longClue'] + ': ' + ', '.join(log['domain']) + ' ->
using ' + log['answer'])
        if log['type'] == 'update':
            print('undoing', log['prevAnswer'], '-> now using', log['nextAnswer'], 'for',
log['longClue'])
        if log['type'] == 'delete':
            if log['answer'] == '':
                print('Add constraints for', log['longClue'], 'again because of the
backtracking')
            else:
                print('Delete', log['answer'], 'from', log['longClue'])
    else:
        print(log)
    if newLine:
        print()

def getClueFromShortVersion(shortVersion, puzzleInformation):
    # Input: 1a -> Output: Clue string for 1 Across with specified puzzle
    if 'a' in shortVersion:
        return '\"' + puzzleInformation['acrossClues'][int(shortVersion[0])] + '\"'
    else:
        return '\"' + puzzleInformation['downClues'][int(shortVersion[0])] + '\"'

def getFilledCells(puzzleInformation, filledDomains):
    # Get cell indices for filled clues
    cells = set()
    for domain, answer in filledDomains.items():
        if answer == '':
            continue
        i = -2
        for j in range(0,25):
            if puzzleInformation['cells'][j]['cellNumber'] == int(domain[0]):
                i = j
                break
        if 'a' in domain:
            while True:
                cells.add(i)
                i = i + 1
                if i == 25 or puzzleInformation['cells'][i]['cellNumber'] == -1 or i % 5 == 0:
                    break
        else:
```

```
        while i < 25 and puzzleInformation['cells'][i]['cellNumber'] != -1:
            cells.add(i)
            i = i + 5
    return list(cells)

# ------- searchWordnet.py -------------

import nltk
from nltk.corpus import wordnet #Import wordnet from the NLTK
from createTokens import getSearchedTokens

#nltk.download('wordnet')

def searchWordnet(clue, length):

    # get the tokens according to clue
    tokens_and_best = getSearchedTokens(clue)
    tokens = tokens_and_best[0]

    allAnswers = list()

    # for each token in tokens get synonyms and add them to syn
    for token in tokens:
        for synset in wordnet.synsets(token):
            for lemma in synset.lemmas():
                str = lemma.name().replace("_", "")
                str = str.replace("-", "")
                allAnswers.append(str.upper())    #add the synonyms


    allAnswersLength = len(allAnswers)
    i = 0
    # remove answers which do not satisfiy length constraint
    while(allAnswersLength > i):
        if i+1 < allAnswersLength and len(allAnswers[i]) + len(allAnswers[i+1]) ==
length: # if two words can combine to create new word the combine
            allAnswers.append(allAnswers[i]+allAnswers[i+1])
            allAnswers.pop(i)
            allAnswers.pop(i)
            i = i - 1
            allAnswersLength = allAnswersLength - 2
        elif len(allAnswers[i]) == length - 1 and allAnswers[i][-1] != 'S': # if does not end
with s then add s
            allAnswers.append(allAnswers[i] +'S')
```

```
            allAnswers.pop(i)
            i = i - 1
            allAnswersLength = allAnswersLength - 1
        elif len(allAnswers[i]) == length + 1 and allAnswers[i][-1] == 'S': # if ends with s
then remove s
            allAnswers.append(allAnswers[i][:-1])
            allAnswers.pop(i)
            i = i - 1
            allAnswersLength = allAnswersLength - 1
        elif len(allAnswers[i]) != length:
            allAnswers.pop(i)
            i = i - 1
            allAnswersLength = allAnswersLength - 1
        i = i + 1

    # convert list to set to remmove duplicates then conver set back to list and return
list
    uniqueAnswers = set(allAnswers)
    uniqueAnswerList = list(uniqueAnswers)  #uniqueAnswerList contains each
answer only once

    return uniqueAnswerList



# ------- searchMerriamWebster.py -------------

import requests
import nltk
import json
import re
import string
from createTokens import getSearchedTokens
from nltk.tokenize import word_tokenize

#Parser for Webster URL response
def iter_webster(d):
    answers = []
    if not isinstance(d, list):
        return answers
    for x in d:
        if isinstance(x, str):
            answers.append(x)
        else:
```

```
            answers = answers + x['shortdef']
    return answers

#Parser for Thesaurus URL response
def iter_thesaurus(d):
    answers = []
    if not isinstance(d, list):
        return answers
    for x in d:
        if isinstance(x, str):
            answers.append(x)
        else:
            for syn in x['meta']['syns']:
                answers = answers + syn
            for ant in x['meta']['ants']:
                answers = answers + ant
    return answers

def searchMerriamWebster(clue, length):

    #Get the tokens for the specified clue
    tokens_and_best = getSearchedTokens(clue)
    tokens = tokens_and_best[0]

    webster = []
    thesaurus = []
    results = []
    allAnswers = []

    for token in tokens:
        webster_url = "http://dictionaryapi.com/api/v3/references/collegiate/json/" +
token +"?key=28fc3ab5-65ce-49ed-8878-6b66eddf8ef5"
        thesaurus_url = "http://dictionaryapi.com/api/v3/references/thesaurus/json/" +
token + "?key=33936e00-efa4-47d5-8ef9-b897f7db33d8"

        response = json.loads(requests.get(thesaurus_url).text) #loading the content of
the Thesaurus webpage
        thesaurus = iter_thesaurus(response)

        response = json.loads(requests.get(webster_url).text) #loading the content of
the Collegiate webpage
        webster = iter_webster(response)
```

```
        results = thesaurus + webster #Merging the results of the Saurus and dictionary
parts

