Bilkent University

Department of Computer Engineering

CS 461 - Artificial Intelligence

Term Project Report

CROSSWIND

Section: 1

Contributing Members:

Melisa Taşpınar	21803668
Yiğit Gürses	21702746
Ömer Ünlüsoy	21702136
Zeynep Berfin Gökalp	21601956

Table of Contents

1 Introduction

Crossword is a type of puzzle, offering clues with word plays for humans to come up with answers that fills a grid. To have any chance of success with such a puzzle, the solver is required to have some knowledge about popular culture, history, and language (a.k.a. common sense). Moreover, for a given clue, the reasoning ability to select a single reasonable answer from many possible options that fits into the grid is crucial [1]. If a set of candidate words are given, a computer program can easily perform such a task using the constraints of the puzzle. However, due to the tricky nature of crossword clues, it is very hard for a program to come up with a reasonable set of candidates. These features of crossword puzzles not only make them engaging and fun for humans, but also make them a great challenge to test the prowess of artificial intelligence techniques [2]. “Information retrieval and set elimination”, “database search”, and “machine learning” are several approaches for solving crossword puzzles [1]. However, using a combination of them to get a better overall performance should also be possible.

An approach that utilizes NLP could be considered more true to the spirit of crossword puzzles than a brute force approach based on only constraint satisfaction. Advancements in the field of NLP enable new and smarter approaches for candidate generation. In [3], Jobin et. al. uses Lesk score and path similarity to rank all words in wordnet in order to get a set of candidate words. According to their results, over half the time, the 1st ranked word was the correct answer. This shows that such an approach can be very successful. However, they stopped their work at the candidate generation stage and did not make a complete crossword solver.

In this project, the aim was to create an AI based program that solves the Newyork Times mini crossword puzzle designed by Joel Fagliano. This is a 5x5 puzzle that has 5 across and 5 down clues and their corresponding entries. The entries usually have many crossings which creates lots of constraints. This makes the puzzle more approachable to AI based programs.

Our program searches several sources on the internet to retrieve possible answers for clues and selects a set of answers from them. The online sources used were merriam-webster's dictionary and thesaurus, and Wikipedia. We also downloaded Wordnet and used it as an offline resource. Constraint satisfaction and depth first search were used in the solution process. In the following sections, the approaches used throughout the project will be described and the results will be evaluated.

2 Description of the Implementation

As stated earlier, the main aim of this project is to come up with answers for Newyork Times mini crossword puzzle and maximize the number of entries guessed correctly. To achieve this, there are several stages that need to be completed successfully. First of all, the puzzle data needs to be scraped from the NYT website. Then, using the puzzle data, candidate words for each clue need to be generated. These candidates should be the required length, be relevant to the clue and hopefully include the correct answer. Then, the number of candidates must be reduced using constraint satisfaction in order to have a reasonable time complexity and reduce unnecessary computations. After the reduction is complete, possible solutions from the remaining candidates need to be generated. This was accomplished using DFS and constraint satisfaction as will be explained later on. Finally, after the possible solutions are generated, the optimal solution should be selected and displayed on the screen.

Before the solution stage which includes most of the AI aspects of the program, candidate words need to be generated in the first stage. First step of candidate generation is scraping words from several sources using the given clues. In this stage, we used a custom method of scoring candidates. These scores were later used in the solution stage. The sources we used for this stage were merriam-webster, wikipedia and wordnet. The clues are searched as a whole in wikipedia, however for merriam-webster and wordnet we split the clues and search individual words. Otherwise, it is not possible to get reasonable sets of candidates from them except for

very short clues. When scraping words from these sources, we also give them a score based on two factors. For all sources, the closer the word is to the top of the page, the higher the score it has. We assumed that more relevant words come up before irrelevant words in general. For example, in a wikipedia page, the word we are looking for is most likely contained in the summary section at the top. For wikipedia, the page we found the word in also affects its score. We check several pages that are relevant to the clue, and here we assumed that the pages that were recommended at the top were the most relevant ones. All scrapers score the words they generate individually according to the following formula: “score = (word_count - index) / word_count” where word_count is the total number of words scraped and index is the index of the given word starting from the top of the page.

After the scraping process is complete, we process the acquired words to obtain our final candidate sets. First of all, all non-alphabetic characters are removed, stopwords are eliminated and all characters are made lowercase. Then, for each word we find its synsets in wordnet and add the synonyms to the candidate set with a score penalty (score of original word * 0.5). Then, all words that are not the required length for the given entry are eliminated. The remaining words give us the finalized candidate sets. We take the union of these finalized sets from different sources and if a word was scraped from multiple sources, we add the scores given by those sources. Here, the idea is that if a word is scraped by multiple sources, it is more likely to be the actual answer.

After the candidate generation stage is completed, we pass the final candidates to the solver and start the solution process. A depth first search tree will be used in order to generate possible solutions. However, before starting the search, an elimination of candidates based on constraint satisfaction is done in order to reduce time complexity. For each crossing/constraint in the puzzle, the pair of candidate sets that are related to the given constraint is compared. During this comparison, if in one candidate set, there exists a word such that none of the words in the other set can satisfy the constraint, we count that as a violation and add 1 to that word's violation count. After checking all

constraints and obtaining all violation counts, we remove all words that have a violation count above or equal to “k” from the candidate sets. We repeat this process until the iteration where no words are removed. In the final implementation, we selected “k” as 2 since setting it to 1 made the program remove the correct answers most of the time. Setting it higher than 2 makes the constraint reduction virtually useless as most of the words will remain after reduction. After this reduction, to further reduce the time complexity, we sort the remaining candidates based on their scores obtained from the scrapers and select the best “n” candidates for each clue. Here, “n” can grow depending on how fast the computer is, how the candidate sets are and how efficient the program is. In our implementation, we used 30.

Now that the unnecessary candidates have been eliminated, the root node is constructed and the search process begins. The states of the nodes in the tree are represented as “selections”. A selection is either empty or a single word, and each Node has a corresponding selection for each clue. When generating a child, for each selection that has not been made in the parent (empty selection), each candidate available for that selection corresponds to a new child. We iterate through these candidates and set them as selections for the new children. However, there is a very important condition that eliminates some of the candidates from being used for generation of new children. After expanding the root node, we only select candidates that satisfy at least some constraint. This prevents the tree from generating branches that have all across selections and no down selections, or vice versa. Note the depth of a Node in this tree corresponds to the number of selections made in that node. The root node with a depth of 0 has no selections. Whenever a new child is constructed, a reduction of candidates based on its current selections is made. This is a similar process to the previously mentioned constraint satisfaction method, however the constraints are only checked between a selection and a candidate set instead of two candidate sets. This allows the program to consider more options as it progresses in the tree. If the previous reduction method was used here, many branches - possibly including the ones that lead to the correct solution - could be lost.

Using DFS, we search through the whole tree and keep a list of leaves encountered. In the implementation of the DFS tree, a queue structure was used. We also keep a dictionary of previously seen nodes so that we do not expand the same node twice which would be unnecessary computation. First, the root node is added to the queue. Then, while the queue is not empty the following steps are taken in each iteration. One node is popped from the end of the queue, and its children are constructed. Non-leaf children which were not seen before (checked from the dictionary) are pushed to the beginning of the queue and added to the seen dictionary. Leaves are held in another dictionary to be used in the next stage.

Once all leaves are found and there is no node left to expand, we progress to the final stage. In this stage, we find the leaf with the highest score and return it as the proposed solution. The score of a leaf is calculated by summing up the scores of its selected words (the scores were obtained during the scraping stage as mentioned). Note that this score is highly correlated with the depth of a node but lower depth nodes can still have higher scores than higher depth nodes. After the proposed solution is acquired, the results are displayed on the screen.

