Senior Project

Bryn Mawr Bee: A Word Puzzle Game

Team Members: Emily Lobel, Eunsoo Jang, Sophia Trump, and Maria Vivanco
Advisors: Deepak Kumar and Geoffrey Towell

Submitted in Partial Fulfillment of the Requirements of the BA in Computer Science
Bryn Mawr College
Spring 2020

Abstract

The Spelling Bee team set out to recreate the NYTimes Spelling Bee application. We did this in order to have the experience of building out a full stack application. We did this by creating algorithms and databases in Python and MySQL, building out a server-side in PHP, and a front-end with HTML, CSS, and Javascript. We were able to build out a fully functioning application inspired by the design of NYTimes. In the future, we hope to iterate on the design to find a more unique and Bryn Mawr inspired front-end, and potentially work to create more ways to play the Spelling Bee game.

Acknowledgements

We would like to thank our advisors, Deepak Kuman and Geoffrey Towell.

Emily would like to thank her family and friends for supporting her through two thesis projects. She would also like to give even bigger thanks to her thesis partners, Eun Soo, Sophia, and Maria, for making this thesis process the best it could be, and her advisors Deepak and Geoff for supporting the team throughout the way.

Maria would like to thank her parents,her sister Caroline, and her friends for supporting her through her time as a computer science student at Bryn Mawr. She'd like to thank Naomi Stock for being her number one supporter during her time at Bryn Mawr and beyond. She couldn't have accomplished half of what she did without Naomi's unwavering encouragement. She would also like to thank her thesis group for helping her write and build this project. Couldn't have done it without them! And lastly, she'd like to thank her advisors, Deepak and Geoff for their guidance and help throughout this project.

Eunsoo would like to thank her parents and friends for supporting her through her journey at Bryn Mawr College. Also, she would like to thank her thesis group and advisors for all the help. If it weren't for them this project could have not finished as it did.

Sophia would like to thank her parents, sister, Campbell, Grace, and Nina for their love, support, encouragement, and friendship. Okeye! She would also like to thank her first grade teacher, Mrs. Kellon. She is also grateful for her teammates, Emily, Eunsoo, and Maria, as well as Deepak and Geoff for their support throughout the process.

Contents

1. Introduction

We designed and built a word puzzle game. We were inspired by the daily Spelling Bee puzzle that appears in the New York Times. We developed a web application for this word puzzle from scratch. This was a team project.

As we transition into software engineers, designing and developing the Spelling Bee application helped us as computer science students by giving us experience with full stack development. Working on this Spelling Bee project allowed us to discover a new or better algorithm to generate the puzzles and to create a new interface for the game.

The main components of our application are the Dictionary, Puzzle Generator, Puzzle Solver, and Puzzle Player. The front end of our application uses a combination of HTML, CSS, and Javascript. The mid-tier of our application is implemented in PHP. This mid-tier deals with user logins, saving the state of the game, and generating/solving puzzles. The backend of our application is implemented in MySQL. Our application also implements user and session tracking, which is supported by the backend and mid-tiers. For version control, we used Github/Git to upload and manage our code.

This report describes the functionality and aesthetics of the NYTimes Spelling Bee and our process in replicating the game. We detail the two parts of recreating the puzzle: the analysis and test procedures of the components of the puzzle, and the process and results of creating the complete end user application.

2. The NYTimes Spelling Bee Puzzle

In the Spelling Bee Game, a puzzle is a collection of seven letters, one of which is called the key letter. Each puzzle has at least one pangram: a word that contains all the seven puzzle letters, and only those seven letters. A pangram may contain repeated letters, and as a result the length of a pangram may be longer than seven letters. A perfect pangram is a pangram that is exactly seven letters long, composed of each of the seven letters exactly once.

Each valid word has to have at least four letters. Letters in a word may be repeated as needed. A puzzle dictionary is used to validate the player's words. The full game rules, as outlined on the NYTimes Spelling Bee, are as follows [Source 1]:

(1) The user must create words that contain at least 4 letters.
(2) Words must include the key/center letter.
(3) No proper nouns, hyphenated, or curse words are allowed.
(4) Letters can be used more than once.
(5) You cannot guess a word twice.

These rules determine whether a word that a user guesses is valid or not.

2.1 Scoring

The player is awarded points for each word. The scoring rules are as follows [1]:

(1) 4-letter words are worth 1 point each

(2) Words longer than 4 letters earn 1 point per letter (a 5 letter word earns 5 points)
(3) Whenever a user enters a word that is a pangram, the user earns 7 extra points.

There is a "scoring ladder" signifying different point levels. This scoring ladder is as follows [1]:

(1) Beginner
(2) Good Start
(3) Moving Up
(4) Good
(5) Solid
(6) Nice
(7) Great
(8) Amazing
(9) Genius

The points allocated to each level are specific to the puzzle and its solutions.

2.2 User Interface

As seen in Figure 1, the web interface consists of several parts: the hive (containing the puzzle), list of input correct words, and the score ladder. The flow of the game is as follows: the player constructs a word using the letters in the hive, the game checks the validity of the word, and (if valid) the word is added to the valid word list. The scoring ladder is displayed on top of the words. As the user gains more points, the level is shown on the ladder along with the points.

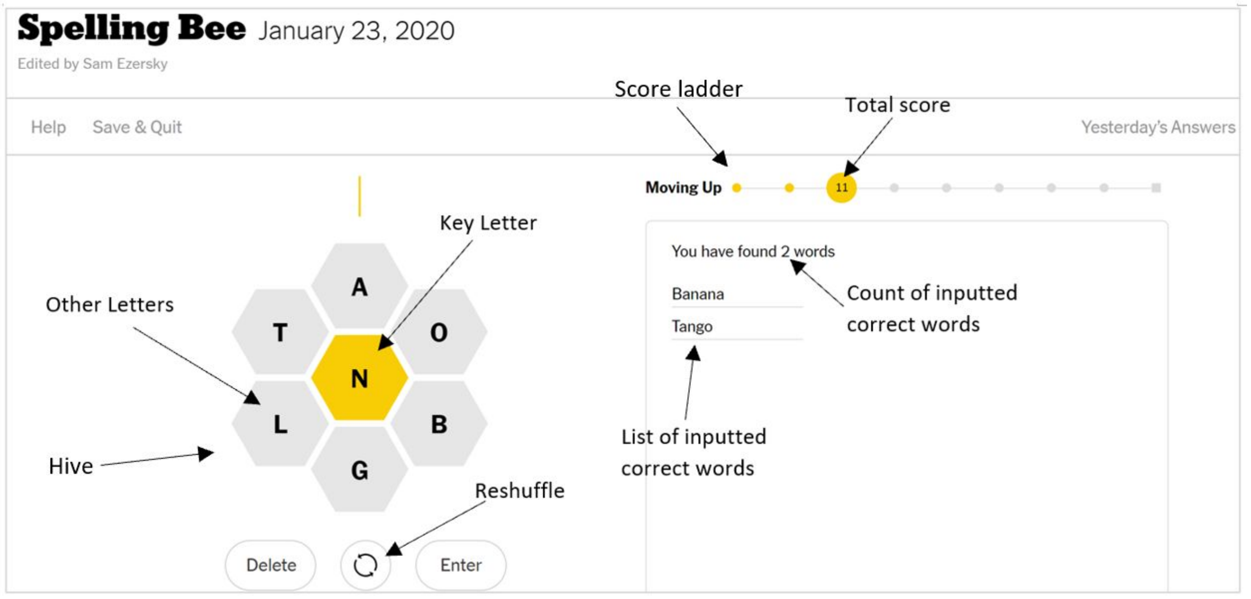


Figure 1: Web View of Spelling Bee Game [1].

2.2.1 The Hive

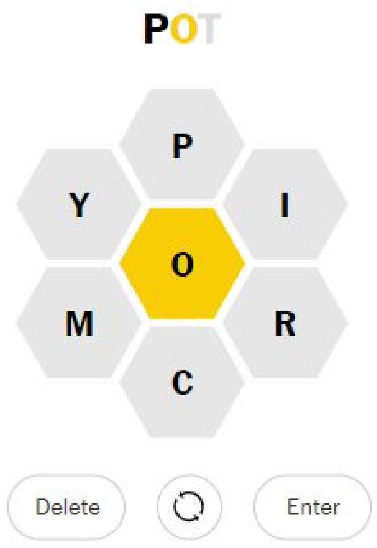


Figure 2: Puzzle of the Day hive [1].

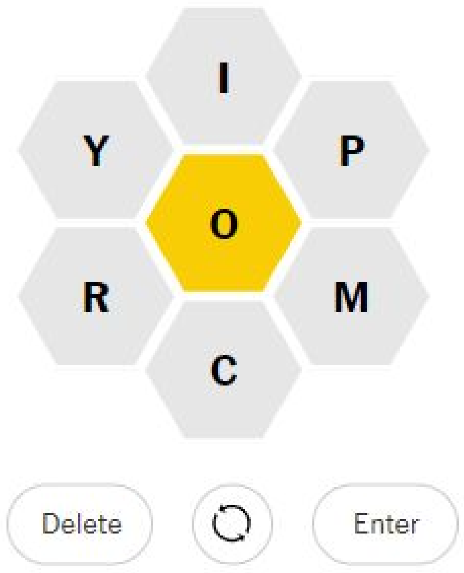


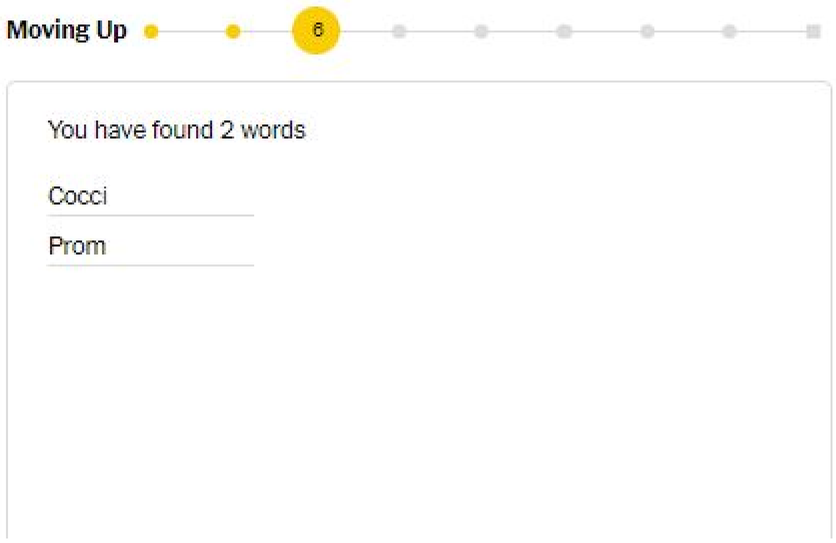
Figure 3: Hive after Refresh button pressed [1].

As shown in Figure 2, the puzzle's letters are split into a hive graphic with the yellow key letter in the center. In this example, the puzzle of the day is "CYOMIPR" where "O" is the key letter. The user can either type the word out using their keyboard or construct the word by clicking on the different letters making up the puzzle. As the user types, letters appear above the hive. If the letter is not a letter in the puzzle, it is colored gray. If a letter is an optional letter, it is colored black. If it is the key letter, it is colored yellow [1].

There are three buttons below the hive: delete, reshuffle, and enter. The reshuffle button scrambles the non-key letters as shown in Figure 3. This may help users to think of a different word. The delete button serves as a backspace button. The enter button submits the word to be checked for validity [1].

2.2.2 List of Input Correct Words

Once a user enters a word, the game checks the validity of the word and responds accordingly. If the word contains less than 4 letters, the game informs the user: "too short". If the word has invalid letters, the game again informs the user: "invalid letters." The different possibilities are demonstrated in Table 1. However, if the word is valid (i.e present in the dictionary and follows the rules) , the word is added to the "found" words as shown in Figure 4. Once a word is added to the "found" words, it cannot be guessed again [1].



You have found 2 words

Cocci

Prom

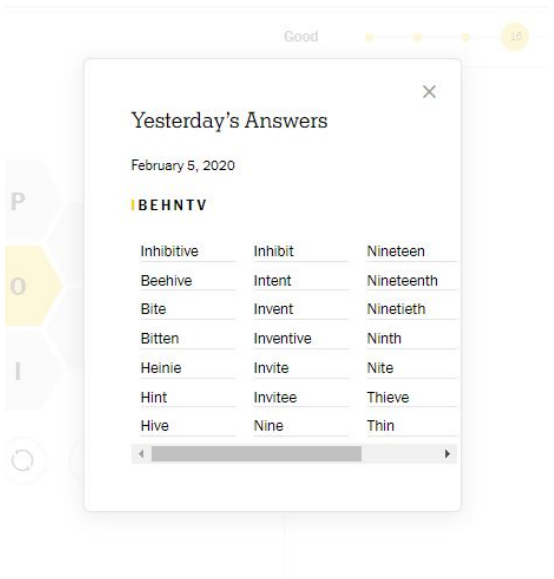
Figure 4: Word List [1].