        #Unnecessary chars and strings are removed from the results and then
tokenize each result in the results list.
        for result in results:
            result = re.sub(r'{\w+}|[^a-zA-Z]', '', result)
            result = result.replace("\n", " ")
            result = result.replace("\t", " ")
            result = result.replace("_", "")
            result = result.replace("-", "")
            punctuationFree = result.translate(str.maketrans('', '', string.punctuation))
            punctuationFree = punctuationFree.upper()
            allAnswers =  allAnswers + word_tokenize(punctuationFree)

    allAnswersLength = len(allAnswers)
    i = 0
    #Remove answers which do not satisfy length constraint and checking other
syntax constraints
    while(allAnswersLength > i):
        if i+1 < allAnswersLength and len(allAnswers[i]) + len(allAnswers[i+1]) ==
length: # if two words can combine to create new word the combine
            allAnswers.append(allAnswers[i]+allAnswers[i+1])
            allAnswers.pop(i)
            allAnswers.pop(i)
            i = i - 1
            allAnswersLength = allAnswersLength - 2
        elif len(allAnswers[i]) == length - 1 and allAnswers[i][-1] != 'S': # if does not end
with s then add s
            allAnswers.append(allAnswers[i] +'S')
            allAnswers.pop(i)
            i = i - 1
            allAnswersLength = allAnswersLength - 1
        elif len(allAnswers[i]) == length + 1 and allAnswers[i][-1] == 'S': # if ends with s
then remove s
            allAnswers.append(allAnswers[i][:-1])
            allAnswers.pop(i)
            i = i - 1
            allAnswersLength = allAnswersLength - 1
        elif len(allAnswers[i]) == length + 1 and allAnswers[i][-2:] == 'ED': # if ends with
d then remove d
            allAnswers.append(allAnswers[i][:-1])
            allAnswers.pop(i)
            i = i - 1
```

```
            allAnswersLength = allAnswersLength - 1
        elif len(allAnswers[i]) != length:
            allAnswers.pop(i)
            i = i - 1
            allAnswersLength = allAnswersLength - 1
        i = i + 1

     #Convert list to set to remove duplicates then conver set back to list and return
list
    uniqueAnswers = set(allAnswers)
    uniqueAnswerList = list(uniqueAnswers)  #uniqueAnswerList contains each
answer only once
    return uniqueAnswerList




# ------- searchWikipedia.py -------------

import nltk
import wikipedia
import string
import re
import warnings
from createTokens import getSearchedTokens
from nltk.tokenize import word_tokenize

warnings.catch_warnings()
warnings.simplefilter("ignore")

def searchWikipedia(clue, length):

    #Get the tokens for the specified clue
    tokens_and_best = getSearchedTokens(clue)
    best_token = tokens_and_best[1]
    tokens = tokens_and_best[0]

    results = []

    """ Searching wikipedia by using library for each token in
        the clue and add the result to results list. Try to get
        the matched page with the specified token.
    """
    for token in tokens:
        search_results = wikipedia.search(token)
```

```
        results = results + search_results
        if token == best_token:
            best_result = search_results[0]

    #Regulating the results obtained from the webpage and tokenize them
    try:
        page = wikipedia.page(best_result)
        content = page.content
        words = word_tokenize(content)
        results.extend(words)
    except:
        print("Page not found in wiki!\n")

    #Unnecessary chars and strings are removed from the results and then tokenize
each result in the results list.
    allAnswers = []
    lenResults = len(results)
    for i in range(lenResults):
        results[i] = re.sub(r'[0-9]+|[^a-zA-Z ]', '', results[i])
        results[i].replace("_", "")
        results[i].replace("-", "")
        punctuationFree = results[i].translate(str.maketrans('', '', string.punctuation))
        punctuationFree = punctuationFree.upper()
        possibleAnswers =  word_tokenize(punctuationFree)
        allAnswers = allAnswers + possibleAnswers # allAnswers (list) may includde
same answer more than once


    #Applying the length constraint to the each word
    allAnswersLength = len(allAnswers)
    i = 0
    while(allAnswersLength > i):
        if len(allAnswers[i]) != length:
            allAnswers.pop(i)
            i = i - 1
            allAnswersLength = allAnswersLength - 1
        i = i + 1

    #Convert list to set to remove duplicates then conver set back to list and return list
    uniqueAnswers = set(allAnswers)
    uniqueAnswerList = list(uniqueAnswers)  #uniqueAnswerList contains each
answer only once
    return uniqueAnswerList
```

```
# ------- search.py -------------

from State import State
from collections import deque, OrderedDict
import copy
from utils import log, getClueFromShortVersion

def calculateOperations(prevState, nextState):
    if prevState is None:
        return []

    prevList = list(prevState.filledDomains.items())
    nextList = list(nextState.filledDomains.items())
    if len(nextState.filledDomains) > len(prevState.filledDomains): # New answer is
inserted
        return [{
            'type': 'insert',
            'clue': nextList[len(nextState.filledDomains) - 1][0],
            'answer': nextList[len(nextState.filledDomains) - 1][1],
            'domain': prevState.domains[nextList[len(nextState.filledDomains) - 1][0]],
            'longClue': getClueFromShortVersion(nextList[len(nextState.filledDomains) -
1][0], nextState.puzzleInformation),
            'filledDomains': prevState.filledDomains
        }]

    if len(nextState.filledDomains) == len(prevState.filledDomains) and
len(nextState.filledDomains) != 0: # Last answer is changed
        return [{
            'type': 'update',
            'clue': nextList[len(nextState.filledDomains) - 1][0],
            'prevAnswer': prevList[len(prevState.filledDomains) - 1][1],
            'nextAnswer': nextList[len(nextState.filledDomains) - 1][1],
            'longClue': getClueFromShortVersion(nextList[len(nextState.filledDomains) -
1][0], nextState.puzzleInformation),
            'filledDomains': prevState.filledDomains
        }]

    if len(nextState.filledDomains) < len(prevState.filledDomains): # Backtrace. Delete
items from prevState one by one starting from the end
        i = len(prevState.filledDomains) - 1
        operations = []
        while i >= len(nextState.filledDomains):
```

```
            operations.append({
                'type': 'delete',
                'clue': prevList[i][0],
                'answer': prevList[i][1],
                'longClue': getClueFromShortVersion(prevList[i][0],
nextState.puzzleInformation),
                'filledDomains': nextState.filledDomains
            })
            i = i - 1
        operations.append({
            'type': 'update',
            'clue': nextList[i][0],
            'prevAnswer': prevList[i][1],
            'nextAnswer': nextList[i][1],
            'longClue': getClueFromShortVersion(nextList[i][0],
nextState.puzzleInformation),
            'filledDomains': nextState.filledDomains
        })
        return operations

    return []

def search(initialState, handleOperation):

    if initialState.isGoal():
        return [initialState]

    currentPath = [initialState]

    queue = deque([currentPath])

    prevState = None

    while queue:
        visited = set()
        currentPath = queue.popleft()
        currentState = currentPath[len(currentPath) - 1]