3 Conclusion

In this project, we implemented an end-to-end program that scrapes the NYT mini-crossword puzzle, scrapes candidate answers from various sources and then proposes a solution to the puzzle. One of the most challenging aspects of this task was to generate a reasonable number of candidate words that is also likely to contain the correct answer for a given clue. Increasing the number of candidates we scraped made it more likely for the candidate sets to include the correct answers. However, as a side effect multiple solutions that did not include the correct answers but satisfied more constraints then the desired solution exist in the results of our search algorithm. This made it very hard to select the desired solution among the many options. To combat this, we tried scoring the words as explained previously, and using those scores for

elimination of candidates and solutions. However, the results were still not satisfactory for us.

Our approach to scoring words was not the most sophisticated but it did improve the results significantly. An NLP approach such as the one in [2] could have been used to further improve the scoring process.

Overall, this project gave us an idea of what is possible to accomplish using simple AI techniques. This opens up new opportunities especially combined with the machine learning techniques we have been seeing in other courses. We learned different approaches for solving puzzles using AI and cemented the knowledge we gained from the course. This whole process was challenging but also quite informative.

4 Screenshots

The proposed program's outputs are presented here, Note that the date on the screenshot is the time the screenshot was taken, not the date of the puzzle.

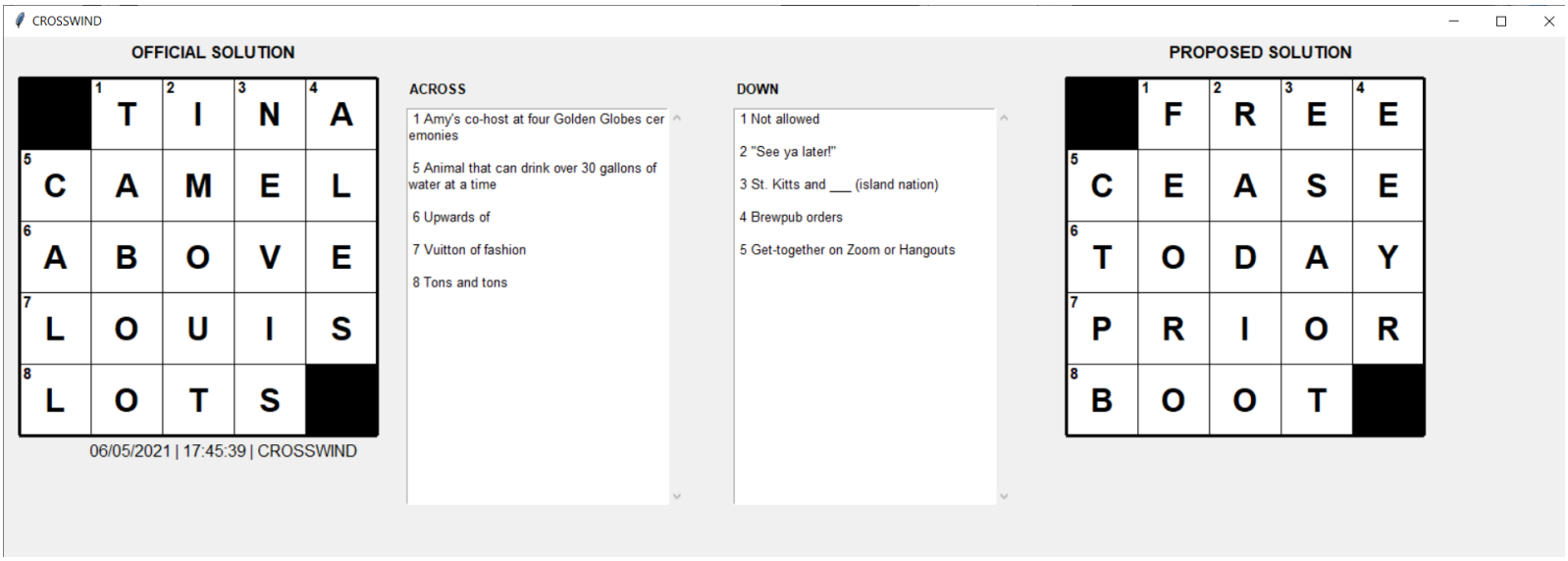

Figure 1: Sample Output

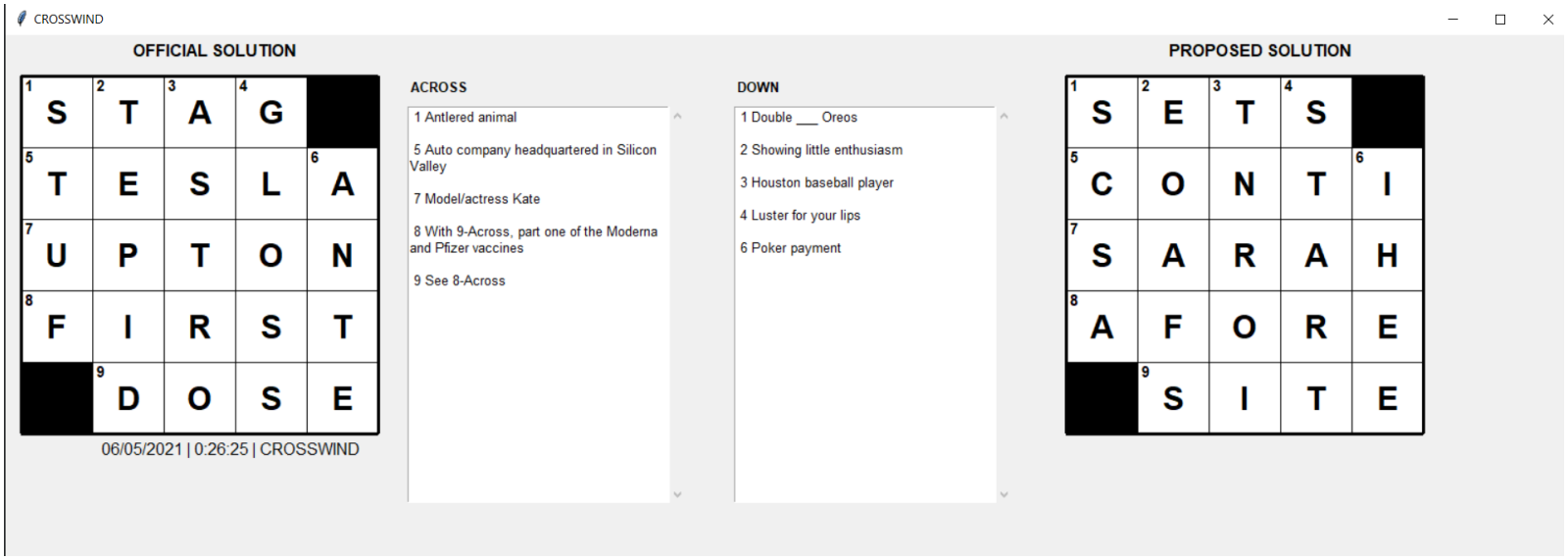

Figure 2: Sample Output

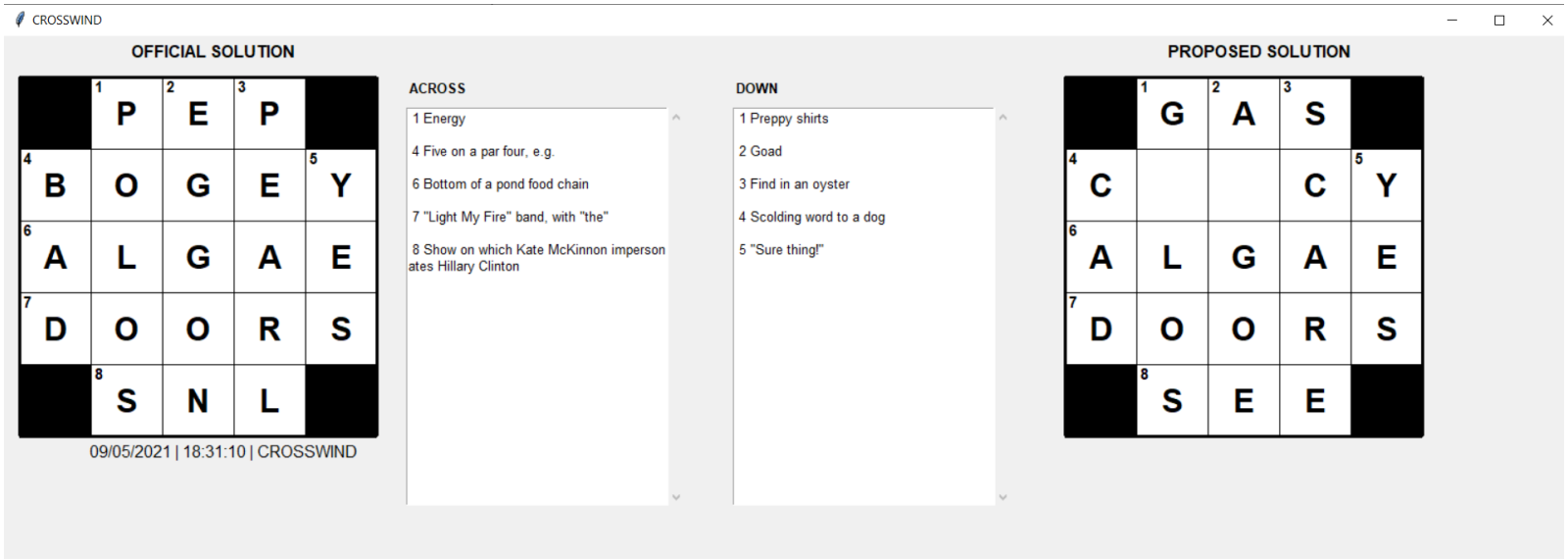

Figure 3: Sample Output

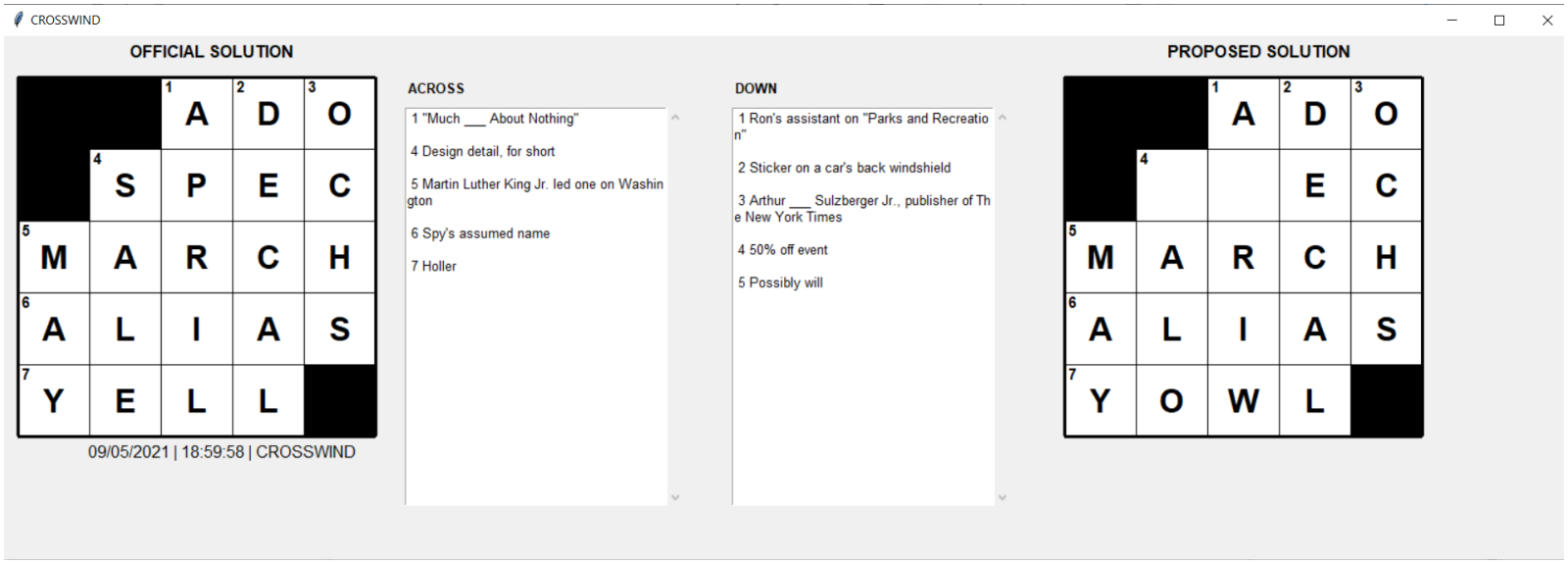

Figure 4: Sample Output

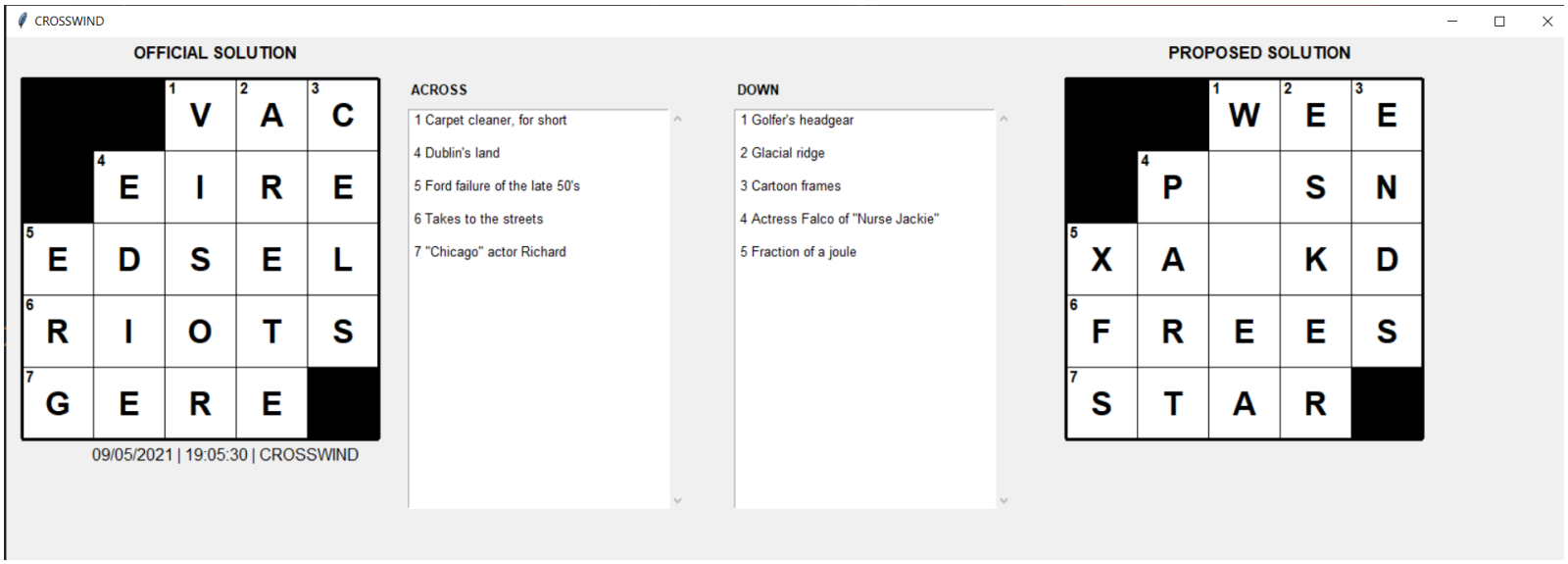

Figure 5: Sample Output

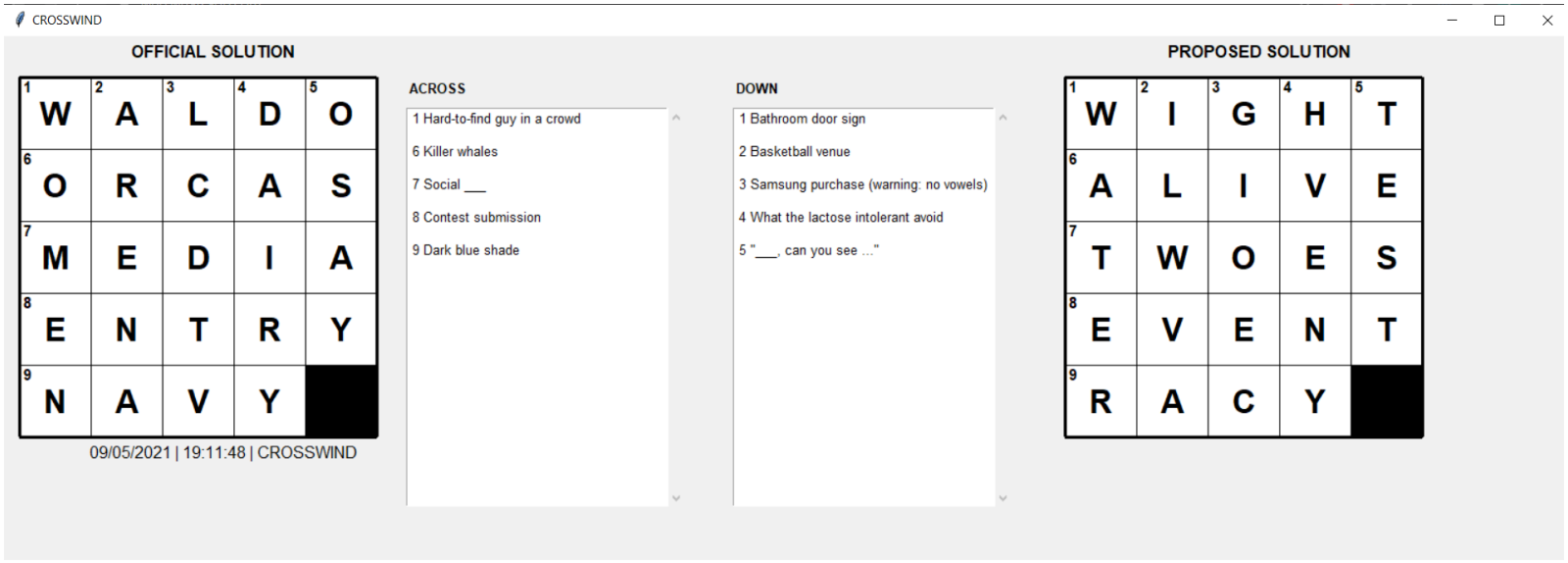

Figure 6: Sample Output

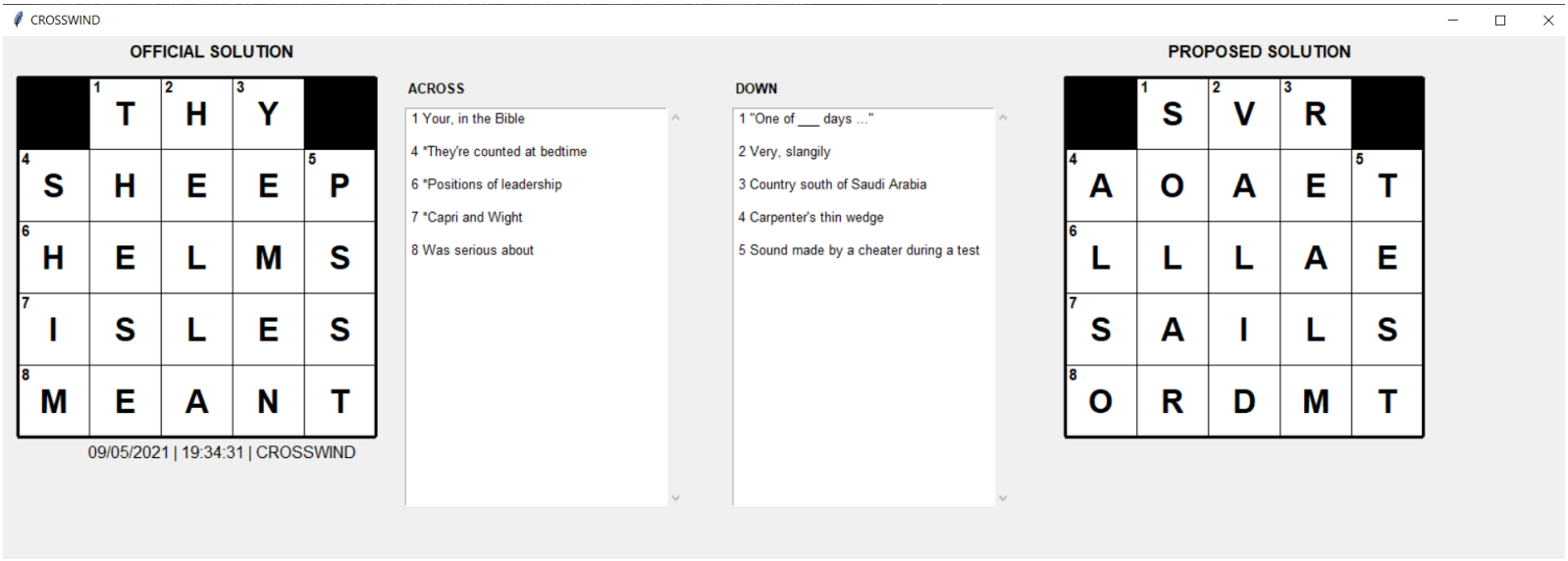

Figure 7: Sample Output

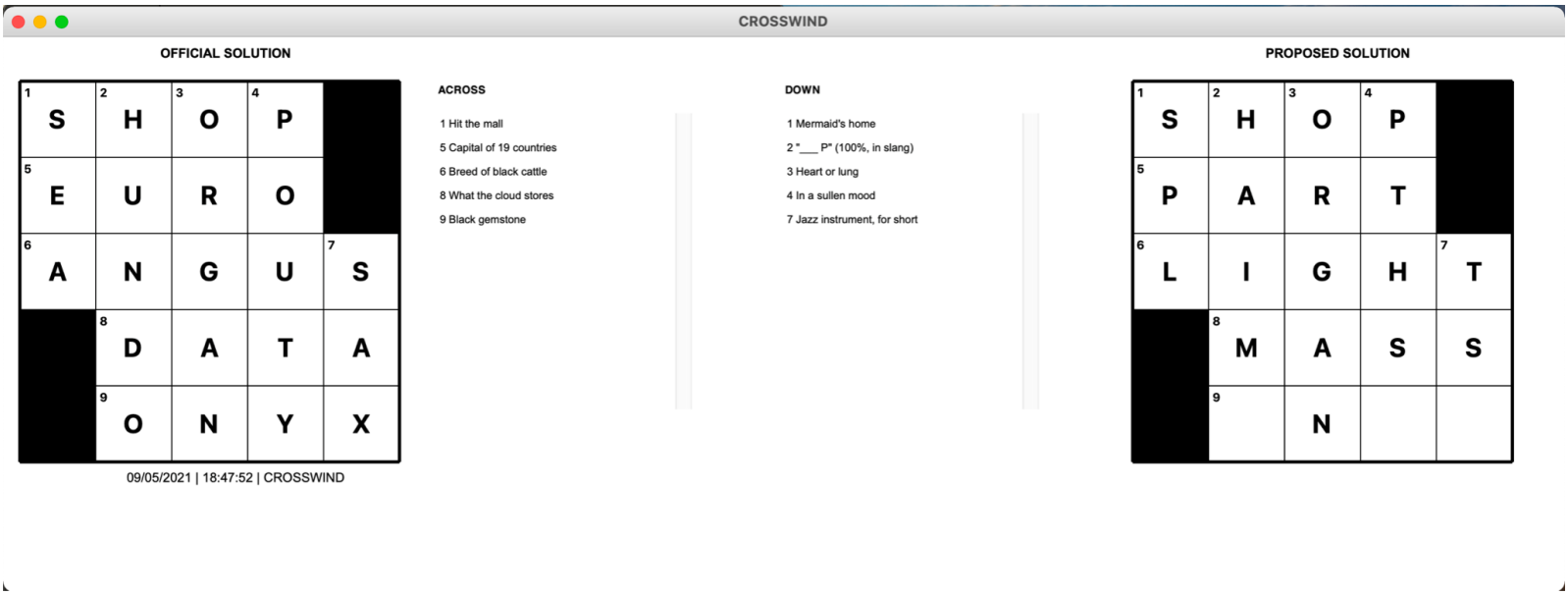

Figure 8: Sample Output

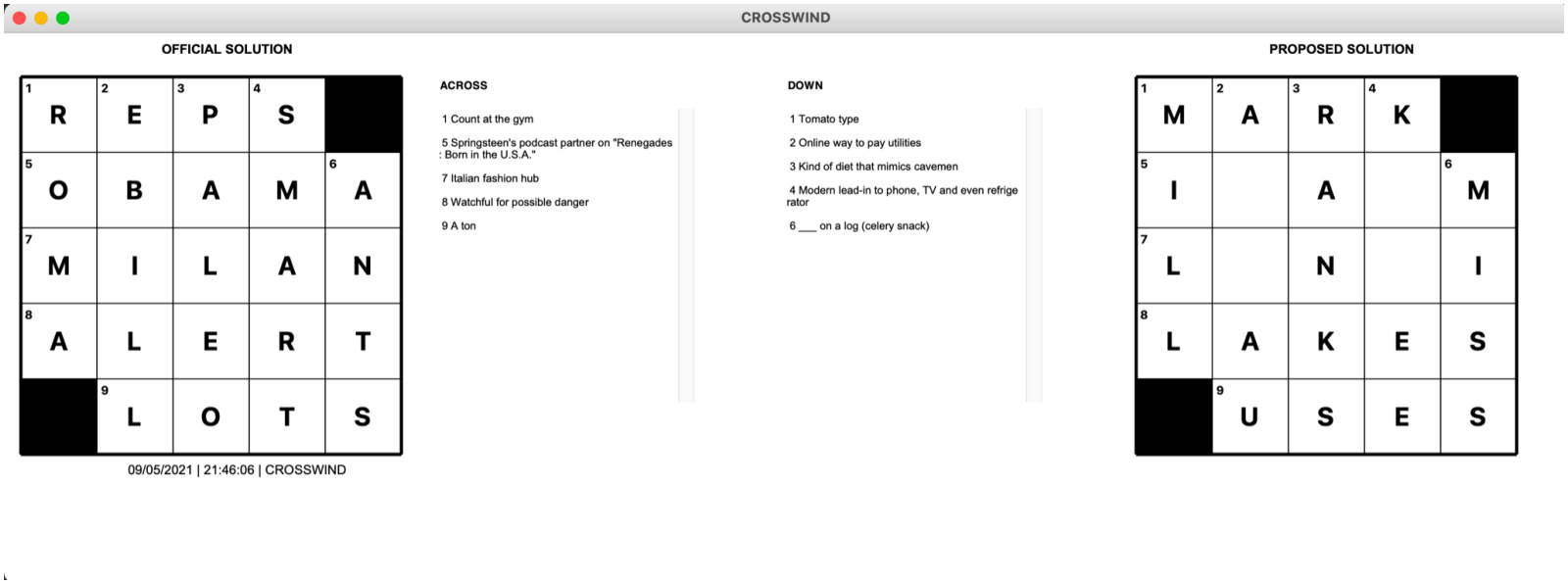