Scenario	**Response**
Word w/ Length < 3	“Too short”
Word that is a proper noun or curse word	“Not in word list”
Word is not found in dictionary	“Not in word list”
Word with invalid letters	“Bad Letters”
Word that’s been used already	“Already used”
Valid word	“Good”, “Great”, “Pangram”

Table 1: Different Scenarios for input words [1].

2.2.3 Additional Features

Additional features are added on the menu bar such as “Yesterday’s Answers”, “Help”, “Save and Quit”. Since NYTimes assigns a unique puzzle daily, answers to the puzzle are released the following day [1]. “Yesterday’s Answers” displays the solutions for the previous day in a scroll view as seen in Figure 5. The “Help” tab shows the user the game’s rules and point system as seen in Figure 6. The “Save and Quit” tab uses cookies to identify and save the user’s process as seen in Figure 7.



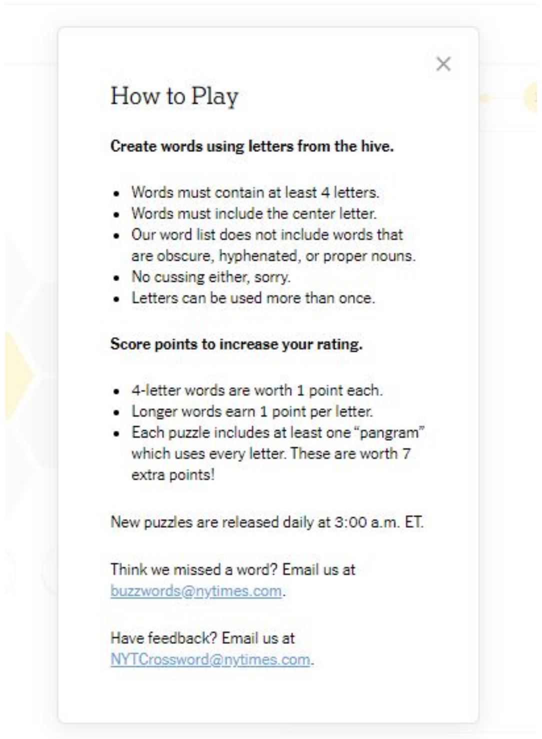
Yesterday's Answers

February 5, 2020

IBEHNTV

Inhibitive	Inhibit	Nineteen
Beehive	Intent	Nineteenth
Bite	Invent	Ninetieth
Bitten	Inventive	Ninth
Heinie	Invite	Nite
Hint	Invitee	Thieve
Hive	Nine	Thin

Figure 5: "Yesterday's Answers" [1].



How to Play

Create words using letters from the hive.

- Words must contain at least 4 letters.
- Words must include the center letter.
- Our word list does not include words that are obscure, hyphenated, or proper nouns.
- No cussing either, sorry.
- Letters can be used more than once.

Score points to increase your rating.

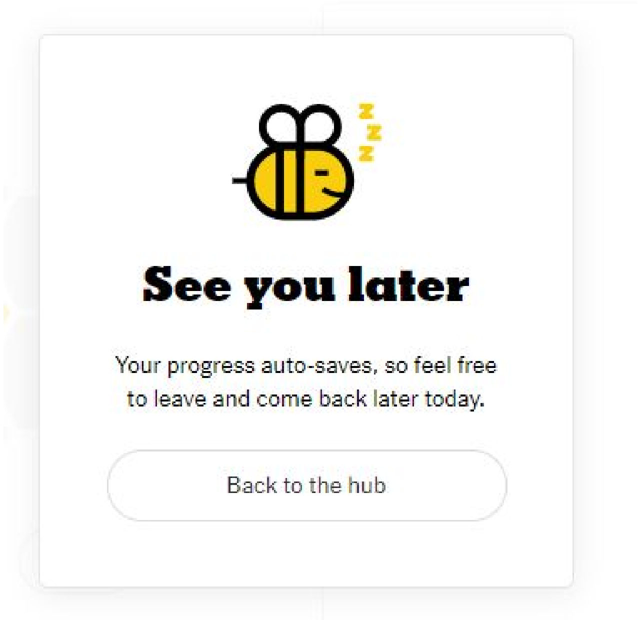
- 4-letter words are worth 1 point each.
- Longer words earn 1 point per letter.
- Each puzzle includes at least one "pangram" which uses every letter. These are worth 7 extra points!

New puzzles are released daily at 3:00 a.m. ET.

Think we missed a word? Email us at buzzwords@nytimes.com.

Have feedback? Email us at NYTCrossword@nytimes.com.

Figure 6: Rules View [1].



See you later

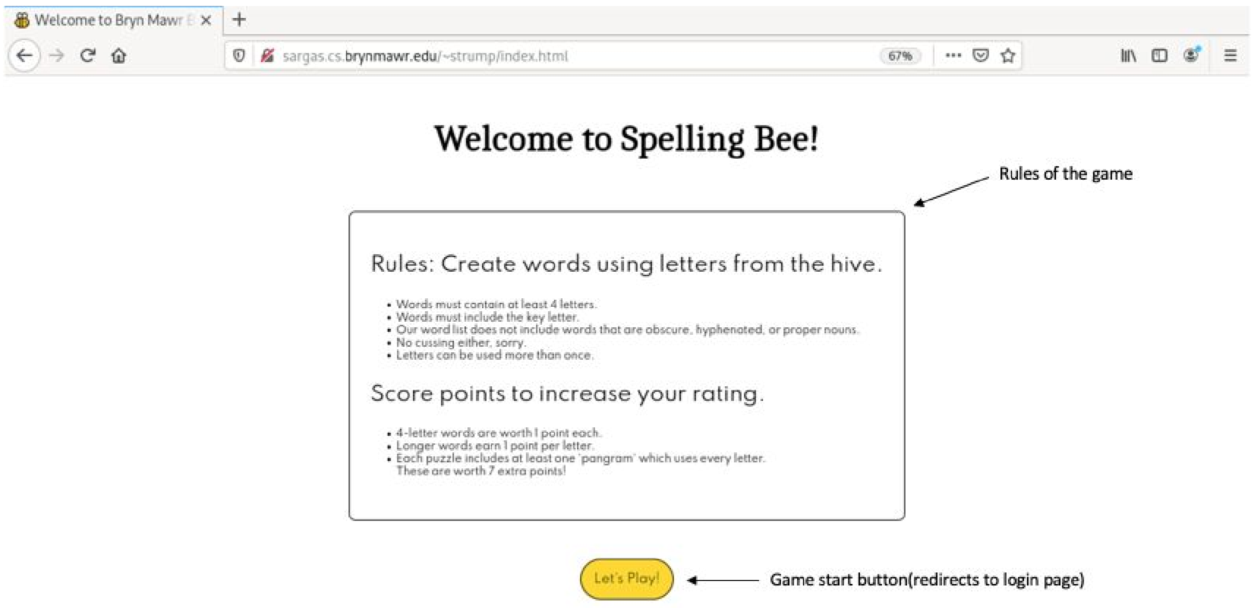
Your progress auto-saves, so feel free to leave and come back later today.

Figure 7: Save and Quit [1].

3. Bryn Mawr Bee

Based on the design of the NYTimes Spelling Bee game, we built our own "Bryn Mawr Bee" from scratch. The Bryn Mawr bee game has the same rules as the New York Times and similar aesthetics. In this section, we will walk through a typical playthrough of Bryn Mawr Bee.

3.1 Landing Page


Welcome to Bryn Mawr B
sargas.cs.brynmawr.edu/~strump/index.html
67%
Rules of the game
Let's Play!
Game start button(redirects to login page)

Welcome to Spelling Bee!

Rules: Create words using letters from the hive.

- Words must contain at least 4 letters.
- Words must include the key letter.
- Our word list does not include words that are obscure, hyphenated, or proper nouns.
- No cussing either, sorry.
- Letters can be used more than once.

Score points to increase your rating.

- 4-letter words are worth 1 point each.
- Longer words earn 1 point per letter.
- Each puzzle includes at least one 'pangram' which uses every letter. These are worth 7 extra points!

Figure 8: The Landing Page of our Bryn Mawr Bee.

Our landing page displays the rules of the game, which are identical to the NYTimes' Spelling Bee rules. The original Spelling Bee had its rules as a pop-up on the same page of the puzzle player. We decided to forgo this because our app's server-side code requires a form submission to activate the puzzle generation and selection, which a button press enables.

After the landing page comes user tracking. Our login is more geared for the Spelling Bee game rather than the NYTimes website as a whole, so that feature was added in a more specific manner and designed by us.

3.2 Login

Once the user starts a new game, the user is prompted to login to our system. We implemented a login system to keep track of user statistics and to save user progress. If the user is a new player, the user can create a new account. If not, the user logs into their account. Our system then loads the state of the user's game so the user can continue where they left off. If they are a new player, our system creates a new game for the user. A more technical description of our implementation is outlined in Section 4.

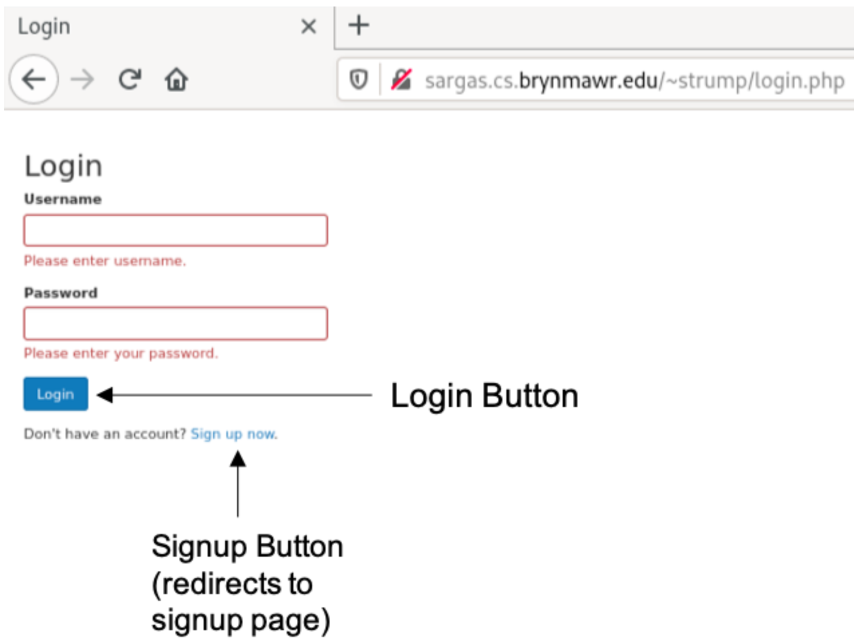

Login
sargas.cs.brynmawr.edu/~strump/login.php
Login
Username
Please enter username.
Password
Please enter your password.
Login
Login Button
Don't have an account? Sign up now.
Signup Button (redirects to signup page)

Figure 9: Our Bryn Mawr Bee's login screen, which appears right after the user clicks "Let's Play!" in the Landing Page (Figure 8). There is validation on each text field.

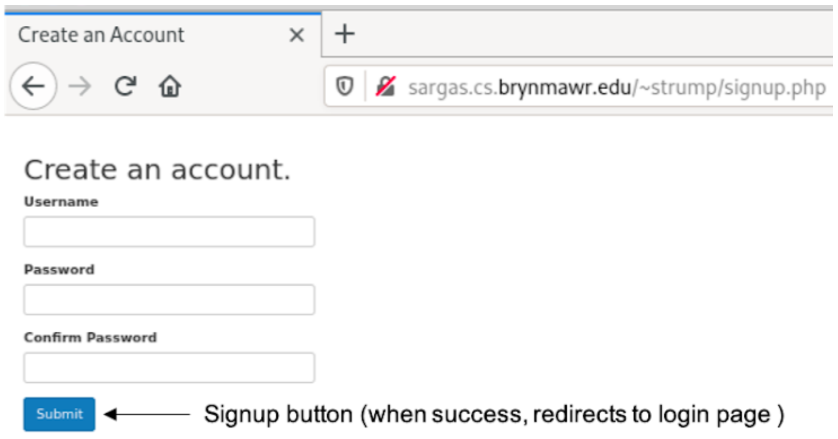

Create an Account
sargas.cs.brynmawr.edu/~strump/signup.php
Create an account.
Username
Password
Confirm Password
Submit
Signup button (when success, redirects to login page)

Figure 10: The Bryn Mawr Bee sign up screen. Passwords are hidden.

3.3 Puzzle Player

Once the user logs in, our system loads the most recent game that the user has played or a new game if the user is new.