        # Calculate operation and call handleOperation for each one
        for operation in calculateOperations(prevState, currentState):
            handleOperation(operation)
            log(operation, operation['type'] != 'delete')

        prevState = currentState
```

```
        # If state is goal state, return that state
        if currentState.isGoal():
            handleOperation({'type': 'goal', 'filledDomains': currentState.filledDomains})
            log('Goal state of the puzzle is found!')
            return currentPath

        # If state is stuck, just continue to next path
        if currentState.isStuck():
            log('Puzzle is stuck! Start backtracing', newLine=False)
            continue


        for state in currentPath:
            visited.add(state)

        nextStates = currentState.getNextStates()

        for nextState in nextStates:
            if nextState in visited:
                continue

            tempPath = copy.deepcopy(currentPath)
            tempPath.append(nextState)
            queue.insert(0, tempPath)

# ------- getPossibleAnswers.py -------------

from searchMerriamWebster import searchMerriamWebster
from searchWikipedia import searchWikipedia
from searchWordnet import searchWordnet

def getPossibleAnswers(clue, length):

    wikipediaAnswers = searchWikipedia(clue, length)    # get answers of wikipedia
    merriamAnswers = searchMerriamWebster(clue, length) # get answers of merriam
webster
    wordnetAnswers = searchWordnet(clue, length)        # get wordnet answers

    allAnswers = wikipediaAnswers + merriamAnswers + wordnetAnswers # merge all
answer lists

    return allAnswers
```

```
# ------- findAnswer.py -------------

from getPossibleAnswers import getPossibleAnswers
from utils import log
import string
import os.path
from os import path

def getAnswersForClue(clue, length):
   log('Fetching answers for clue: ' + clue, newLine=False)

   punctuationFreeClue = clue.translate(str.maketrans('', '', string.punctuation))
   filename = punctuationFreeClue + '.txt'
   if path.exists(filename):
      f = open(filename, "r")
      possible_answers_str = f.read()
      possible_answers = possible_answers_str.split(',')
   else:
      possible_answers = getPossibleAnswers(clue, length)
      f = open(filename, "w")
      f.write(','.join(possible_answers))
      f.close()
   log('Fetched answers: ' + ', '.join(possible_answers))
   return possible_answers


def determineSuccessfulFetch(key, isAcross, puzzleInformation, alternatives):
   if isAcross:
      return puzzleInformation['answers'][str(key) + 'a'] in alternatives
   else:
      return puzzleInformation['answers'][str(key) + 'd'] in alternatives

def getLengthOfClueAnswer(key, isAcross, puzzleInformation):
   for i in range(0, len(puzzleInformation['cells'])):
      if puzzleInformation['cells'][i]['cellNumber'] == key:
         count = 0
         if isAcross:
            while True:
               i = i + 1
               count = count + 1
               if i == 25 or puzzleInformation['cells'][i]['cellNumber'] == -1 or i % 5 == 0:
                  break
         else:
            while i < 25 and puzzleInformation['cells'][i]['cellNumber'] != -1:
```

```
            i = i + 5
            count = count + 1
        return count

def calculateInitialDomains(puzzleInformation):
   log('Fetching initial domains from internet')
   domains = {}
   for key, value in puzzleInformation['acrossClues'].items():
      domains[str(key) + 'a'] = {}
      domains[str(key) + 'a']['domain'] = getAnswersForClue(value,
getLengthOfClueAnswer(key, True, puzzleInformation))
      domains[str(key) + 'a']['isTrue'] = determineSuccessfulFetch(key, True,
puzzleInformation, domains[str(key) + 'a']['domain'])

   for key, value in puzzleInformation['downClues'].items():
      domains[str(key) + 'd'] = {}
      domains[str(key) + 'd']['domain'] = getAnswersForClue(value,
getLengthOfClueAnswer(key, False, puzzleInformation))
      domains[str(key) + 'd']['isTrue'] = determineSuccessfulFetch(key, False,
puzzleInformation, domains[str(key) + 'd']['domain'])

   log('All domains are fetched')
   return domains

# ------- createTokens.py -------------

import nltk
import string
from nltk.corpus import stopwords
from nltk.tokenize import word_tokenize


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

"""This function creates tokens to be searched on the web-sites"""

def getSearchedTokens(clue):

   best_token = clue
   good_tokens = clue.split('"')[1::2] # find good tokens by searching among
quotation marks
   if len(good_tokens) != 0:
      best_token = good_tokens[0]
```

```
    punctuationFree = clue.translate(str.maketrans('', '', string.punctuation))

    tokens = word_tokenize(punctuationFree)            # split the punctiuation free clue
to tokens (list)
    tokens = [w for w in tokens if not w.lower() in stopWords]  # remove stopwordss
from tokens

    if best_token not in tokens:
        tokens.append(best_token)

    return [tokens, best_token]

# ------- Constraints.py -------------

from utils import log

class Constraint(object):
    def __init__(self, acrossClue, acrosIndex, downClue, downIndex):
        self.acrossClue = acrossClue
        self.acrossIndex = acrosIndex
        self.downClue = downClue
        self.downIndex = downIndex

    def getReductionCountForAnswer(self, clue, answer, domains, filledDomains):
        # Determine how many answers this clue,answer pair reduces
        count = 0
        if clue == self.acrossClue and self.downClue not in filledDomains.keys():
            downChars =  list(map(lambda x: x[self.downIndex],
domains[self.downClue]))
            for char in downChars:
                if answer[self.acrossIndex] != char:
                    count = count + 1
        elif clue == self.downClue and self.acrossClue not in filledDomains.keys():
            acrossChars = list(map(lambda x: x[self.acrossIndex],
domains[self.acrossClue]))
            for char in acrossChars:
                if answer[self.downIndex] != char:
                    count = count + 1

        return count

    def applyConstraint(self, clue, answer, domains):
        # Apply constraints for specified answer (Reduce domains accordingly)
        i = 0
```