Figure 9: Sample Output

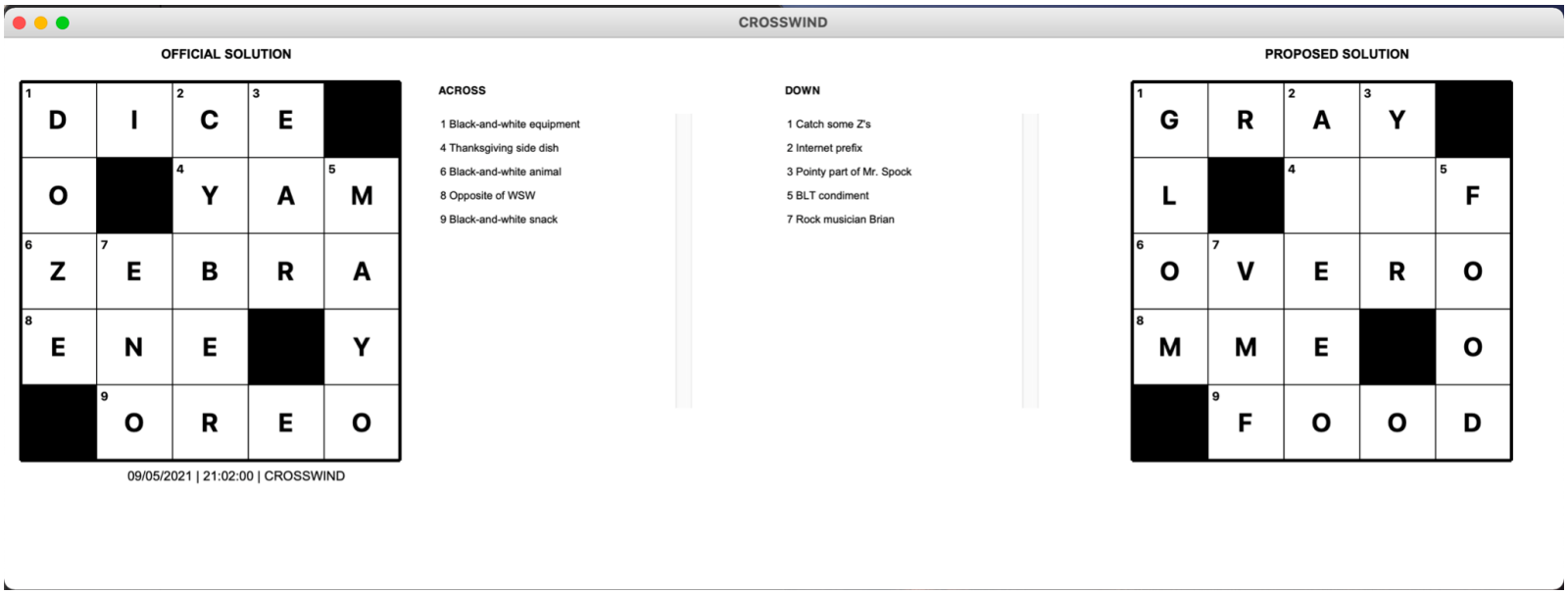

Figure 10: Sample Output

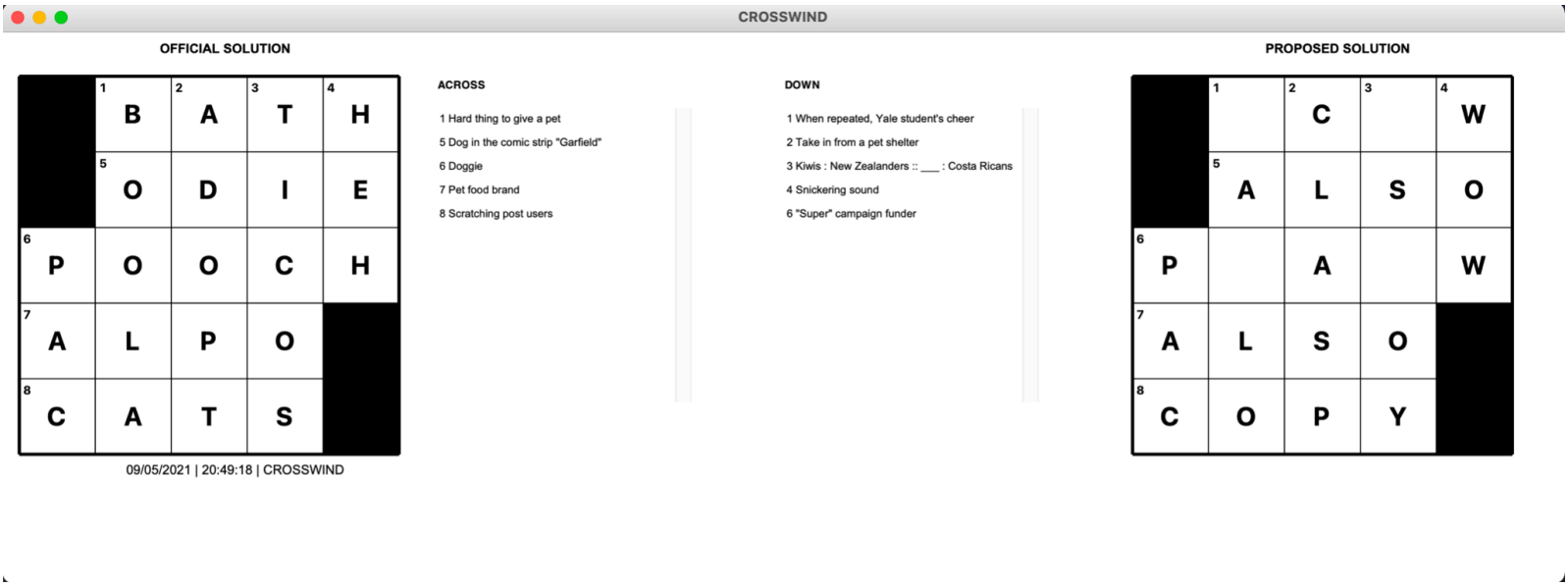

Figure 11: Sample Output

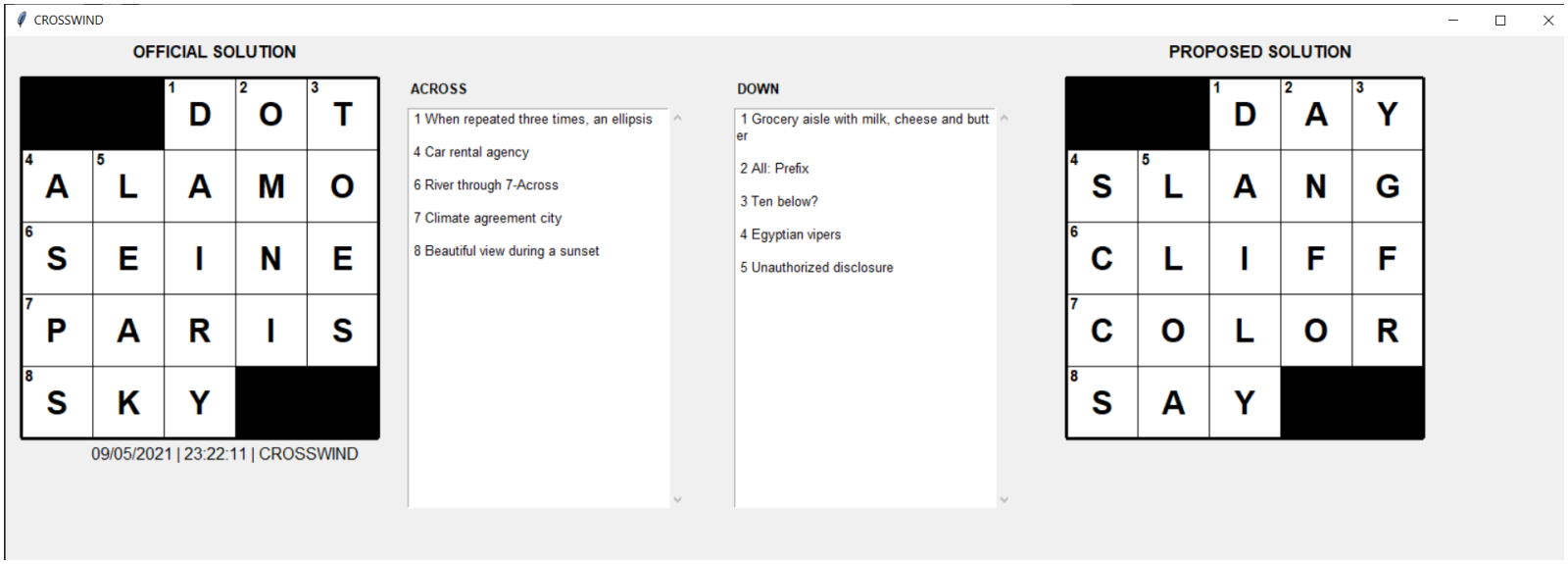

Figure 12: Sample Output

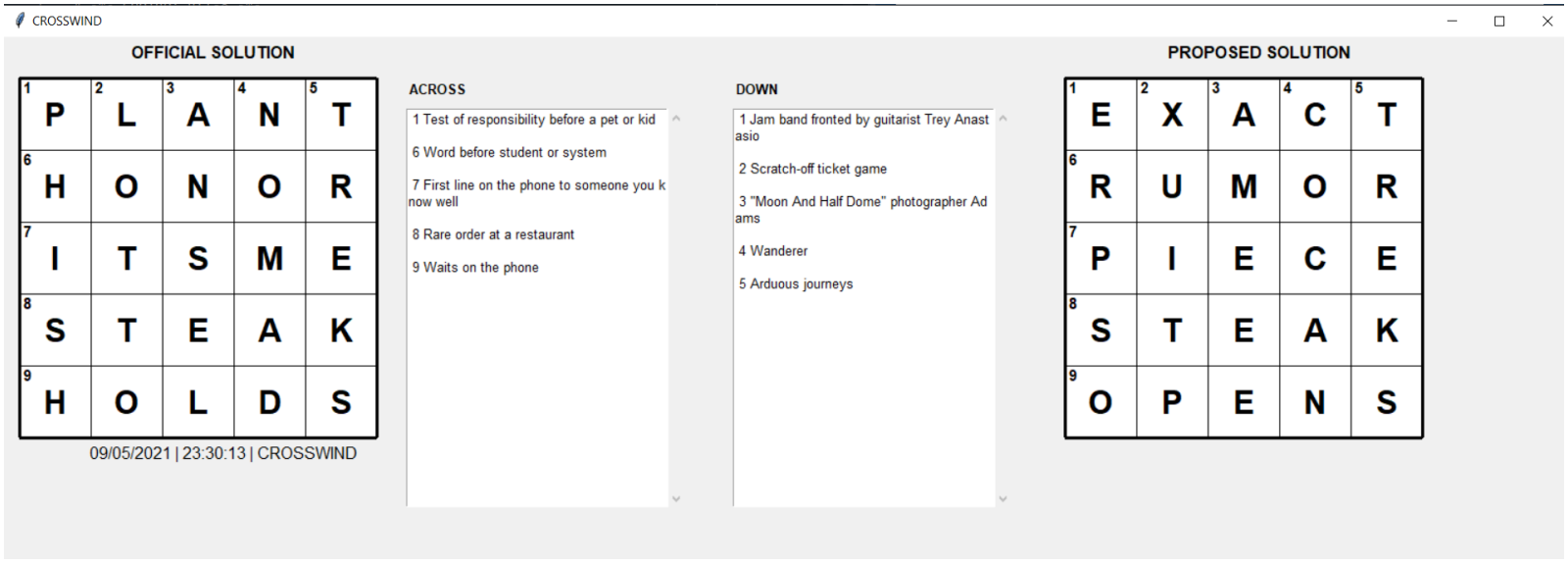

Figure 13: Sample Output

Bibliography

[1] Littman, M.L., Keim, G.A., Shazeer, N., 2002. “A probabilistic approach to solving crossword puzzles.” *Artificial Intelligence* 134, 23–55. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S000437020100114X doi:10.1016/S0004-3702(01)00114-X. [Accessed: 09-May-2021].