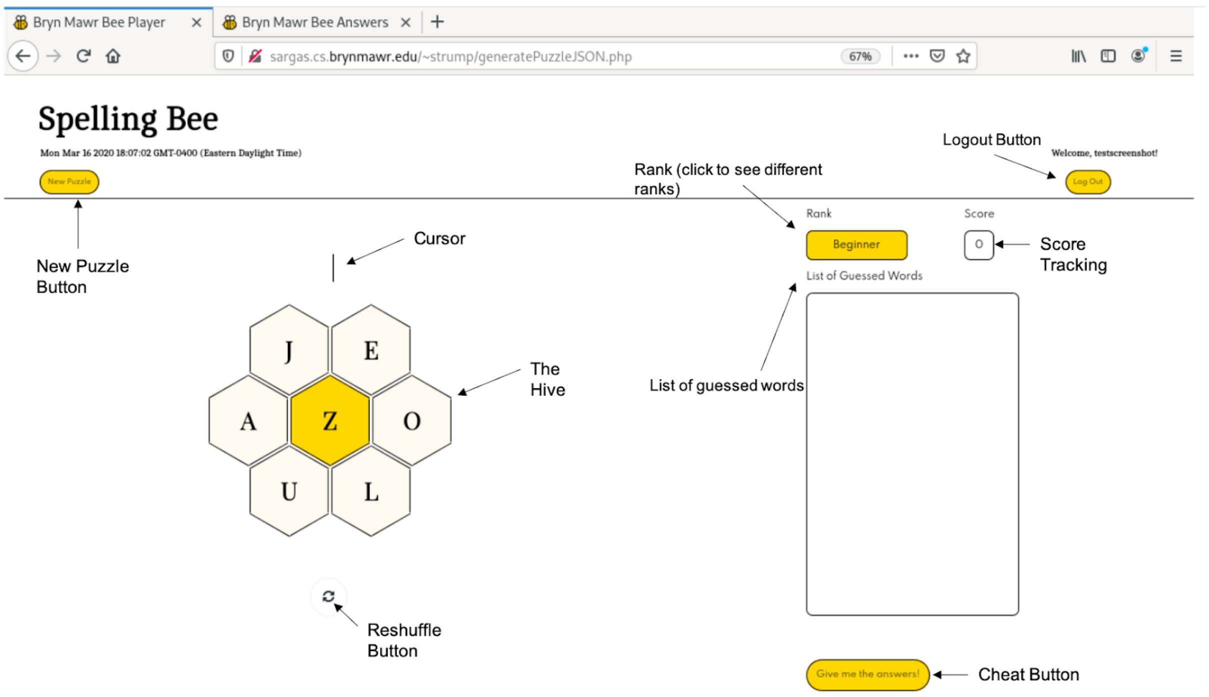
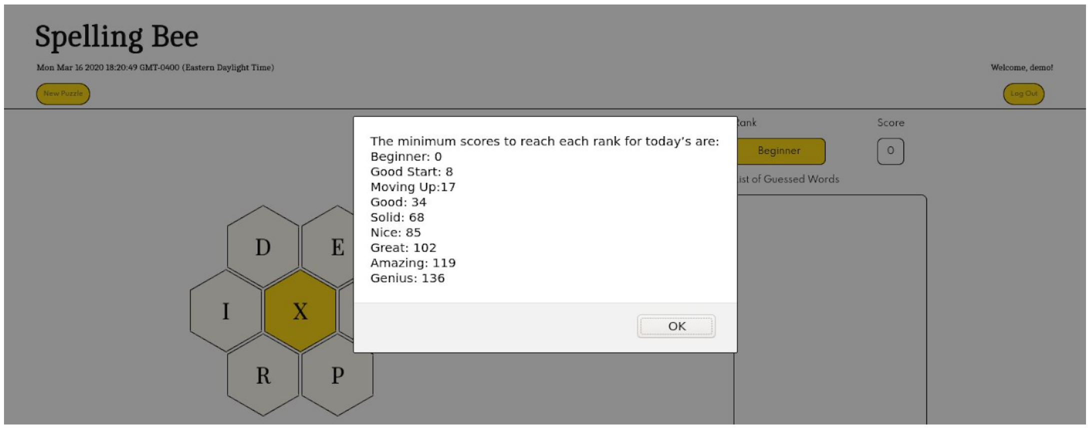


Figure 11: The Homepage of our Bryn Mawr Bee.

This is our core puzzle player. Like the NYTimes app, we decided on a yellow and white color scheme and followed a similar layout. Like NYTimes, we have a hive layout for the puzzle letters. We also implemented the reshuffle button but omitted the delete and submit button. We thought these were redundant in the design and users are more likely to use their keyboards to do these actions. We kept the same functionality for the reshuffle button as NYTimes. For the layout of the hive, we used an online tutorial from CSS script.

To play, users guess a word using the letters in the hive and, if it's in our dictionary and valid according to the rules, it is added to the "List of the Guessed Words." If invalid, relevant error messages are shown. The Bryn Mawr Bee app follows the New York Times version for outputting messages when the user's guess is valid (see Table 1). The user can also view the scoring ladder by clicking the rank button. As in NYTimes, ranks are calculated through different percentages unique to each puzzle. These features were all inspired by NYTimes.

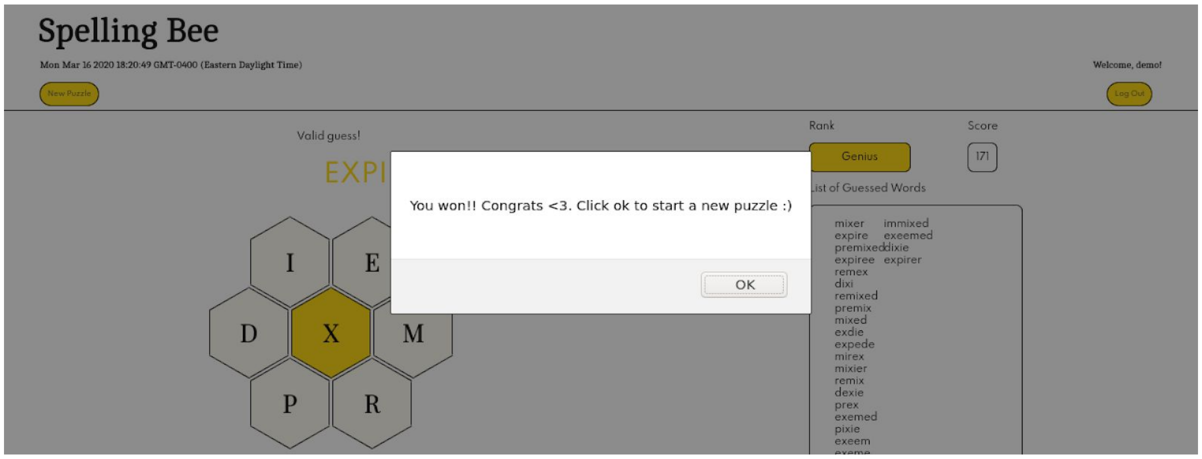


The minimum scores to reach each rank for today's are:
Beginner: 0
Good Start: 8
Moving Up:17
Good: 34
Solid: 68
Nice: 85
Great: 102
Amazing: 119
Genius: 136

Figure 12: The pop-up for our Ranks.

3.4 Finishing the Game

Once the user guesses all of the solutions for the puzzle, they get an alert to notify them that they’ve won! Upon dismissing the alert, the user is presented with a new puzzle.



You won!! Congrats <3. Click ok to start a new puzzle :)

```
mixer    immixed
expire   exeemed
premixeddixie
expiree  expirer
remex
dixi
remixed
premix
mixed
exdie
expede
mirex
mixier
remix
dexie
prex
exemed
pixie
exeem
```

Figure 13: User has filled out the list of words, achieved rank Genius, and completed the puzzle.

3.5 Additional Features

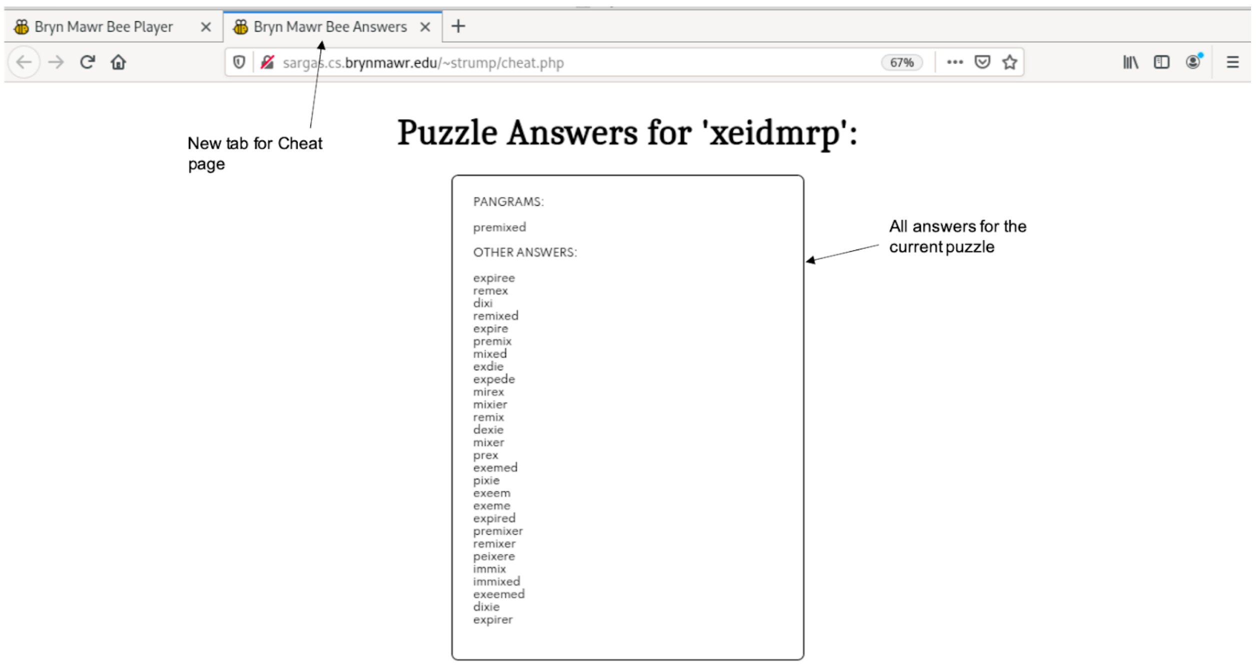
We added two more features unique to Bryn Mawr Bee. In our Spelling Bee, a user can ask for a new puzzle and cheat. Although the NYTimes does allow cheating in some of its other games, their version of Spelling Bee does not allow cheating. We decided that these two features can ease the difficulty of the game if needed and provided the user with a more pleasant experience.

3.5.1 New Puzzle

The “New Puzzle” button allows a logged in user to generate a new puzzle. This is in lieu of the daily puzzle feature that the NYTimes offers, which only allows users to play one puzzle per day. In our version, you can generate a new puzzle at any time! Once the user presses the new puzzle button, our front-end is reloaded to display a new puzzle.

3.5.2 Cheating

We added a “Give me the answers!” button -- a way to cheat at the game that we provided ourselves. Once the user presses this button, a new tab is opened with the solutions to the current puzzle. The pangram is separated out from the rest of the data for easy and fast point grabbing. See Figure 14.


Bryn Mawr Bee Player
Bryn Mawr Bee Answers
sargas.cs.brynmawr.edu/~strump/cheat.php
67%
New tab for Cheat page
All answers for the current puzzle

Puzzle Answers for 'xeidmrp':

PANGRAMS:

premixed

OTHER ANSWERS:

expiree
remex
dixi
remixed
expire
premix
mixed
exdie
expede
mirex
mixier
remix
dexie
mixer
prex
exemed
pixie
exeem
exeme
expired
premixer
remixer
peixere
immix
immixed
exeemed
dixie
expirer

Figure 14: The Bryn Mawr Bee Cheat page.

4. Architecture of Bryn Mawr Bee

This section describes the implementation design of our Bryn Mawr Bee. Our system is made up of the four components : Dictionary, Puzzle Generator, Puzzle Solver, and Puzzle Player. We also implemented user and session tracking, supported by a database, which is also described below. The front end uses a combination of HTML, CSS, and Javascript. The mid-tier of our application is implemented in PHP, and deals with user logins, saving the state of the game, and generating/solving puzzles. The backend of our application is implemented in MySQL. For version control, we used Github/Git to upload and manage our code. The Appendix contains a full listing of relevant files and implementation details, organized by component.

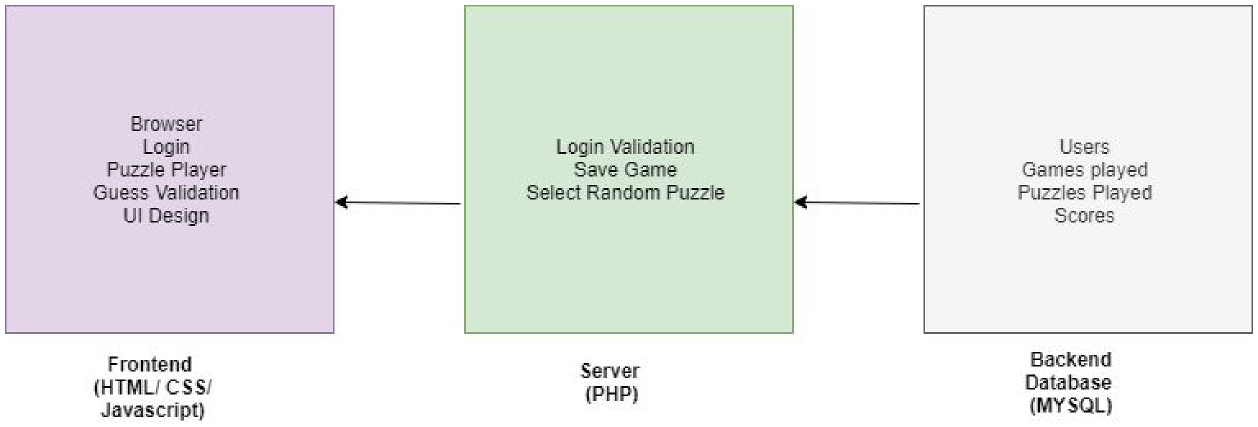


Figure 15: A diagram of our app’s stack.