```
        if clue == self.acrossClue:
            while i < len(domains[self.downClue]):
                if answer[self.acrossIndex] != domains[self.downClue][i][self.downIndex]:
                    domains[self.downClue].remove(domains[self.downClue][i])
                else:
                    i = i + 1
        else:
            while i < len(domains[self.acrossClue]):
                if answer[self.downIndex] != domains[self.acrossClue][i][self.acrossIndex]:
                    domains[self.acrossClue].remove(domains[self.acrossClue][i])
                else:
                    i = i + 1

class Constraints(object):
    def __init__(self, puzzleInformation):
        self.constraints = self.generateConstraints(puzzleInformation)

    def generateConstraints(self, puzzleInformation):
        log('Generating constraints according to puzzle information')
        result = []
        for acrossKey in puzzleInformation['acrossClues'].keys():
            for i in range(0, len(puzzleInformation['cells'])):
                if puzzleInformation['cells'][i]['cellNumber'] == acrossKey:
                    acrossIndex = 0
                    while True:
                        result.append(self.findDownClueMatch(puzzleInformation, i,
acrossKey, acrossIndex))
                        i = i + 1
                        acrossIndex = acrossIndex + 1
                        if i == 25 or puzzleInformation['cells'][i]['cellNumber'] == -1 or i % 5 ==
0:
                            break
                    break
        return result

    def findDownClueMatch(self, puzzleInformation, i, acrossKey, acrossIndex):
        for downKey in puzzleInformation['downClues'].keys():
            for j in range(0, len(puzzleInformation['cells'])):
                if puzzleInformation['cells'][j]['cellNumber'] == downKey:
                    downIndex = 0
                    while j < 25 and puzzleInformation['cells'][j]['cellNumber'] != -1:
                        if i == j:
                            return Constraint(str(acrossKey) + 'a', acrossIndex, str(downKey) +
'd', downIndex)
```

```
                j = j + 5
                downIndex = downIndex + 1
            break

    def findConstraintsForClue(self, clue):
        result = []
        if 'a' in clue:
            result = filter(lambda constraint: constraint.acrossClue == clue,
self.constraints)
        else:
            result = filter(lambda constraint: constraint.downClue == clue,
self.constraints)
        return list(result)

    def getTotalReductionForAnswer(self, clue, answer, domains, filledDomains):
        total = 0
        for constraint in self.findConstraintsForClue(clue):
            reduction = constraint.getReductionCountForAnswer(clue, answer, domains,
filledDomains)
            if reduction == -1:
                # One of the constraints indicates that filling this clue with specified answer
will eliminate all answers for a different domain
                return -1
            total = total + reduction
        return total

    def reduceDomainsWithAnswer(self, clue, answer, domains):
        for constraint in self.findConstraintsForClue(clue):
            constraint.applyConstraint(clue, answer, domains)

    def removeConstraintsForClue(self, clue):
        for constraint in self.findConstraintsForClue(clue):
            self.constraints.remove(constraint)

    def shrinkInitialDomains(self, domains, clues):
        log('Shrinking domains according to crossword constraints. Updated Domains:',
newLine=False)
        while True:
            allConstraintsAreSatisfied = True
            for constraint in self.constraints:
                acrossChars = list(map(lambda x: x[constraint.acrossIndex],
domains[constraint.acrossClue]))
                downChars =  list(map(lambda x: x[constraint.downIndex],
domains[constraint.downClue]))
```

```
        i = 0
        while i < len(domains[constraint.acrossClue]):
            if domains[constraint.acrossClue][i][constraint.acrossIndex] not in
downChars and domains[constraint.acrossClue][i] != clues[constraint.acrossClue]:

domains[constraint.acrossClue].remove(domains[constraint.acrossClue][i])
                allConstraintsAreSatisfied = False
            else:
                i = i + 1

        i = 0
        while i < len(domains[constraint.downClue]):
            if domains[constraint.downClue][i][constraint.downIndex] not in
acrossChars and domains[constraint.downClue][i] != clues[constraint.downClue]:

domains[constraint.downClue].remove(domains[constraint.downClue][i])
                allConstraintsAreSatisfied = False
            else:
                i = i + 1
    if allConstraintsAreSatisfied:
        break
```

Screenshots

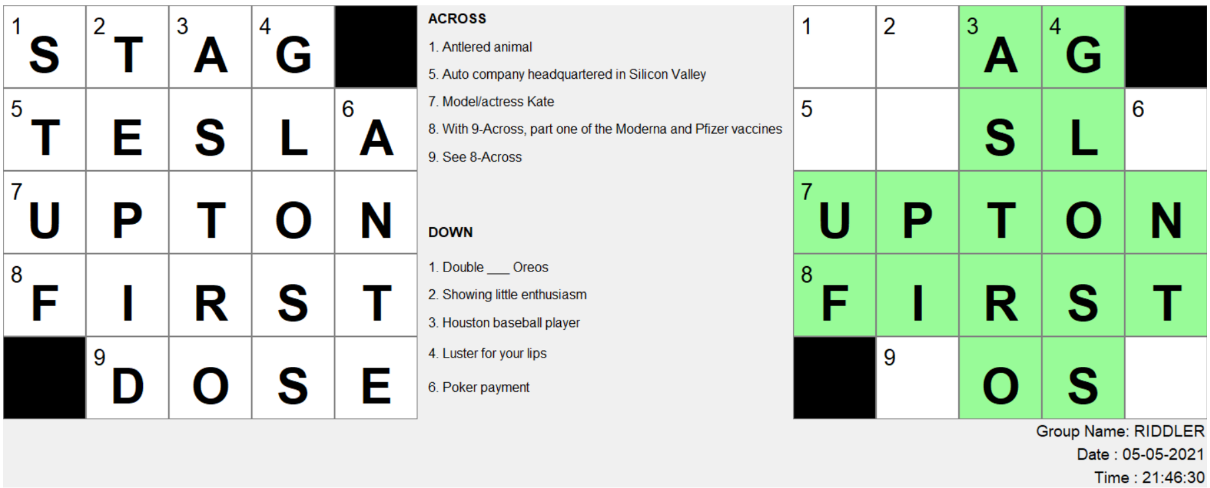

Figure 1. Screenshot of the puzzle solver program for date 05-05-2021

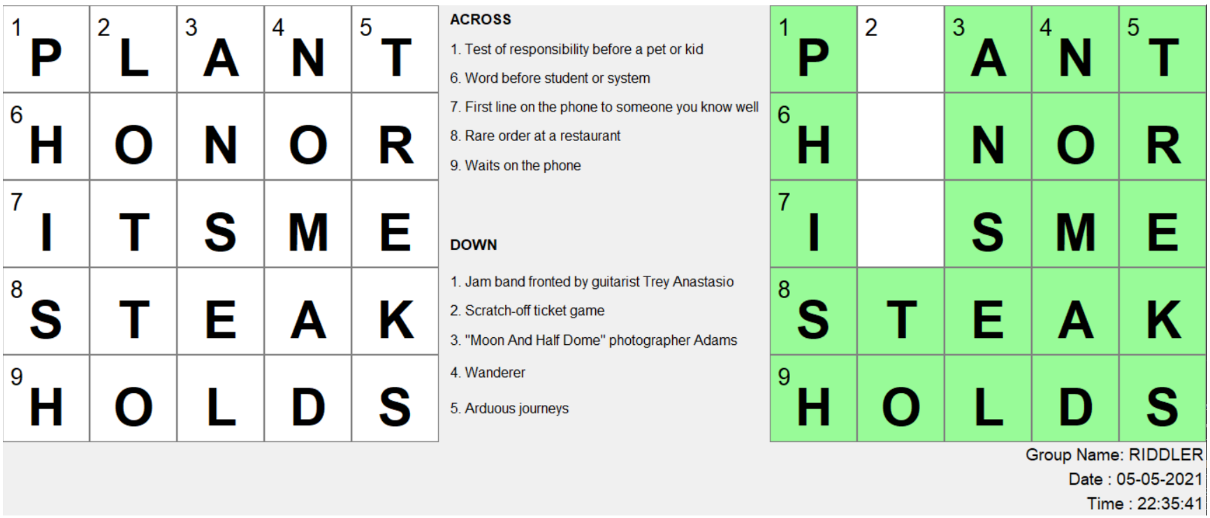

Figure 2. Screenshot of the puzzle solver program for date 26-04-2021

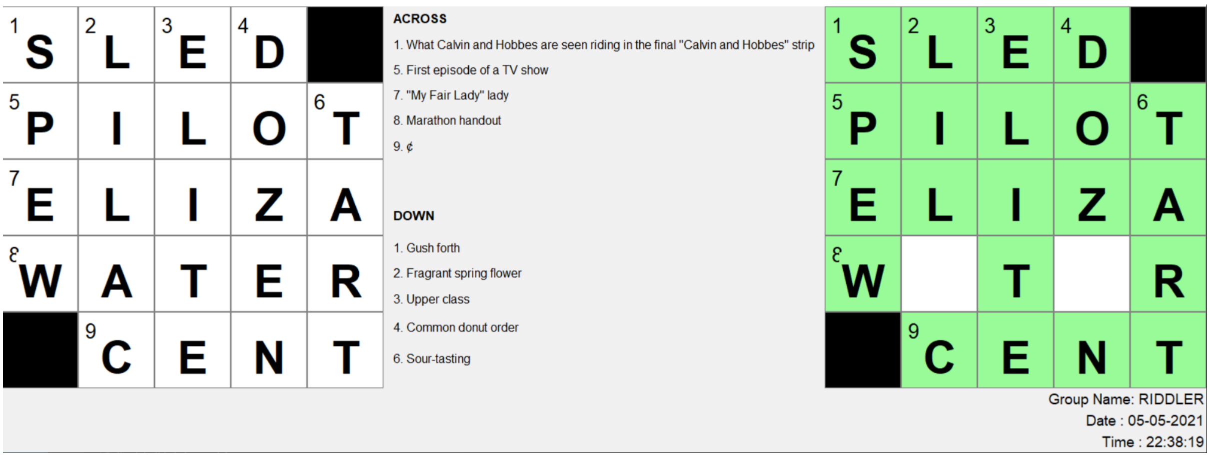

Figure 3. Screenshot of the puzzle solver program for date 28-04-2021

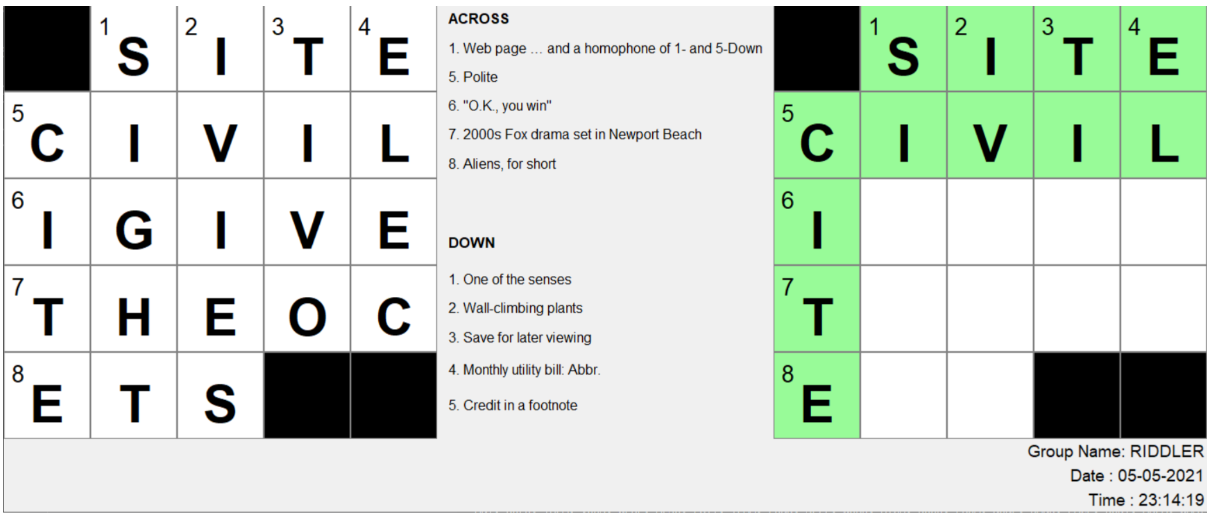