[2] Thanasuan, K. and Mueller, S. T., “Crossword expertise as recognitional decision making: an artificial intelligence approach.” *Frontiers*, 26-Aug-2014. [Online]. Available: https://www.frontiersin.org/articles/10.3389/fpsyg.2014.01018/full. [Accessed:09-May-2021].

[3] Jobin, A., Menon, A., Sekhar, A., Damodaran, V., 2017. “Key to crossword solving : NLP.” 929-933. 10.1109/ICACCI.2017.8125960. [Online]. Available: https://ieeexplore.ieee.org/document/8125960 [Accessed: 07-May-2021].

Appendix

```
from functools import cmp_to_key
from nltk.corpus import words
import copy
import re
import sys
from nltk.corpus import wordnet as wn

# global variables
all_words = wn.all_lemma_names()
word_scores = None

# This Node class is the main component of our search tree
# it contains the state of a solution which includes:
# current selections, remaining candidates, and depth of node
# constraints are also propagated through the tree for ease of coding,
# however constraints are passed by reference and each node is looking
# at the same constraints, these constraints are never modified
class Node():
    # A simple init function that stores the given  values and does
    # reduction based on current selections and constraints
    def __init__(self, candidates, selections, depth, constraints):
        self.candidates = candidates
        self.depth = depth
        self.constraints = constraints
        self.selections = selections
        self.set_selections()
        self.set_constraint_counts()

    # this removes all words from candidates that do not obey the constraints
    # from the current selections
    def reduce(self):
        self.candidates = reduce_by_selections(self.candidates,
                                               self.selections,
                                               self.constraints)

    # this method is used to find any new implicit selections other than
    # the one given by parent
```

```
# an implicit selection would be a candidate list that has only one member
# left due to reduction by constraints
# each such implicit selection is set to the actual selections
# and reduction is done again
def set_selections(self):
    self.reduce()

    for i, selection in enumerate(self.selections[0]):
        if len(selection) == 0:
            if len(self.candidates[0][i]) == 1:
                self.selections[0][i] = list(self.candidates[0][i].keys())[0]
                self.reduce()

    for i, selection in enumerate(self.selections[1]):
        if len(selection) == 0:
            if len(self.candidates[1][i]) == 1:
                self.selections[1][i] = list(self.candidates[1][i].keys())[0]
                self.reduce()

# find the number of constraints each candidate will have to obey
# due to the current selections
# it is later used to decide which children will be generated
# see Node.get_children()
def set_constraint_counts(self):
    self.has_constraints = False
    self.constr_counts = ([0,0,0,0,0],[0,0,0,0,0])

    acc_selections = self.selections[0]
    down_selections = self.selections[1]

    for constraint in self.constraints:
        acc_idx = constraint[0]
        down_idx = constraint[2]

        acc_selection = acc_selections[acc_idx]
        down_selection = down_selections[down_idx]

        if len(acc_selection) > 0:
            self.has_constraints = True
            self.constr_counts[1][down_idx] += 1
```

```
        if len(down_selection) > 0:
            self.has_constraints = True
            self.constr_counts[0][acc_idx] += 1

    def is_leaf(self):
        for candidate_map_list in self.candidates:
            for candidate_map in candidate_map_list:
                if len(candidate_map) > 1:
                    return False

        return True

    def get_hashable(self):
        keys = []

        for candidate in self.candidates:
            for cand_map in candidate:
                keys += cand_map.keys()

        return hash(frozenset(sorted(keys)))

    def get_child(self, direction, index, word):
        # direction: 0->accross, 1->down
        candidates = copy.deepcopy(self.candidates)
        candidates[direction][index] = {word: 1.0}
        selections = copy.deepcopy(self.selections)
        selections[direction][index] = word
        depth = self.depth + 1

        return Node(candidates, selections, depth, self.constraints)

    def get_children(self):
        acc_candidates = self.candidates[0]
        down_candidates = self.candidates[1]

        children = []

        # if the node is a leaf, return empty list to terminate
        # since it wont have any children
```

```
if self.is_leaf():
    return children

# generate children for all possible accross candidates
for i, cand_map in enumerate(acc_candidates):
    # if the new selection wont be obeying any constraints
    # we do not need to check that route since it will lead
    # to nodes with all accross selections
    # which is not desired
    # therefore we skip children that does not obey any new
    # constraints
    if self.has_constraints:
        if self.constr_counts[0][i] < 1:
            continue
            pass

    if len(cand_map) <= 1:
        continue

    for word in cand_map.keys():
        children.append(self.get_child(0, i, word))

# generate children for all possible down candidates
for i, cand_map in enumerate(down_candidates):
    # if the new selection wont be obeying any constraints
    # we do not need to check that route since it will lead
    # to nodes with all down selections
    # which is not desired
    # therefore we skip children that does not obey any new
    # constraints
    if self.has_constraints:
        if self.constr_counts[1][i] < 1:
            continue
            pass

    if len(cand_map) <= 1:
        continue

    for word in cand_map.keys():
        children.append(self.get_child(1, i, word))
```

```
        return children

    def get_score(self):
        score = 0.0
        for selection in self.selections:
            for word in selection:
                if len(word) > 0:
                    score += 0
                if word in word_scores:
                    score += word_scores[word]

        return score


# This is a function used to sort leafs at the end in order to make
# the best guess
# leafs are sorted by their scores, for more info see Node.get_score()
def node_compare(node_1, node_2):
    if node_1.get_score() > node_2.get_score():
        return -1
    elif node_1.get_score() == node_2.get_score():
        return 0
    else:
        return 1

# given candidates, current selections and all constraints
# remove all candidates that do not obey the constraints of the current selections
def reduce_by_selections(candidates, selections, constraints):
    acc_selections = selections[0]
    down_selections = selections[1]

    acc_cands = candidates[0]
    down_cands = candidates[1]

    for constraint in constraints:
        acc_idx = constraint[0]
        down_idx = constraint[2]

        acc_at = constraint[1]
```

```
        down_at = constraint[3]

        acc_selection = acc_selections[acc_idx]
        down_selection = down_selections[down_idx]

        acc_len = len(acc_selection)
        down_len = len(down_selection)

        to_be_removed = []

        if acc_len > 0 and down_len > 0:
            continue

        if acc_len > 0:
            letter = acc_selection[acc_at]
            for candidate in down_cands[down_idx].keys():
                if candidate[down_at] != letter:
                    to_be_removed.append(candidate)
            for word in to_be_removed:
                down_cands[down_idx].pop(word, None)

        if down_len > 0:
            letter = down_selection[down_at]
            for candidate in acc_cands[acc_idx].keys():
                if candidate[acc_at] != letter:
                    to_be_removed.append(candidate)
            for word in to_be_removed:
                acc_cands[acc_idx].pop(word, None)

    return (acc_cands, down_cands)

# helper function of get_letter_counts
# if the word is n characters, returns a list of n dictionaries
# i'th dictionary hold the number of letters seen for the i'th character
# the dictionary has letters as keys, and number of letters seen as values
def get_letter_count_of_word_list(word_list):
    counts = []

    if len(word_list) == 0:
        length = -1
```

```
    else:
        length = len(word_list[0])

    for i in range(length):
        count = {}
        for word in word_list:
            letter = word[i]
            if letter in count:
                count[letter] += 1
            else:
                count[letter] = 1
        counts.append(count)