In the subsequent sections, we describe each of the components (see Figure 16).

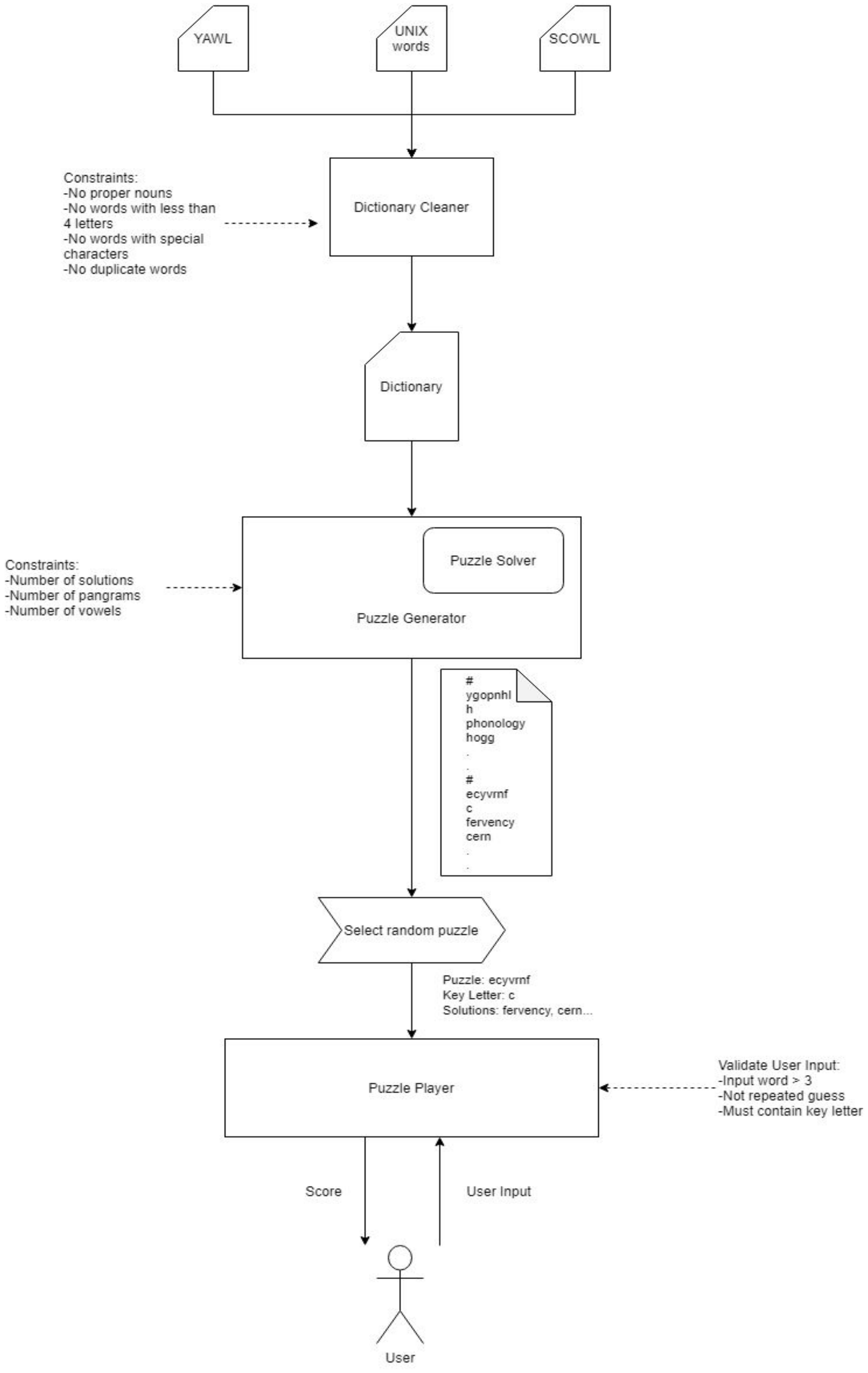


Figure 16: Diagram of the major components.

4.1 Dictionary

The first main component of our application is the dictionary. This dictionary must contain only English words with 4 or more letters, no proper nouns, no curse words, and no hyphenated words. The generation of our dictionary was implemented using a Python script. The cleaned dictionary is output to a text file. See Appendix Section 1 for details.

We created the dictionary by combining three dictionaries. The first dictionary, comprising 263,534 words, was the YAWL (Yet Another Word List) word list, which is derived from the Google WebTrillion Word Corpus and issued by the Linguistic Data Consortium [3]. The second dictionary, comprising 235,887 words, was the UNIX words dictionary, which originates from Webster's Second International Dictionary (1934). The third dictionary, comprised of 81,884 words, was the"2of12inf" list of words from SCOWL (And Friends), which derives its words from 12 English-language dictionaries, the GNU Aspell spell checker, Moby Words, and WordNet [4;5]. See Appendix Section 1 for relevant details.

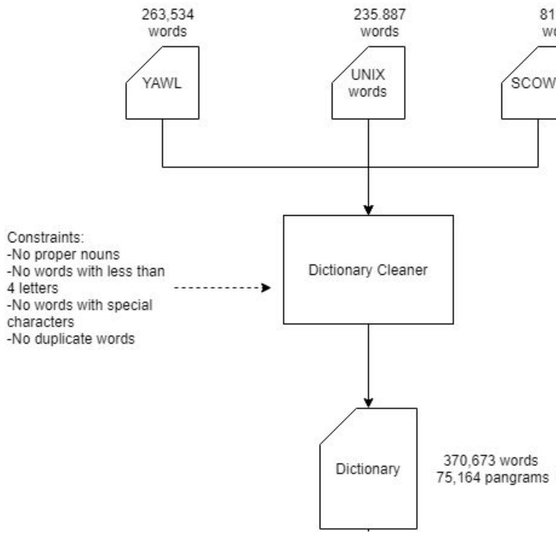


Figure 17: Dictionary Construction.

Our algorithm for combining the three sub-dictionaries and cleaning them is as follows:

```
Procedure DICTIONARY-CLEANER(dictionary 1, dictionary 2, dictionary 3)
Inputs:
   -  dictionary 1..3: filenames of sub dictionaries
Output: A cleaned master dictionary file with no proper nouns, no words less
than 4 letters long or special characters, and no duplicates.
1. Find the set of each word in the sub-dictionaries.
2. Combine the sub-dictionaries by taking the union of each dictionary set.
Call this <combined_dictionary>.
3. For each word in <combined_dictionary>,
  A. If the word is less than 4 letters: do nothing.
  B. If the word contains special characters: do nothing.
  C. If the word begins with a capital letter (is a proper noun): do nothing.
  D. Otherwise: write the word to the cleaned dictionary file.
```

After cleaning all three dictionaries, our combined dictionary contains 370,673 words. Our dictionary contains 75,164 pangrams, and 24,575 unique pangrams (where the pangrams “primarily” and “primally” are considered the same). This dictionary is generated once throughout the lifetime of the app.

4.2 Puzzle Solver

The puzzle solver is the second main component of our application. Given the dictionary, and a puzzle (received from either the Puzzle Generator or the Puzzle Player), the Puzzle Solver finds all the words in the dictionary. The Puzzle Solver is run in the Puzzle Generator step, which is run only once in the lifetime of the web app. The Puzzle Solver is implemented as a Python function, which is called by the Puzzle Generator and the Puzzle Player (See Appendix 2). The list of solutions generated by the Puzzle Generator are written to a file, which is used by the Puzzle Player to play the game, or to cheat if the user desires (See Appendix Section 3). Our algorithm for the puzzle solver is as follows:

```
Procedure PUZZLE-SOLVER (k, p, d)
Inputs:
   -  k: key letter
   -  p: string of puzzle letters
   -  d: dictionary file
Output: List of puzzle solutions to the given puzzle
1. For each word in d,
    A. If k is in word:
      i. If the letter set of the current dictionary word is equal to the set
      of the puzzle letters:
         a. The word is a puzzle answer AND it is a pangram. Add it to the
            list of pangrams and the list of puzzle answers.
      ii.  If the set of the current dictionary word is a subset of the set
      of the puzzle letters:
         a. Then the word is a puzzle answer. Add it to the list of puzzle
            answers.
2. Return the list of puzzle answers.
```

4.3 Puzzle Generator

The third main component in our application is the puzzle generator. The puzzle generator is the component of our application that searches through the dictionary to compile the list of suitable puzzles. The Puzzle Generator is implemented as a Python script. Its output is a file with all the generated puzzles, separated by ‘#’ (See Appendix Section 3).

A puzzle contains a key letter as the first letter followed by six other unique letters and generates at least one pangram. Our dictionary contains 75,164 pangrams, 13,162 perfect pangrams, and 24,575 unique pangrams (where the pangrams "primarily" and "primally" are considered the same). Two puzzles with identical puzzle letters, but different key letters, are considered distinct puzzles. As such, each pangram can generate 7 unique puzzles, where each unique letter in the pangram is selected to be the key letter once. However, following this exhaustive puzzle generation approach would generate a total of 172,025 puzzles, which will be enough to provide approximately 471 years of daily puzzles. Since this amount of puzzles is exceedingly large, exploration was required in order to restrict the number of puzzles generated, while still maintaining the most suitable puzzles for our application.

To generate the appropriate amount of puzzles, we set constraints on the number of answers that each puzzle generates. We consider puzzles valid only if they fall within our specified range of answers allowed per puzzle. The first range we considered was 15 to 60 solutions. The rationale for this range came from asking the opinions of regular users of Spelling Bee. They considered puzzles with less than 15 solutions both frustrating and unsatisfying to play, and puzzles with over 60 solutions exhausting and too time-consuming.

We designed and implemented two algorithms to generate puzzles within our desired range of 15 to 60 answers per puzzle: the Pangrams-First algorithm and the Random Pangrams algorithm.

4.3.1 The Pangrams-First Algorithm

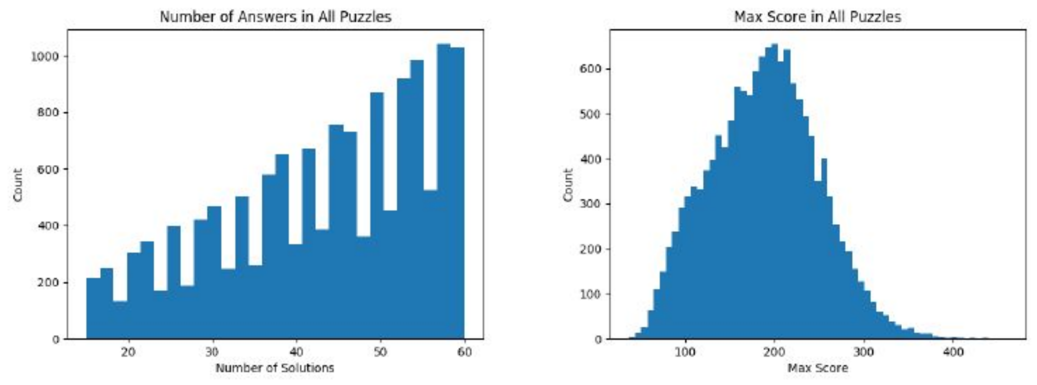
The first algorithm that was designed and implemented to generate puzzles was the Pangrams-First algorithm. The Pangrams-First Algorithm was implemented in Python (See Appendix Section 3).