Figure 4. Screenshot of the puzzle solver program for date 29-04-2021

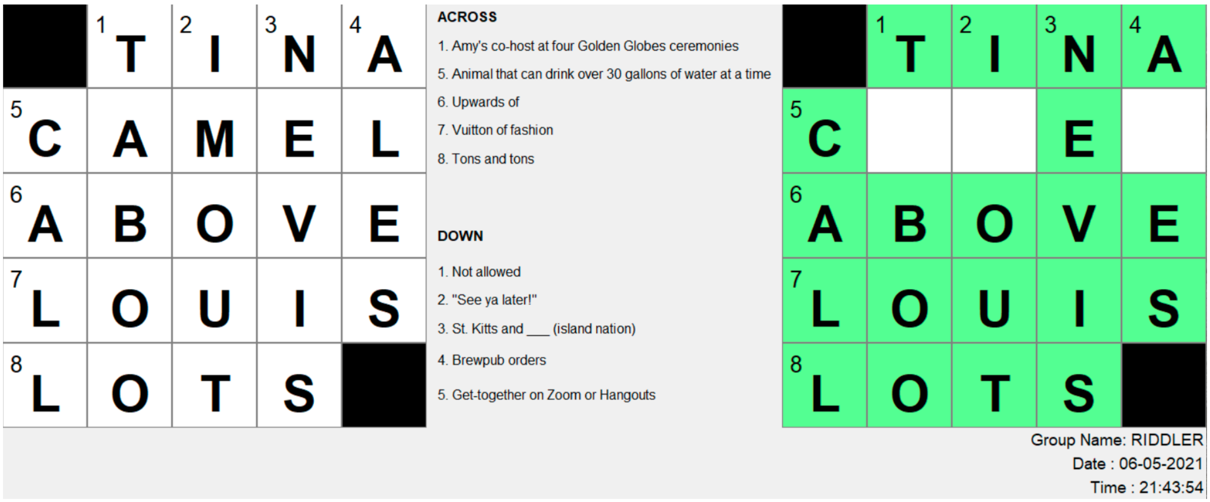

Figure 5. Screenshot of the puzzle solver program for date 06-05-2021

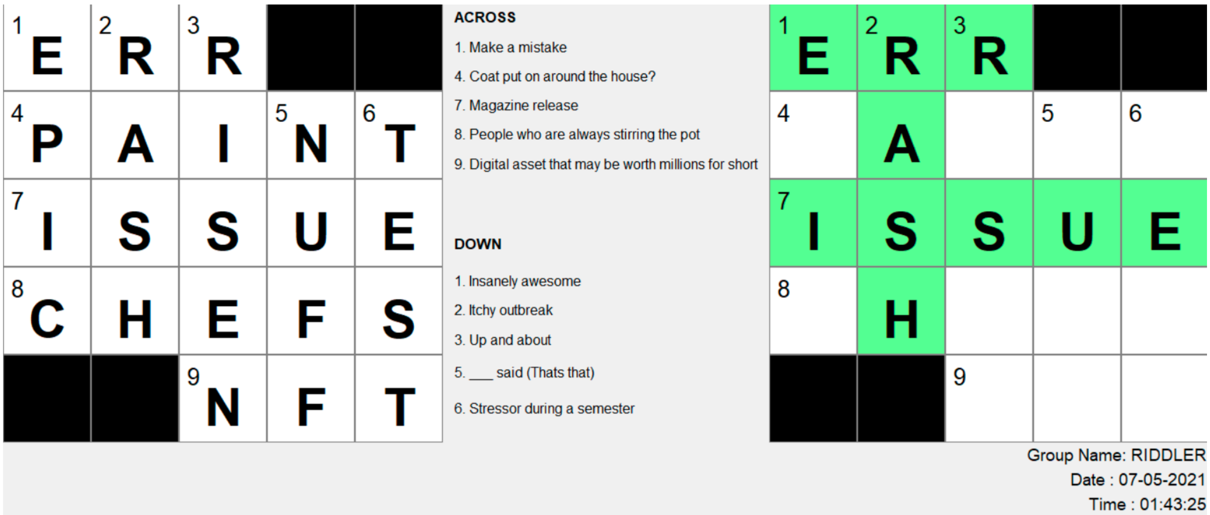

Figure 6. Screenshot of the puzzle solver program for date 27-04-2021

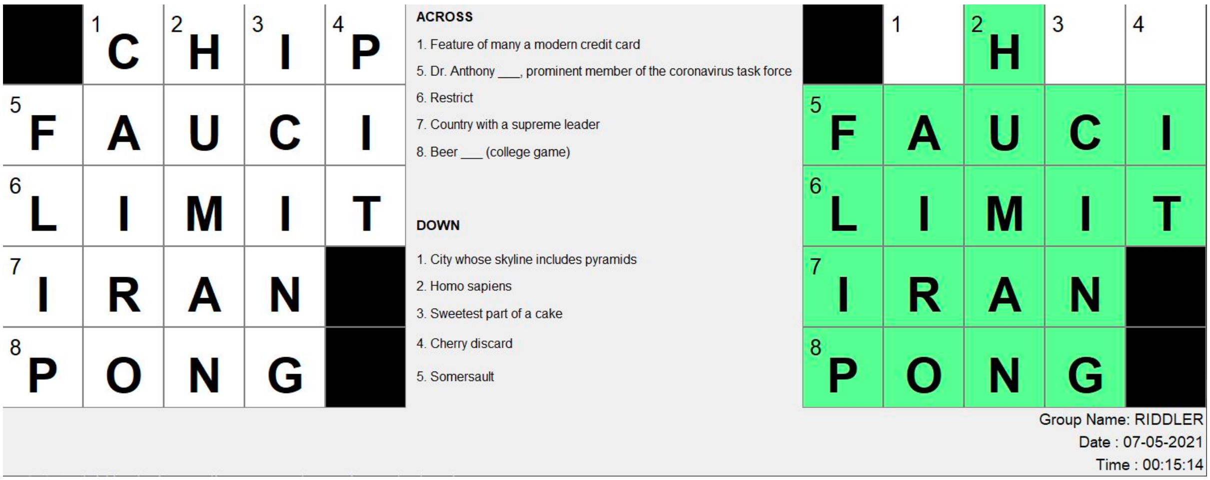

Figure 7. Screenshot of the puzzle solver program for date 01-04-2020

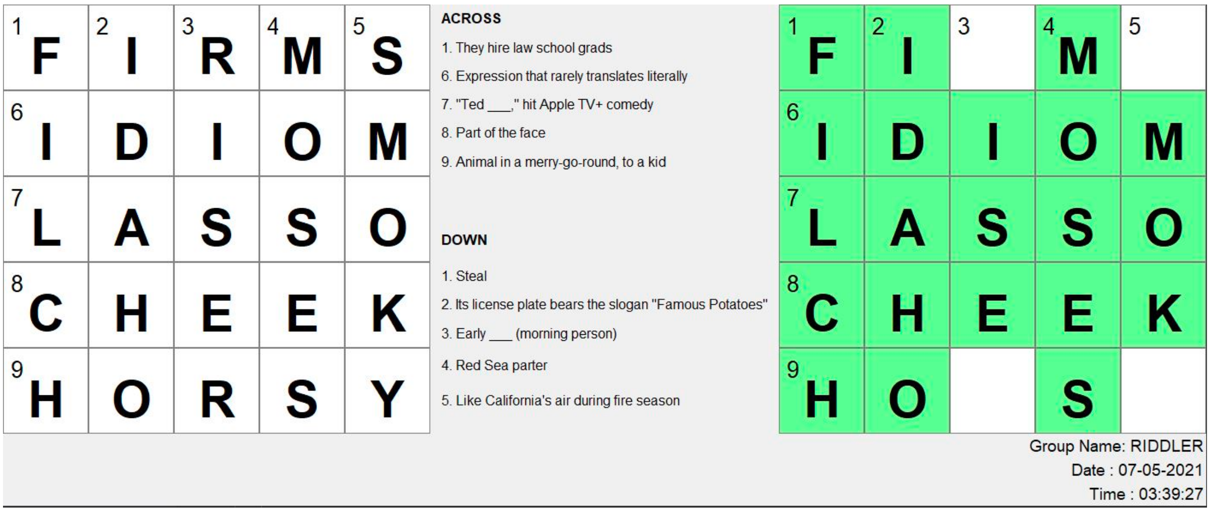

Figure 8. Screenshot of the puzzle solver program for date 01-04-2021

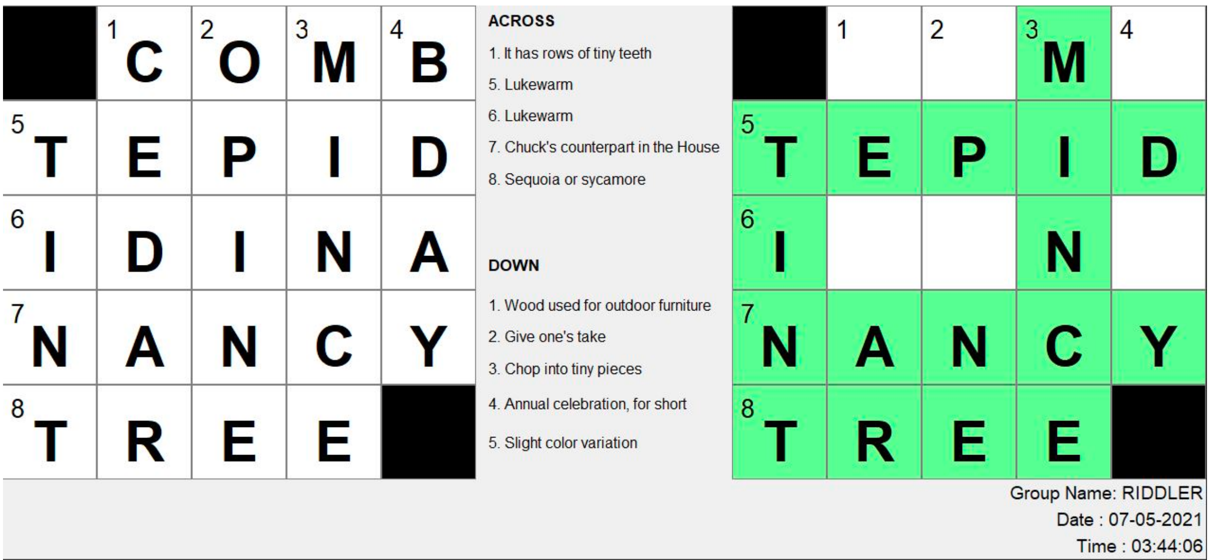

Figure 9 Screenshot of the puzzle solver program for date 02-04-2021

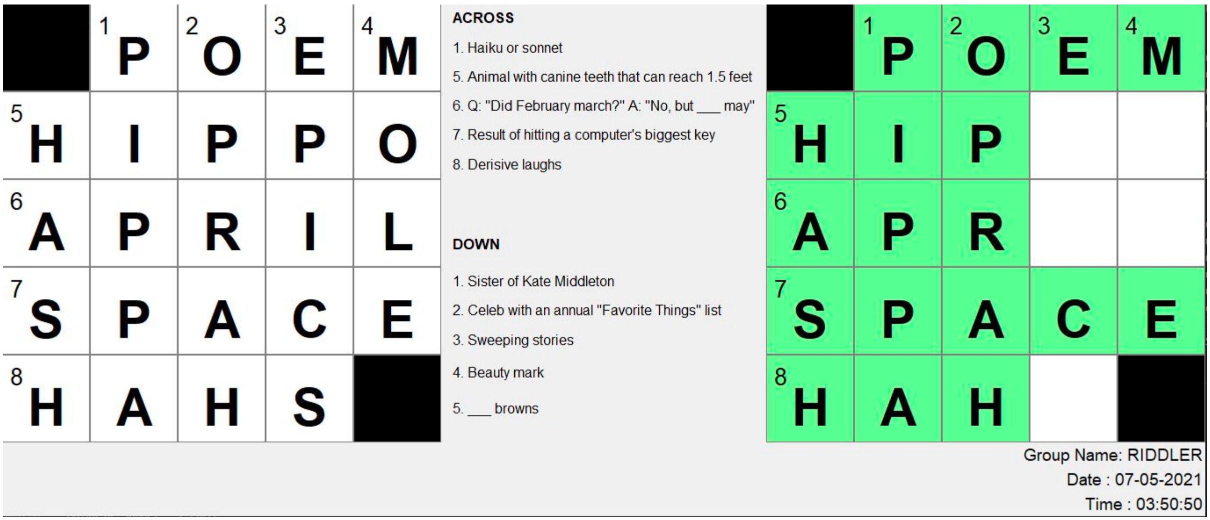