    return counts

# finds the letter counts for all candidate lists
# we will have 10 candidate lists, 5 for accross, 5 for down
# if a candidate list containts words of length n, its letter counts will
# consist of n dictionaries, see get_letter_count_of_word_list(word_list)
# these letter counts are then used for reduction based on constraints
def get_letter_counts(candidates):
    acc_candidates = candidates[0]
    down_candidates = candidates[1]

    acc_counts = []
    down_counts = []

    for acc_cand_map in acc_candidates:
        word_list = list(acc_cand_map.keys())
        acc_counts.append(get_letter_count_of_word_list(word_list))

    for down_cand_map in down_candidates:
        word_list = list(down_cand_map.keys())
        down_counts.append(get_letter_count_of_word_list(word_list))

    return acc_counts, down_counts

def reduce_by_constraints(candidates, letter_counts, constraints):
    # k is the number of constraints a candidate needs to violate in order to
    # be removed
```

```
# setting k higher than 1 increases number of remaining candidates
# which can help keep the correct answers in the inital list
# however, it also keeps more of wrong candidates which might lead to
# wrong answers
k = 2

acc_candidates = candidates[0]
down_candidates = candidates[1]

acc_counts = letter_counts[0]
down_counts = letter_counts[1]

# this flag is used to keep track of changes
# so that if any candidate was removed, constraint checking is done again
# if there are no candidates removed in an iteration, then we know
# there is no more reduction to be done and quit the loop
flag = True

while (flag):
    # lists of candidates to be removed
    # we need this list because we cannot remove candidates while
    # iterating over them
    to_be_removed = ([{},{},{},{},{}],[{},{},{},{},{}])

    flag = False

    for constraint in constraints:
        acc_idx = constraint[0]
        down_idx = constraint[2]

        acc_at = constraint[1]
        down_at = constraint[3]

        acc_candidate = acc_candidates[acc_idx]
        down_candidate = down_candidates[down_idx]

        acc_count = acc_counts[acc_idx]
        down_count = down_counts[down_idx]

        for candidate in acc_candidate.keys():
```

```
            letter = candidate[acc_at]
            count = 0
            if len(down_count) > 0 and letter in down_count[down_at]:
                count = down_count[down_at][letter]

            if len(down_candidate) > 0 and count == 0:
                if candidate in to_be_removed[0][acc_idx]:
                    to_be_removed[0][acc_idx][candidate] += 1
                else:
                    to_be_removed[0][acc_idx][candidate] = 1


        for candidate in down_candidate.keys():
            letter = candidate[down_at]
            count = 0
            if len(acc_count) > 0 and letter in acc_count[acc_at]:
                count = acc_count[acc_at][letter]
            if  len(acc_candidate) > 0 and count == 0:
                if candidate in to_be_removed[1][down_idx]:
                    to_be_removed[1][down_idx][candidate] += 1
                else:
                    to_be_removed[1][down_idx][candidate] = 1

    # remove the candidates that were set to be removed
    for direction in range(2):
        for i in range(5):
            removing = to_be_removed[direction][i]
            for word in removing:
                if to_be_removed[direction][i][word] >= k:
                    flag = True
                    candidates[direction][i].pop(word, None)
                    for j, letter in enumerate(word):
                        letter_counts[direction][i][j][letter] -= 1

    return candidates, letter_counts

# uses either DFS or BFS method to find all leaves from the given root node
# note that it does not stop at some leaf, instead it generates all leaves
# and returns them all
def get_leafs(root):
```

```
# the method for searching can be selected here
# BFS lets us see how the stack is growing
# DFS is going to take less memory and hence be faster
# however, their time complexities would be the same in this situation
method = "bfs" # "bfs"

# initialize the stack and list of found leaves
stack = [root]
leafs = []

# these dicts are used to store previously seen nodes
# they are used to prevent expansion of the same node twice
# if it is acquired through some other path before
# hence they reduce the total runtime
seen_dict = {}
leaf_dict = {}

# while nodes are remaining in the stack
while len(stack) > 0:
    #print(len(stack))

    # pop and expand a node
    node = stack.pop()
    node_hashable = node.get_hashable()
    children = node.get_children()

    # add the leaves to leaves list
    if len(children) == 0 and node_hashable not in leaf_dict:
        leaf_dict[node_hashable] = None
        leafs.append(node)

    # add the non-leaves to the stack to later be expanded
    for child in children:
        child_hashable = child.get_hashable()
        if child_hashable not in seen_dict:
            seen_dict[child_hashable] = None
            if method == "dfs":
                stack.append(child)
            elif method == "bfs":
                stack.insert(0, child)
```

```
            else:
                print("\nSearch method is invalid, exiting:")
                sys.exit()

    return leafs

def get_top_k_helper(cands, k):
    for key in cands.keys():
        cands[key] = word_scores[key]

    return dict(sorted(cands.items(), key=lambda item: item[1])[-k:])

def get_top_k(candidates, k):
    for i, cand_map_list in enumerate(candidates):
        for j, cand_map in enumerate(cand_map_list):
            candidates[i][j] = get_top_k_helper(cand_map, k)

    return candidates


# this is the main function to be called from somewhere else
# it takes in a puzzle which holds clues and constraints
# constraints are a list of tuples that store which entries have crossings
# at which indices
# candidates contains two lists of dictionarites
# each list containts 5 dictionaries and each of these dictionaries holds
# the candidates for the corresponding enrty
# the function then calls all the other helper functions to find a solution
# print it on the console and return it so it can be shown on the gui
def solve_for(puzzle, candidates):
    constraints = puzzle.get_constraints()

    # store the word scores globally so they can later be used for ranking
    # possible solutions
    global word_scores
    word_scores = get_word_scores(candidates)

    #print_candidates(candidates, puzzle.clues)

    # reduce the candidates by constraints
```

```
    letter_counts = get_letter_counts(candidates)
    # candidates, letter_counts = reduce_by_constraints(candidates, letter_counts,
constraints)

    candidates = get_top_k(candidates, 30)
    print_candidates(candidates, puzzle.clues)
    # sys.exit()

    print("\nStarting Solving, The candidates we currently have:")
    print()
    # set the initial selections as empty
    # these selections will be held by each node in the search tree
    # and will be making up the state of a node
    selections = [["","","","","",""],["","","","","",""]]

    # construct root node and get the leafs of the tree
    # each leaf will be a possible solution, the best scored leaf will
    # be returned as the proposed solution
    root = Node(candidates, selections, 0, constraints)
    leafs = get_leafs(root)
    leafs = sorted(leafs, key=cmp_to_key(node_compare))

    # printing the best 10 leafs and their scores for tracing purposes
    print("\n%d leaves were found: " % (len(leafs)))
    print("Following were the 10 leaves with highest scores: ")
    for i, leaf in enumerate(leafs):
        print(leaf.selections)
        print(leaf.get_score())
        if i > 10:
            break

    # print the best guess and return it
    print("\nBest Guess:")
    print(leafs[0].selections)
    # sys.exit()
    return leafs[0].selections


## Helper methods
```

```
def get_word_scores(candidates):
    word_scores = {}
    for cand_map_list in candidates:
        for cand_map in cand_map_list:
            for word in cand_map:
                word_scores[word] = cand_map[word]

    return word_scores

def print_candidates(candidates, clues):
    acc_cands = candidates[0]
    down_cands = candidates[1]
    acc_clues = clues[0]
    down_clues = clues[1]

    for i, cands in enumerate(acc_cands):
        print()
        print("Clue: %s" % (acc_clues[i]))
        print(cands)

    for i, cands in enumerate(down_cands):
        print()
        print("Clue: %s" % (down_clues[i]))
        print(cands)
```

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 CROSSWIND.

Word Count: 2006