The general premise of the Pangrams-First algorithm is to start with each pangram in the dictionary, then build out the puzzle, so long as the number of puzzle answers fall within the puzzle constraints. The Pangrams-First algorithm is as follows:

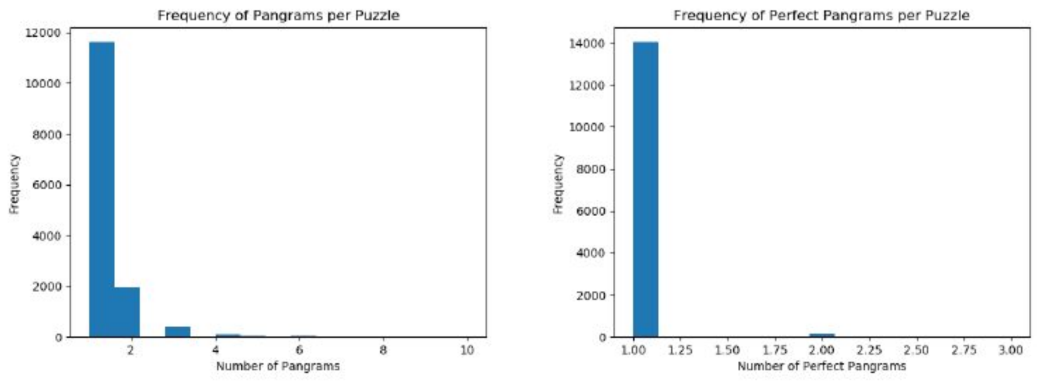
```
Procedure PANGRAM_FIRST(dictionary)
Input:
   -  dictionary: cleaned dictionary output from dictionary cleaner.
Output: A file with all the generated puzzles (puzzle letters, the key
letter, and the puzzle solutions), separated by '#'.
1. Given the dictionary, find all the unique pangrams.
2. For each unique letter in each pangram,
   A. Generate a puzzle such that the current letter is the key letter.
      i. Call the puzzle solver to generate all answers from this puzzle.
      ii. If the number of answers generated from this puzzle is within the
      specified range of answers allowed per puzzle: it is a valid puzzle.
      Therefore, write the puzzle letters, the key letters, and the puzzles
      solutions to the file.
      iii. Otherwise: the puzzle is invalid as should be disregarded.
```

4.3.1.1 Analysis of Pangrams-First with 15 to 60 Puzzle Answer Range

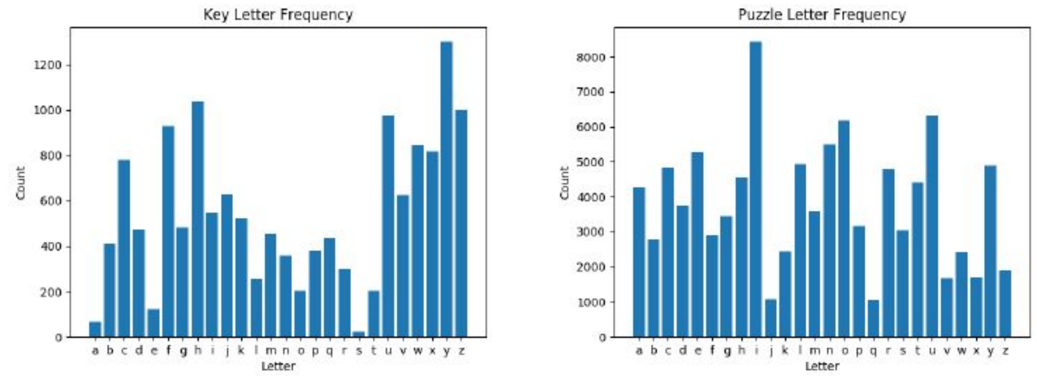
The Pangrams-First algorithm generated a total of 14,190 puzzles within the specified range of 15 to 60 answers. This would give our application enough daily content for approximately 38 years. Explorations of the puzzles with 15 to 60 answers that were generated from Pangrams-First algorithms is as follows:



(a) (b)



(c) (d)



(e) (f)

Figure 18: Analysis of the type of puzzles generated with the Pangrams-First algorithm, allowing puzzles with 15-60 answers. (a) shows the distribution of number of puzzle answers within the range, (b) shows the distribution of the maximum score, (c) shows the distribution of the number of pangrams, (d) shows the distribution of the number of perfect pangrams, (e) shows the frequency of key letters, and (f) shows the frequency of puzzle letters.

4.3.1.2 Analysis of Pangrams-First with 15 to 50 Puzzle Answer Range

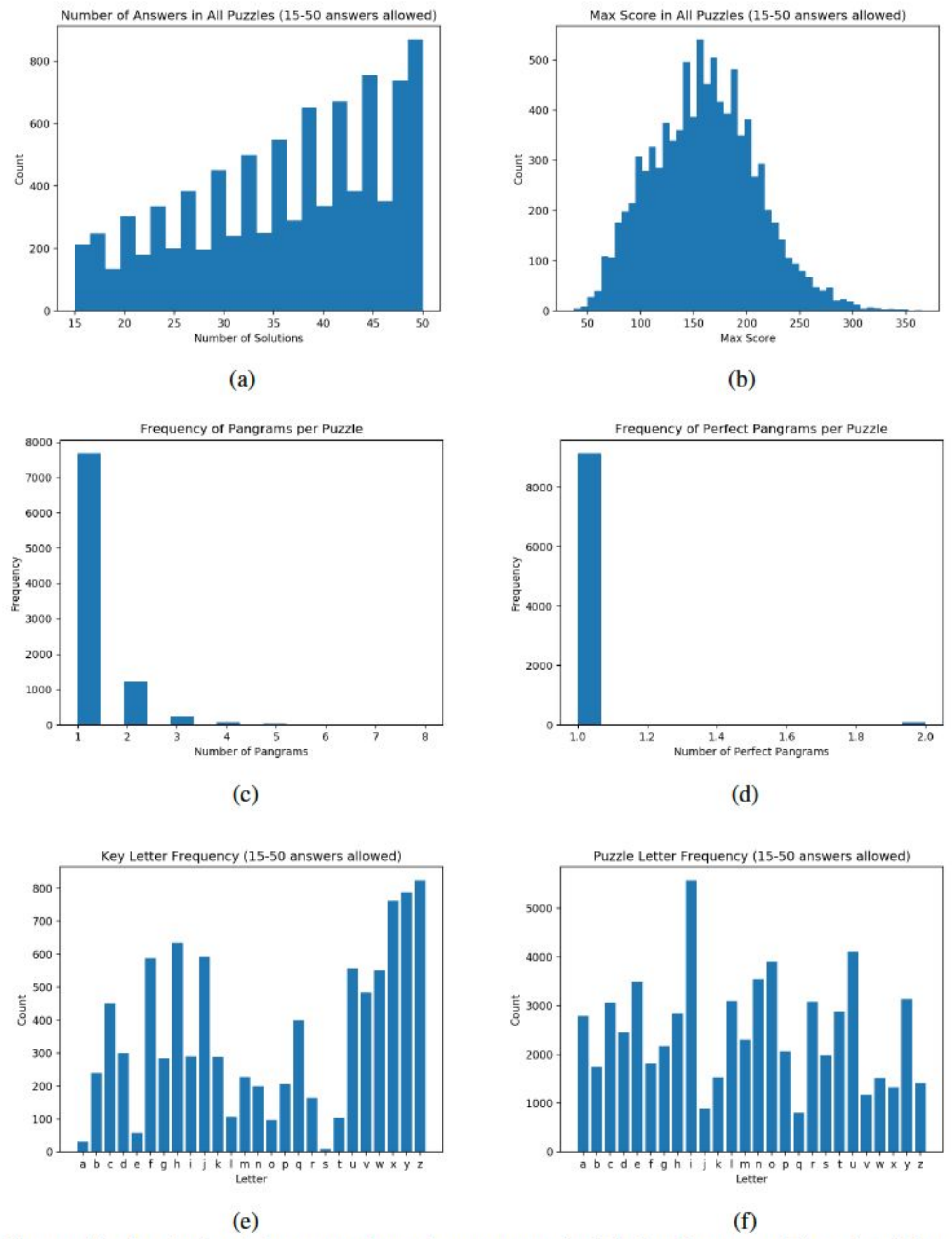


Figure 19: Analysis of the type of puzzles generated with the Pangrams-First algorithm, allowing puzzles with 15-50 answers. (a) shows the distribution of number of puzzle answers within the range, (b) shows the distribution of the maximum score, (c) shows the distribution of the number of pangrams, (d) shows the distribution of the number of perfect pangrams, (e) shows the frequency of key letters, and (f) shows the frequency of puzzle letters.

The number of puzzles generated with the Pangrams-First algorithm for 15 to 60 answers per puzzle exceeded our estimate. As a result, we re-ran the Pangrams-First algorithm using the modified range of 15 to 50 answers per puzzle.

The Pangrams-First algorithm generated 9,237 puzzles with 15 to 60 answers per puzzle. This would give our application enough daily content for about 25 years. Explorations of the puzzles with 15 to 50 answers that were generated from Pangrams-First algorithm is shown in Figure 19.

4.3.2 The Random Pangrams Algorithm

The second algorithm that was created to generate puzzles was the Random Pangrams algorithm, which was designed and implemented in Python.

Unlike the Pangrams-First algorithm, instead of beginning with pangrams, the general premise of the Random Pangrams algorithm is to start by randomly selecting seven letters from the alphabet, then try to build out the puzzle. The Random Pangrams algorithm is as follows:

```
Procedure RANDOM_PANGRAMS(dictionary)
Input:
   -  dictionary: cleaned dictionary output from dictionary cleaner.
Output: A file with all the generated puzzles (puzzle letters, the key
letter, and the puzzle solutions), separated by '#'.
1. Given the dictionary, find all the unique pangrams.
2. For 1 million times, pick 2 random vowels and 2 random consonants. The
letter 'y' is counted as a vowel.
   A. If the 7 randomly-drawn letters are a pangram: save it as a pangram.
   B. Otherwise: disregard the random guess.
3. For each unique letter in each pangram,
   B. Generate a puzzle such that the current letter is the key letter.
      i. Call the puzzle solver to generate all answers from this puzzle.
      ii. If the number of answers generated from this puzzle is within the
      specified range of answers allowed per puzzle: it is a valid puzzle.
      Therefore, write the puzzle letters, the key letters, and the puzzles
      solutions to the file.
      iii. Otherwise: the puzzle is invalid as should be disregarded.
```

For a fair comparison between algorithms, the Random Pangrams algorithm was run using the same puzzle answer ranges that were used to investigate the Pangrams-First algorithm.

4.3.2.1 Analysis of Random Pangrams with 15 to 60 Puzzle Answer Range

The Random Pangrams algorithm generated 6,107 puzzles within the 15 to 60 answer range. This would provide enough daily content for about 16 years. Further, from the 1 million guess attempts, the Random Pangrams algorithm constructed 10,437 unique pangrams out of the 24,575 unique pangrams possible in our dictionary. Approximately 1% of the 1 million guesses resulted in an actual pangram from the dictionary. Explorations of the puzzles with 15 to 60 answers that were generated from the Random Pangrams algorithm is shown in Figure 10:

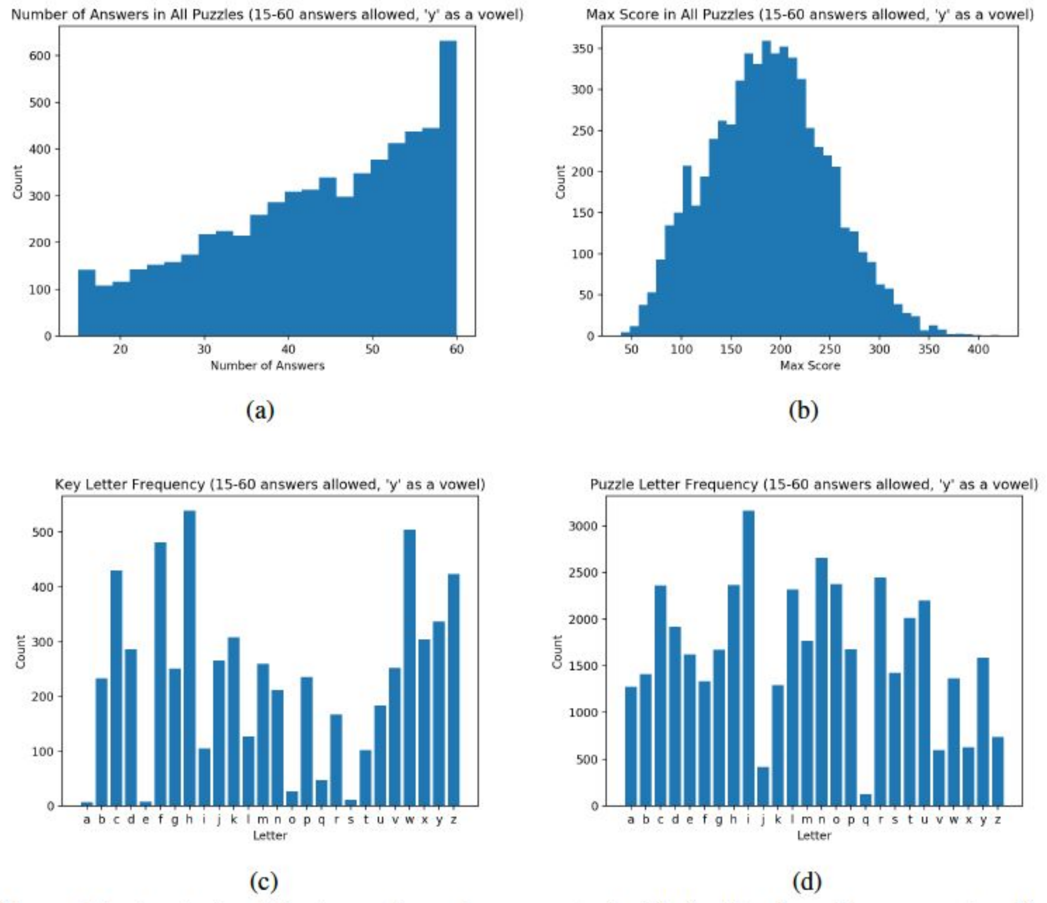
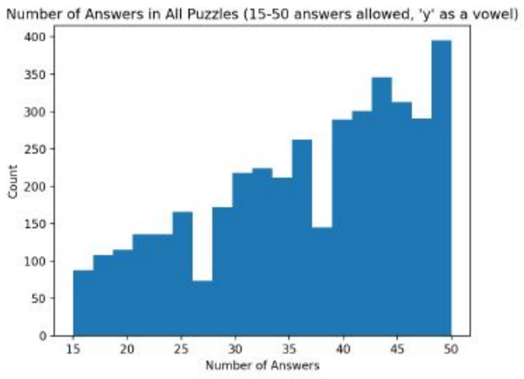


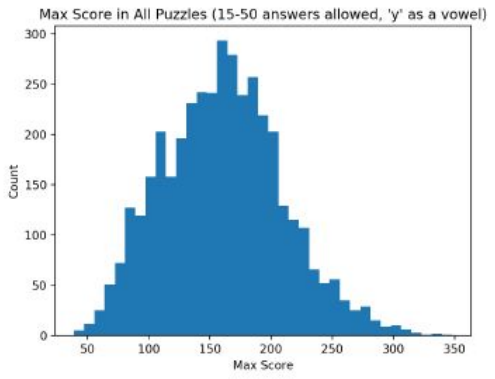
Figure 20: Analysis of the type of puzzles generated with the Random Pangrams algorithm, allowing puzzles with 15-60 answers. (a) shows the distribution of the number of puzzle answers within the range, (b) shows the distribution of the maximum score, (c) shows the frequency of key letters, and (d) shows the frequency of puzzle letters for the puzzles generated.