Figure 10. Screenshot of the puzzle solver program for date 06-04-2021

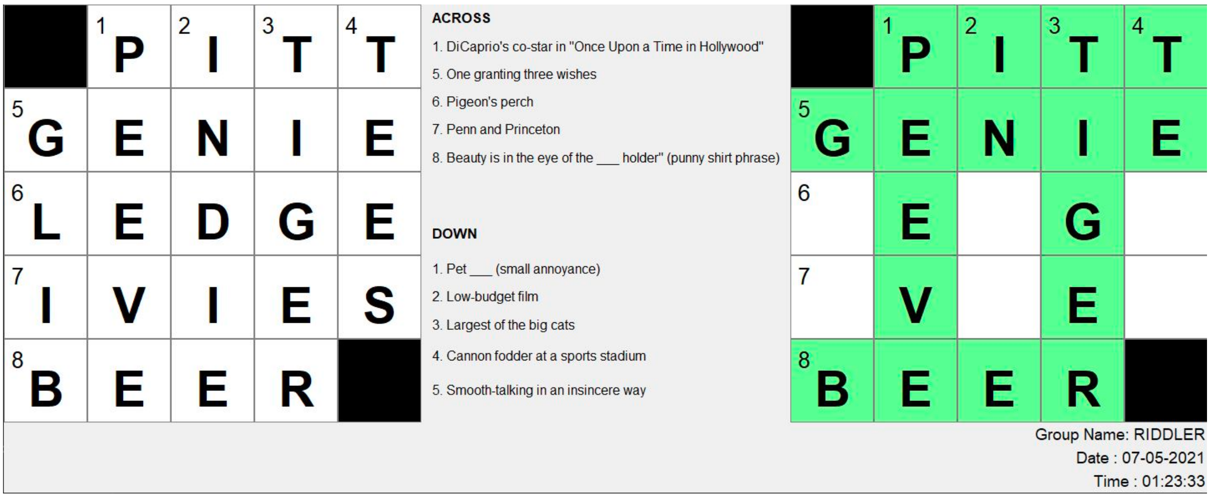

Figure 11. Screenshot of the puzzle solver program for date 11-04-2021

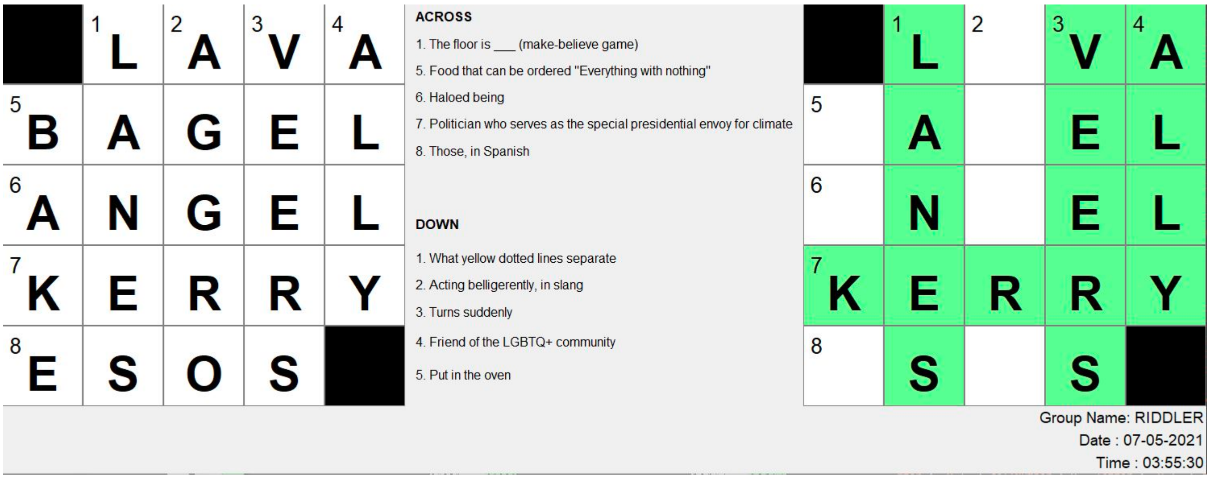

Figure 12. Screenshot of the puzzle solver program for date 07-04-2021

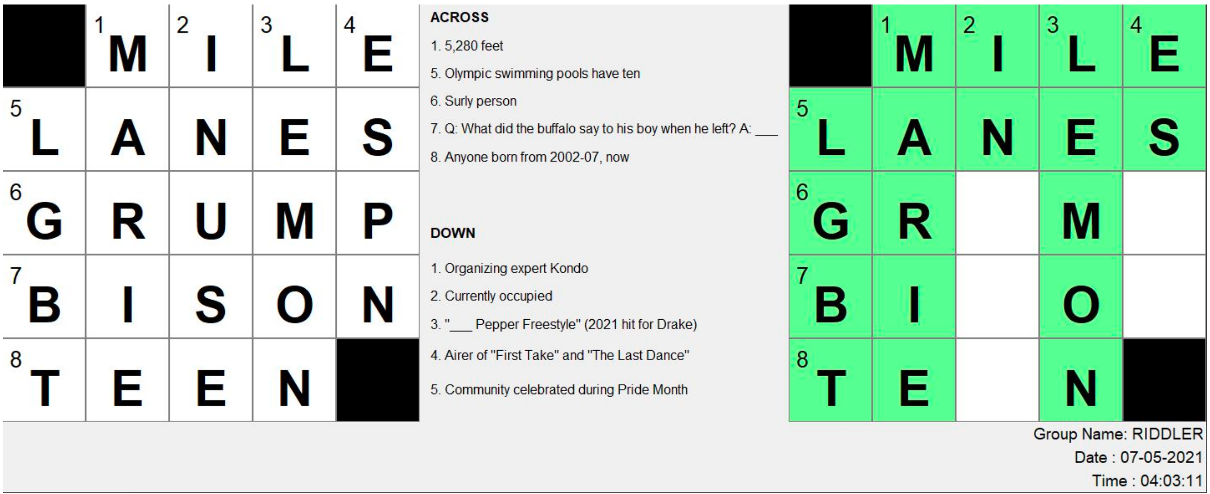

Figure 13. Screenshot of the puzzle solver program for date 20-04-2021

Bibliography

Ginsberg, M. L. (2011, December). Dr.Fill: Crosswords and an Implemented Solver for Singly Weighted CSPs. *Journal of Artificial Intelligence Research*, *42*, 851-886. https://arxiv.org/pdf/1401.4597.pdf

Thanasuan, K., & Mueller, S. T. (2014, September 11). Crossword expertise as recognitional decision making: an artificial intelligence approach. *Frontiers in Psychology*, *5*. https://www.researchgate.net/publication/266950933_Crossword_Expertise_as_Recognitional_Decision_Making_An_Artificial_Intelligence_Approach

Thomas, A., & Sangeetha, S. (2020). Towards a Semantic Approach for Candidate Answer Generation in Solving Crossword Puzzles. *Procedia Computer Science*, *171*, 2310-2315. https://www.sciencedirect.com/science/article/pii/S1877050920312424

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

Word Count = 2082