4.3.2.2 Analysis of Random Pangrams with 15 to 50 Puzzle Answer Range

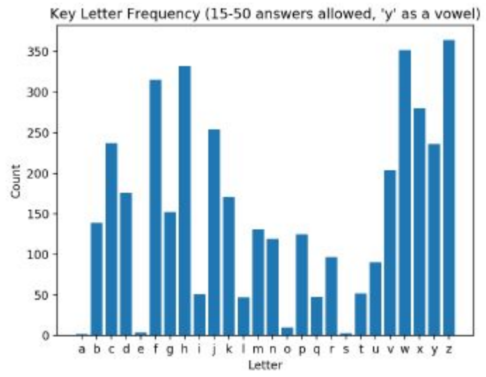
The Random Pangrams algorithm generated 3,991 puzzles within the 15 to 50 answer range. This would provide enough daily content for about 10 years. Further, from the 1 million guess attempts, the Random Pangrams algorithm constructed 10,428 unique pangrams. This differs from the 15 to 60 run by only 9 pangrams. Again, approximately 1% of the 1 million guesses resulted in an actual pangram from the dictionary. Explorations of the puzzles with 15 to 50 answers that were generated from the Random Pangrams algorithm is as follows:



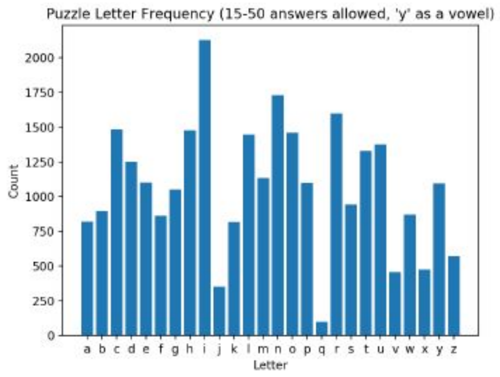
(a)



(b)



(c)



(d)

Figure 21: Analysis of the type of puzzles generated with the Random Pangrams algorithm, allowing puzzles with 15-50 answers. (a) shows the distribution of the number of puzzle answers within the range, (b) shows the distribution of the maximum score, (c) shows the frequency of key letters, and (d) shows the frequency of puzzle letters for the puzzles generated.

4.3.3 Comparison of the Pangrams-First and Random Pangrams Algorithms

Both algorithms considered, the Pangrams-First and Random Pangrams algorithms have varying results and efficiencies. Only about 42% of the 24,575 possible pangrams in our dictionary were correctly generated by the Random Pangrams algorithm in both the 15 to 60 and 15 to 50 runs. Further, only about 1% of the Random Pangrams guesses were successful in generating a pangram. This resulted in the Random Puzzles algorithm generating about 2.3 times fewer puzzles than that of the Pangrams-First algorithm. As such, the Random Pangrams algorithm has a shorter run time, while the Pangrams-First algorithm generates more puzzles and is an exhaustive puzzle generation of our dictionary within the specified range. The comparison between the performance of the Pangrams-First and the Random Pangrams algorithms is shown in Figures 22 and 23.

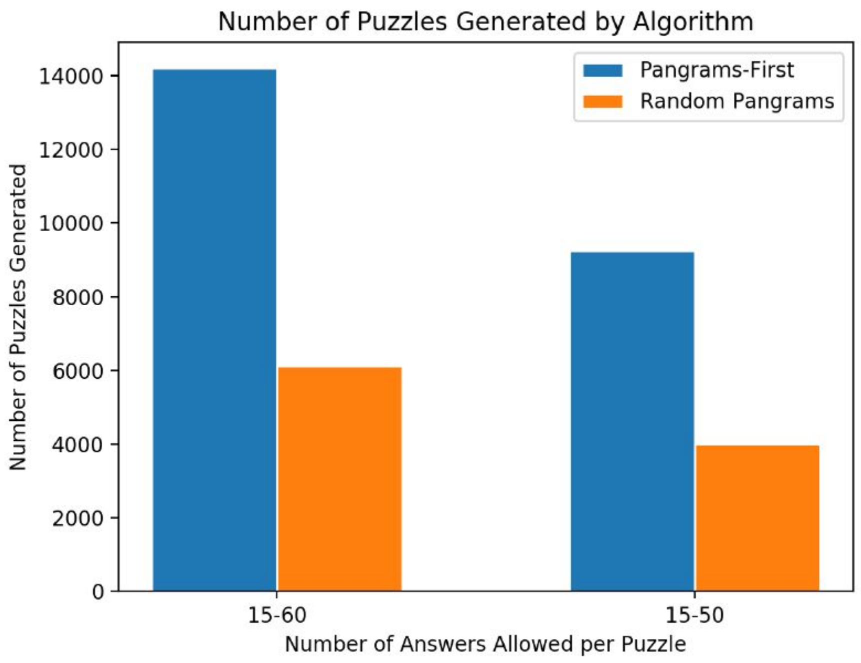


Figure 22: The performance of the two puzzle generation algorithms, Pangrams-First algorithm and Random Pangrams algorithm, in terms of the number of puzzles generated. The highest yield of puzzles provides 38 years of content, while the lowest yield provides only 10. This range improves on the brute-force 471 years.

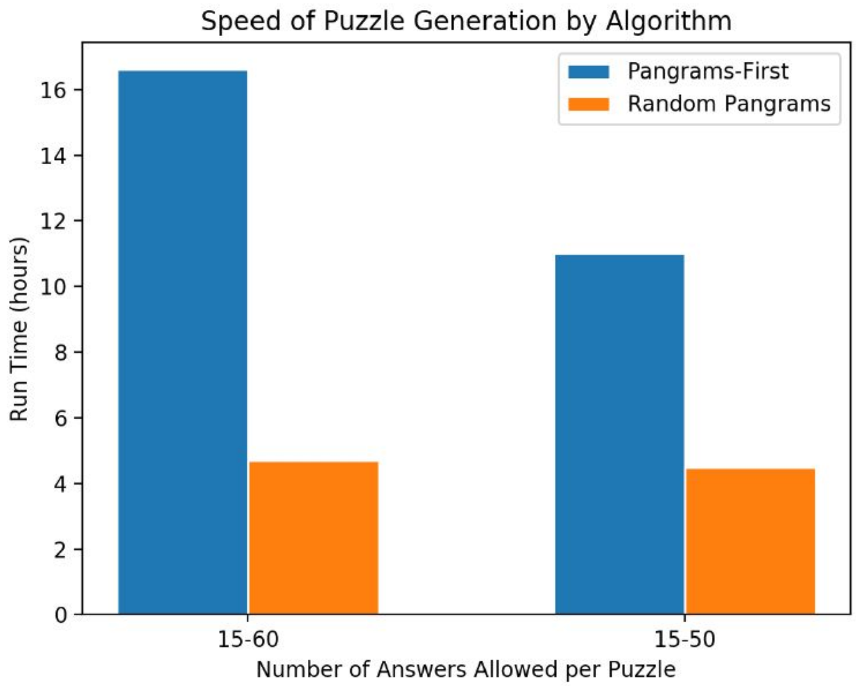


Figure 23: The performance of the two puzzle generation algorithms, Pangrams-First and Random pangrams, in terms of total run time.

4.3.3 Selection of Puzzle Generation Algorithm

Our explorations reveal that we were successful in restricting the number of puzzles generated with our dictionary, while still maintaining the most suitable puzzles for our application. In other words, we determined that both the Pangrams-First and Random Pangrams puzzle generation algorithms and both the 15 to 60 and 15 to 50 puzzle-answer ranges were viable.

With the results of the Pangrams-First algorithm, we found that our dictionary can produce a maximum of 9,237 puzzles (25 years of daily content) with 15 to 50 answers, and 14,190 puzzles (38 years of daily content) with 15 to 60 answers. In comparison, the Random Pangrams algorithm generated a total of 6,107 puzzles (16 years of daily content) within the specified range of 15 to 60 answers per puzzle, and 3,991 puzzles (10 years of daily content) within the range of 15-50 answers per puzzle.

Both of the findings from the puzzle answer ranges are a major improvement from the brute-force exhaustive puzzle generation algorithm, which generates an excessive amount (approximately 471 years' worth) of daily puzzles. However, for both algorithms, the puzzle-answer constraint will be set at 15-50 answers per puzzle, not at 15-60. This is because 25 years of daily content is sufficient for our game.

Although our explorations found that both the Pangrams-First and Random-Pangrams algorithms were viable for our team going forward, we have decided to use the Pangrams-First algorithm for our application. The Pangrams-First algorithm will be used to pre-generate all the puzzles for our game once and for all. As we are planning to generate puzzles only once in the lifetime of our application, and not generate puzzles on the fly, this has led us to favor Pangrams-First over Random Pangrams.

4.4 Puzzle Player

The Puzzle Player is the final main component of our application. It is the engine for our application. The Puzzle Player was implemented in a combination of HTML, CSS, and Javascript for the frontend and PHP for the mid-tier in order for the frontend to talk to the backend (See Appendix Section 4). Our algorithm for the puzzle player is as follows:

```
Procedure: PUZZLE_PLAYER(puzzle, user_input)
Input:
   -  puzzle: randomly selected puzzle and its solutions from output of
      Puzzle Generator
   -  user_input: user's guesses
Output:
   -  Updated user score
   -  List of user guesses
1. Start playing the game. While the user has not quit the game and has not
solved the puzzle (input all the puzzle answers),
   A. If user_input is invalid (less than 4 letters long, missing the key
      letter, is not in the puzzle answers, or is in previously inputted
      answers):
      i. Prompt the user to try again.
   B. If the guess is valid and a puzzle answer:
         a. Add the score of the puzzle answer to the user's score.
         b. Add the answer to the inputted answers list.
         c. If the guess is a pangram:
             i.   Add 7 points to score and inform user
2. Return user's final score and list of user's guesses.
```

4.4.1 User Tracking

When the user has successfully logged in, the game is constructed by loading in the *SESSION* variables (See Section 4.4.1.1). See Appendix Table 1 for a listing of the SESSION variables. These *SESSION* variables are loaded differently, depending on the following three scenarios (See Appendix 4.1.1 for a full description of how these Session Variables are loaded into the frontend, See Appendix Table 2 for the PHP variables that are loaded in from the session depending on the scenario, see Appendix Table 3 for the Javascript variables that load in the contents of these PHP variables):

(1) The user is in the middle of playing a puzzle.
(2) The user is a new user and has never played the game before.
(3) The user wants to abandon the current puzzle and start playing a new puzzle by pressing the "New Puzzle" button. This may arise when either the user has completed their current puzzle and wishes to start playing a new one, or when the user is in the middle of a puzzle and wishes to abandon it.

The implementation of each of these scenarios is described below. Scenarios #2 and #3 are similar in implementation, and as such are discussed in the same section.

4.4.1.1 Scenario #1: The User is in the middle of playing a puzzle.

In this scenario, since the user was in the middle of playing a puzzle on the last logout, the Puzzle Player restores the state of the game from the last log out. The Puzzle Player does this by loading in the puzzle and the guessed word list, which are stored in the SESSION, to PHP variables, which are then accessed by the frontend. See Appendix Tables 1, 2, and 3 for a listing of session, PHP, and Javascript variables.

4.4.1.2 Scenarios #2 & #3: The User is a New User, or the User requests “New Puzzle”.

In this scenario, when the user is a new user (i.e., does not have a puzzle they are currently playing), or when the user wishes to start a new puzzle by clicking the "New Puzzle" button, a new puzzle is randomly selected from the output generated by the Puzzle Generator (see Section 4.2).

A new random puzzle is selected randomly. The process of checking if the user has clicked the "New Puzzle" button is done via an HTTP request. In all scenarios, after the corresponding PHP variables are set, then they are also updated in the session. Additionally, after the PHP variables are set, they are sent and accessed by the frontend as JSON objects (this includes the puzzle, the key letter, puzzle solutions, the username, and the list of guessed words. See Appendix Tables 1, 2, and 3 for a listing of session, PHP, and Javascript variables.

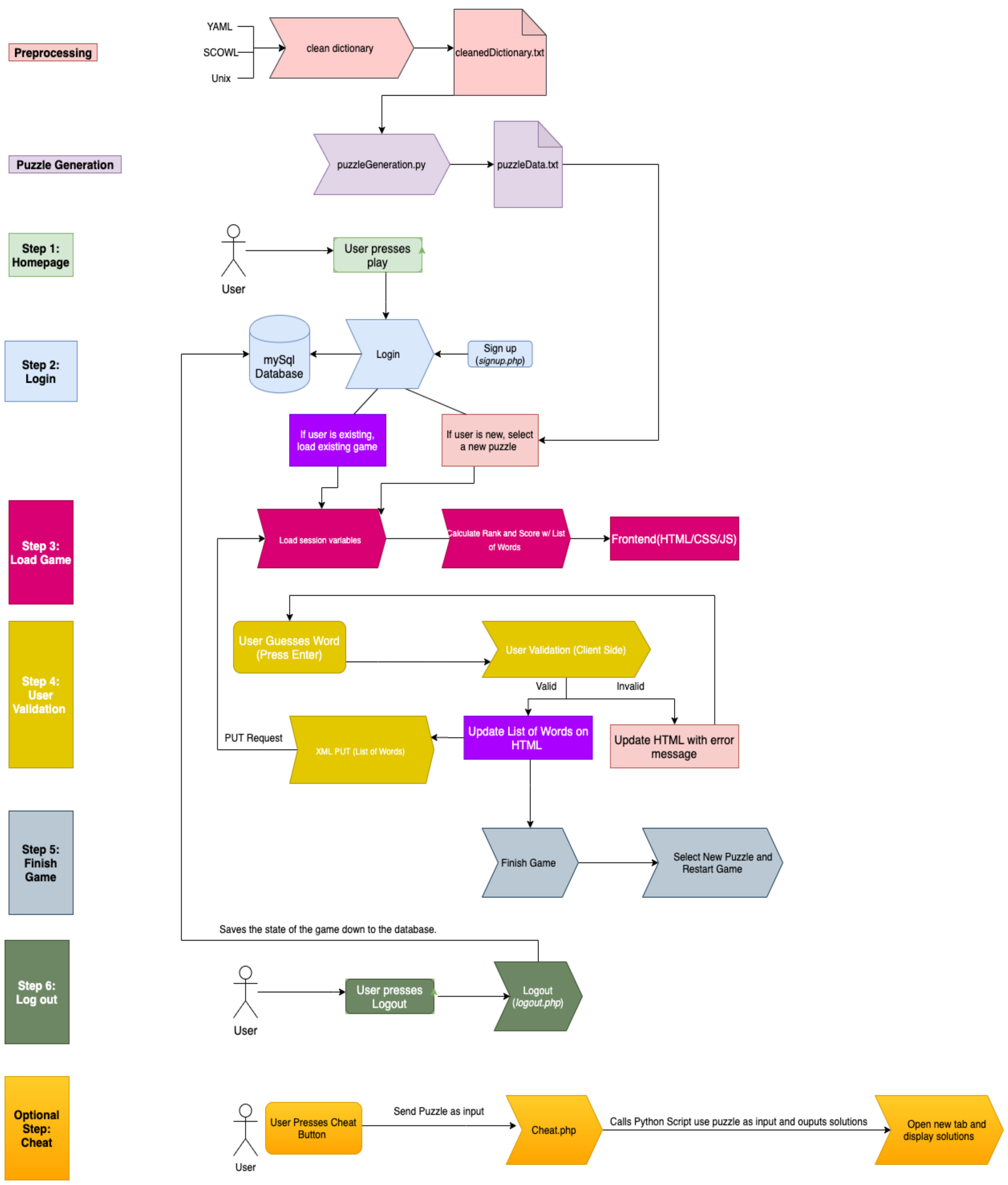


Figure 24: Diagram of the Workflow of the Entire Application

4.4.2 Front-end

A description of our frontend implementation follows. See Appendix 4.2 for a description of the front-end files for the Puzzle Player.

4.4.2.1 Loading Session Variables
To keep track of the state of the game when the user refreshes the screen, or returns to the game after logging out, the application stores variables in the browser *SESSION*. Variables stored in the session are used to populate the frontend. This way, the application is able to save the state of the user's game and use this data to reload the game when the user logs in. The *SESSION* variables (Appendix Table 1) are loaded into PHP variables (Appendix Table 2). These PHP variables, sent as JSON objects, are then pulled into the frontend via Javascript (Appendix Table 3).

The state of the game can be broken down into the following parts: the puzzle being played, the guessed words, the user's score and rank (see Figure 19). How the frontend of the application pulls in the state of the game from the *SESSION* variables is described in detail below.

1. The puzzle: A JSON object containing the puzzle letters, the key letter, and the puzzle solutions is passed from PHP to the frontend. Each of the indices of this JSON object are accessed by the frontend by being loaded into separate Javascript variables.

2. The list of guessed words: The list of guessed words is passed from PHP to the frontend as a JSON object, then loaded into a Javascript variable. This Javascript variable is then used to populate the list of guessed words in the frontend.

3. The score: Upon login or page refresh, using the scoring system described in Section 4.7.2.5, the score is computed based on the score of each word in the list of guessed words.

4. The rank: The rank is calculated by using the score.

4.4.2.2 Checking if a user guess is valid
When the user presses `ENTER`, the user's input is checked to see if it is a valid guess. The procedure for handling valid guesses is described in Section 4.7.2.4.

4.4.2.3 Handling valid guesses
If the user's input is checked as a valid guess, the guessed word is appended to the guessed word list. Then, a *GET* request is submitted to update the list of guessed words stored in the Session. The score and rank are also updated (see Section 4.7.2.6). Finally, the application checks if the user has completed the game (see Section 4.7.2.8).

4.4.2.4 Scoring Ladder
Like the New York Times, we have a scoring ladder that is calculated based on the max score of the given puzzle and the user's current score. Our scoring ladder is as follows:

Rank	Range
Beginner	0 - 10% of max score
Good Start	10-20%
Moving Up	20-30%
Good	30-40%
Solid	40-50%
Nice	50-60%
Great	60-70%
Amazing	70-80%
Genius	80-100%

Table 2: The scoring ladder for Bryn Mawr Bee.

The scoring ladder can be viewed when a user presses the yellow rank box as seen in Figure 11 and Figure 12. Whenever a user submits a valid guess, the user's score is updated accordingly by checking the updated score against the levels of the scoring ladder. If the user's new score falls within a different level, the user's rank is updated accordingly.

4.4.2.5 The Reshuffle button

See Appendix 4.2.2 for details on the Reshuffle button's implementation.

4.4.2.6 Checking if the user has completed the puzzle

Every time a user submits a valid guess, the application checks if the user has completed the puzzle, i.e., has correctly guessed all of the solutions associated with the current puzzle. The application determines if the user has completed the puzzle by checking if the guessed word list contains all the possible puzzle solutions. See Appendix 4.2.3 for more details.

When the user completes the puzzle, an alert message is output to the user (shown in Figure 21). Once the user exits from this alert, a new puzzle is generated by triggering a click event on the "New Puzzle" button. See Sections 4.7.2.2 and 4.7.2.9 for a description of how new puzzles are generated.

4.4.2.7 Cheating

When the user clicks on the "Give me the answers!" button, the application opens the cheat page in a new tab. The Puzzle Solver algorithm (Section 3.3) is used to output a file of the solutions for the puzzle. The solutions in the file are organized by the pangram(s) and then the other solutions of the puzzle (refer to Figure 22). Once this file is outputted, it is displayed on the cheat page. See Appendix 4.2.4 for a full description of the cheating implementation.

4.5 Database

The database, which contains all the user tracking data relevant to our application, is implemented with MySQL. The frontend of the application communicates with the database via PHP. See Appendix Section 5 for a full description of the files used to implement the database.

4.5.1 Database Configuration

The database itself is called *userInfoSB*. It contains one table, called *users*. Each row in the *users* table corresponds to one user in our application. The *users* table has five columns: *id*, *username*, *password*, *puzzle*, and *guessedAnswers*. As such, each user in the database has an associated username, password, puzzle, and guessedAnswers. The database is created via a combination of the MySQL command line client, and a PHP file. See Appendix 5 for a description of the file used to create the database. See Appendix 5.2 for a description of how to connect to the database.

The *users* table is created once in the lifetime of the application via the MySQL command line client. See Appendix 5.1 for details on the configuration of the database. A description of each column follows:

1. id: The *id* is the primary key to uniquely define each user in the database.

2. username: The user's username, unique to each user.

3. password: The user's password. Passwords are stored as a password hash.

4. puzzle: This field specifies the current puzzle the user is playing in the game. The *puzzle* column contains the puzzle letters, the key letter, and all the puzzle solutions for the currently-being-played puzzle.

5. guessedAnswers: This field contains the serialized `string` representation of the `array` of answers the user has correctly guessed from the currently-being-played puzzle.

The overall layout of the *users* table in the *userInfoSB* database, demonstrated for a single row, can be conceptualized as follows:

USER_X	ID	USERNAME	PASSWORD	PUZZLE	GUESSED ANSWERS

Table 3: An abstraction of a single row in our user tracking database, corresponding to one user. USER_X is meant to signify that the given row applies to some single user in the database.

4.6 Sign Up

After the user clicks on “Sign up now” from the login page (refer to description of the login page), the user is taken to the sign up page. Sign up is handled via a PHP file, which then modifies the database. See Appendix 5.3 for a description of the sign up. The sign up page requires the user to input a username, a password, and then to confirm the password.

When the user submits a username and password, the program checks if the username and password are valid. Once the username, password, and confirm password are validated, a new row is created in the *users* table with this username and password. The password is stored in the database as a password hash. However, if the inputted username or password is invalid, the sign up page outputs an error message to the user. Once the user is signed up, the user is redirected to the login page (see Section 4.6).

4.7 Login

After the user clicks on the “Let’s Play” from the landing page (refer to description of the landing page), the user is taken to the login page. Login is handled via a PHP file, which interacts with the database in order to validate the user’s credentials. See Appendix 5.4 for a description of the file used to implement logging in. The login page requires the user to input their username and password. If the user does not have an existing account, they can be redirected to the signup page by clicking the “Sign up now” button.

When the user submits their username and password, the program first checks that both inputs are not empty. Then the username and password are validated by checking the username and password stored in the database for that user via a SELECT statement. If the inputted username or password is not verified, the login page outputs an error message to the user.

A user’s status as “logged in” is handled via *SESSION* storage. The session variables used in the application are used to store the user’s credentials, and restore the state of the game from when the user last played. See Appendix Table 1 for a listing of the SESSION variables relevant to logging in. If the user kills the browser, their session ends, and they are logged out (i.e., the user will not be logged in if they reopen the browser). Killing the session, instead of logging out (see Section 4.8), does not save the state of the game. Once the user is logged in, the user is redirected to the puzzle player page (see Section 4.7).

4.8 Log Out

After the user clicks on the “Log out” button found in the main page, the user is logged out of the game. When the user logs out, the state of the game is saved down to the database. Log out is handled via a PHP file, which then interacts with and modifies the database. See Appendix 5.5 for a description of the file used to handle logging out.

To save the user’s current state in the game, the puzzle and guessedAnswers columns (see Table 2) for that user in the database are updated. This is done by loading in the puzzle, guessedWordList, and id SESSION variables from the session (see Appendix Table 1) into their relevant PHP variables (see Appendix Table 2), then executing an UPDATE statement for the puzzle and guessedAnswers columns in the database. Then, the user's session is destroyed and they are redirected to the landing page.

5. Future Work

If we had two more weeks to work on this, we would mostly want to iterate once more on our design and work on the difficulty level of the puzzles. Although our current design works well, it is ultimately a recreation of the NYTimes design, and we were always interested in creating a design that was unique to us or at least unique to Bryn Mawr. To iterate on design and slowly get to one we were happy with, as many real technologists do, would have been a very educational experience. As for difficulty level, we came up with the idea that users can alternate what difficulty they'd like their puzzle to be with our advisors. This would involve exploring how to classify how difficult puzzles are, actually classifying them as such, and building out front-end functions to allow users to choose which difficulty they'd like. It is definitely ambitious, but a very exciting prospect that we certainly would explore if given two more weeks. Lastly, we would want to quickly refactor our code as it is still in a quite primitive state.

If we had six more months to work on this, we would definitely want to develop a mobile application for our Bryn Mawr Bee. The design would lend itself to a mobile interface, as a touch screen would be very helpful in creating and submitting words by using the Hive. A mobile application has also been a goal of ours since the very beginning. If we had time after that, we would want to invest time in developing a version of Bryn Mawr Bee that is multiplayer. Many people already play NYTimes Crossword games with their friends, so investing in a multiplayer version would be very user-friendly and a great exercise for us as engineers. Finally, if we decided to start from scratch within the next six months, we would want to try rebuilding the application using React and Node.js, two technologies we have been interested in using since the beginning. Ultimately, however, we are extremely proud of our work, and look forward to any future work we may achieve together.

6. Summary

The Spelling Bee team set out to recreate the NYTimes Spelling Bee application. We did this in order to have the experience of building out and iterating on a full stack application. We did this by creating algorithms and databases in Python and MySQL, building out a server-side in PHP, and a front-end with HTML, CSS, and Javascript. We were able to build out a fully functioning application inspired by the design of NYTimes. In the future, we hope to iterate on the design to find a more unique and Bryn Mawr inspired front-end, and potentially work to create more ways to play the Spelling Bee game.

References

[1] "Spelling Bee." *The New York Times,*
https://www.nytimes.com/puzzles/spelling-bee
Accessed on 3 February 2020.

[2] Norvig, Peter. *Natural Language Corpus Data: Beautiful Data*, 8 July 2008,
https://norvig.com/ngrams/
Accessed on 3 February 2020.

[3] *SCOWL (And Friends)*, SOURCEFORGE,
http://wordlist.aspell.net
Accessed on 3 February 2020.

[4] "Version 6 of the 12dicts word lists." *SCOWL (And Friends)*, SOURCEFORGE,
http://wordlist.aspell.net/12dicts-readme/
Accessed on 3 February 2020

Appendix

1. Dictionary

This Python script, as well as the sub-dictionaries and final dictionary files, are described below.

1. *gitWords.txt*: The "2of12inf" list of words from SCOWL (And Friends).

2. *scrabbleWords.txt*: The YAWL (Yet Another Word List) word list.

3. *unix.txt*: The UNIX words dictionary.

4. *makeCleanDictionary.py*: This Python script is run once for the lifetime of the application, and is used to create the cleaned, final dictionary for use in our application. Its input is three dictionary files, *gitWords.txt*, *scrabbleWords.txt*, and *unix.txt*, expected as command line arguments. It merges all three dictionaries, then iterates through each word in the combined dictionary. The clean words are written to the file *cleanedDictionary.txt*. It is run via the command line, using the following command: `python3 makeCleanDictionary.py <dictionary file1> <dictionary file2> <dictionary file3>`. More specifically, using our dictionary files, `python3 makeCleanDictionary.py gitWords.txt scrabbleWords.txt unix.txt`.

5. *cleanedDictionary.txt*: The final dictionary outputted from *makeCleanDictionary.py*.

2. Puzzle Solver

The list of solutions are written to the *puzzleData.txt* file. The Puzzle Solver is also run in the *cheatPuzzleSolver.py*, which is run by *cheat.php* when the user wishes to cheat in the game.

3. Puzzle Generator

To implement puzzle generation, the files used were *puzzleGeneration.py* and *randomPangramsOnTheFly.py*. However, since we decided to use the Pangrams-First algorithm to generate puzzles, *randomPangramsOnTheFly.py* is not actually used in our application. As such, *randomPangramsOnTheFly.py* will not be discussed further.

1. *puzzleGeneration.py*: This Python file is run once for the lifetime of the application. It implements the Pangrams-First algorithm (see Section 3.4.1) to generate the list of all possible puzzles. Its input, expected as a command line argument, is the cleaned dictionary file (*cleanedDictionary.txt*) that is generated by *makeCleanDictionary.py*. It writes its output to the file *puzzleData.txt* (described next). *puzzleGeneration.py* is not called by any other files in our application. It is run via the command line, using the following command: `python3 puzzleGeneration.py <cleaned_dictionary_file>`. More specifically, using the

dictionary generated for our application, `python3 puzzleGeneration.py cleanedDictionary.txt`.

2. *puzzleData.txt*: This file is the output of *puzzleGeneration.py*. This file consists of all the generated puzzles, separated by "#". For each hashtag-separated puzzle in the file, the first line consists of the hashtag sign "#", the second line is the seven puzzle letters, the third line is the key letter of the puzzle, and the following lines are the puzzle solutions. Refer to Figure 8. The *puzzleData.txt* file is used by the Puzzle Player to select a random puzzle.

4. Puzzle Player

4.1 User Tracking

The user tracking portion of the Puzzle Player is implemented in the PHP code in *generatePuzzleJSON.php*. User tracking from the frontend relies on SESSION variables to store the state of the game and maintain gameplay. These SESSION variables are then used to save the state of the game down to the database, once the user has logged out.

The session variables used in the application are as follows:

SESSION Variable	**Contents**
loggedin	stores the login status of the current user. This variable is set to *true* once the user has logged in. This variable is a `boolean`.
id	stores the current user's ID. This variable is an `integer`.
username	stores the current user's username. This variable is a `string`.
puzzle	stores the *puzzle* field from the database. This variable is a `string`. If the user has never played the game before, this variable is `NULL`.
guessedWordList	stores the *guessedAnswers* field from the database. This variable is an `array`. If the user has never played the game before, this variable is an empty array.

Table 1: Session variables and their contents.

A table describing the relevant PHP variables that are referenced in the scenarios below follows.

PHP Variable	Contents, depending on the scenario
`$randomPuzzleString`	Either the *puzzle SESSION* variable (see Table 3), or a new, randomly generated puzzle.
`$guessedWordList`	Either the *guessedWordList SESSION* variable (see Table 3), or an empty `array`. This variable is encoded as a JSON object, which contains an `array` of words the user has guessed already.
`$username`	The *username SESSION* variable (see Table 3). This variable is encoded as a JSON object.
`$data`	The JSON representation of `$randomPuzzleString`. This JSON object contains three entries, in the following order: the puzzle letters, the key letter, and the `array` of puzzle solutions.

Table 2: PHP variables relevant to user tracking. These variables are accessed in the frontend to play the game.

4.1.1 Loading Session Variables

How the frontend of the application pulls in the state of the game from the *SESSION* variables is described in detail below.

1. *The puzzle*: The PHP variable `$data` (described in Table 4), is passed to the frontend. `$data` contains all of the information needed by the frontend to populate the hive. Each of the indices of the JSON object `$data` are loaded into a separate Javascript variable. In particular, `$data.puzzleLetters` (the letters of the puzzle) is stored in the Javascript variable `puzzleLetters`, `$data.keyLetter` (the key letter of the puzzle) is stored in the Javascript variable `keyLetter`, and `$data.solutions` (the solutions to the puzzle) is stored in the Javascript variable `solutions`. Once each of the pieces of the JSON object are loaded into their appropriate Javascript variables, these Javascript variables are then used to populate the hive.

2. *The list of guessed words*: The PHP variable `$guessedWordList` (described in Table 4), is passed to the frontend as a JSON object. `$guessedWordList` is loaded into the Javascript variable `guessedWordList`. This Javascript variable is then used to populate the list of guessed words in the frontend.

3. *The score*: The score is calculated by using the `guessedWordList`. This is because the score itself is not stored in the *SESSION*, nor is it passed from PHP to the frontend. Upon login or page refresh, using the scoring system described in Section 4.7.2.5, the score is computed based on the score of each word in `guessedWordList`.

4. *The rank*: The rank is calculated by using the score.

4.2 Front end

The front end of the Puzzle Player is implemented in the files *generatePuzzleJSON.php* (which handles the user interaction), and *index3.css* (for styling).

A listing of the Javascript variables that are loaded in from the SESSION are as follows:

Javascript Variable	**Contents**
puzzleLetters	The letters of the puzzle. Loaded in from `$randomPuzzleString`.
solutions	The solutions to the puzzle. Loaded in from `$data.solutions` (the solutions field from the `$data` PHP variable).
keyLetter	The key letter of the puzzle. Loaded in from `$data.keyLetter` (the solutions field from the `$data` PHP variable).
guessedWordList	The list of guessed words. Loaded in from the `$guessedWordList` PHP variable.
username	The user's username. Loaded in from the `$username` PHP variable.

Table 3: Javascript variables relevant to user tracking. These variables are what the frontend uses to play the game.

4.2.2 The Reshuffle button
When the Reshuffle button is clicked, the `reshuffle` function is called in *generatePuzzleJSON.php*. The `reshuffle` function takes in the variable `puzzleLetters` that originally went into the puzzle: a string of the 6 non-key letters. The `reshuffle` button uses the `.split()`, `.sort()`, and `.join()` built-in functions to split each letter into an array, randomly sort that array using `Math.random()`, and join them back into one string. Since the Javascript already dynamically sets the innerHTML of each button to its corresponding letter in puzzleLetters, the change that `reshuffle` makes is reflected visually as soon as it is called (and subsequently changes the `puzzleLetters`).

4.2.3 Checking if the user has completed the puzzle

Every time a user submits a valid guess, the application checks if the user has completed the puzzle, i.e., has correctly guessed all of the solutions associated with the current puzzle. This is handled in the `validate()` function.

The application determines if the user has completed the puzzle by checking if the length of the Javascript variable `guessedWordList` is equal to the length of the Javascript variable `solutions`. `guessedWordList` contains all the puzzle solutions the user has correctly guessed, and `solutions` contains all the puzzle solutions (as described in Section 4.7.2.2). This is handled in the `checkEndGame()` function.

If `checkEndGame()` returns true, then the user has completed the puzzle. When the user completes the puzzle, an alert message is output to the user (shown in Figure 21). Once the user exits from this alert, a new puzzle is generated by triggering a click event on the "New Puzzle" button. See Sections 4.7.2.2 and 4.7.2.9 for a description of how new puzzles are generated.

4.2.4 Cheating

When the user clicks on the "Give me the answers!" button, the application redirects to *cheat.php,* opening the cheat page in a new tab. When *cheat.php* is run, it submits a *POST* request for the current puzzle letters. Then the *cheatPuzzleSolver.py* is called, a Python script which is run with the puzzle letters and the final dictionary. *cheatPuzzleSolver.py* uses the Puzzle Solver algorithm (Section 3.3) to output a file of the solutions for the puzzle. The solutions in the file are organized by the pangram(s) and then the other solutions of the puzzle (refer to Figure 22). Once this file is output by the *cheatPuzzleSolver.py*, it is displayed on the cheat page.

5 Database

The database is created in the file *createDatabase.php*, which is run only once in the lifetime of the application. This *createDatabase.php* file is run in the browser.

5.1 Database Configuration

The database itself is called *userInfoSB*. It contains one table, called *users*. Each row in the *users* table corresponds to one user in our application. The *users* table has five columns: *id*, *username*, *password*, *puzzle*, and *guessedAnswers*. As such, each user in the database has an associated username, password, puzzle, and guessedAnswers. The *users* table is created once in the lifetime of the application via the MySQL command line client. A description of each column follows:

1. id: The *id* is the primary key to uniquely define each user in the database. It is an integer value.

2. username: The user's username. It is stored as a variable-length string that may contain up to 50 characters. The username must be a unique value in the table, and must be non-null.

3. password: The user's password. It is stored as a variable-length string that may contain up to 255 characters. It must be non-null. Passwords are stored as a password hash.

4. puzzle: This field specifies the current puzzle the user is playing in the game. It is stored as a variable-length string that may contain up to 5000 characters. The *puzzle* column contains the puzzle letters, the key letter, and all the puzzle solutions for the currently-being-played puzzle.

5. guessedAnswers: This field contains the serialized `string` representation of the `array` of answers the user has correctly guessed from the currently-being-played puzzle. It is stored as a variable-length string that may contain up to 5000 characters. When storing or retrieving the *guessedAnswers*, the data must be serialized into a `string` or unserialized to an `array` accordingly.

5.2 Connecting to the Database

All php files that wish to modify the *userInfoSB* database do so by first connecting to the database. This is achieved by calling the *connectToDB.php* file. The *connectToDB.php* file is called in *signup.php* (see Section 4.5), *login.php* (see Section 4.6), and *signout.php* (see Section 4.7).

5.3 Signup

Sign ups are handled in the file *signup.php*. A description of *signup.php* follows.

1. *signup.php*: This PHP file connects to the database with the *connectToDB.php*. This connection is necessary in order to update the database when new accounts are created. When the user submits a username and password, the program checks if the username and password are valid. The username and password are submitted via a *POST-request* in a Bootstrap form. A username is valid if it is nonempty and if it is not a username that is already present in the database. If the username is valid, then the program checks if the password is valid. A password is valid if it is six or more characters long. Lastly, the program checks if the "confirm password" matches the password. Once the username, password, and confirm password are validated, the *signup.php* creates a new row in the *users* table with this username and password. The password is stored in the database as a password hash. However, if the inputted username or password is invalid, the sign up page outputs an error message to the user. Error handling is achieved in *signup.php* via a Bootstrap form.

5.4 Login

A description of *login.php* follows.

1. *login.php*: This PHP file connects to the database with the *connectToDB.php*. This connection is necessary in order to search in the database to validate the username and password a user inputs. The username and password are submitted via a *POST-request* in a Bootstrap form. When the user submits their username and password, the program first checks that both inputs are not empty. Then the program checks if the username matches an existing username in the

database via a *SELECT* statement. If the *SELECT* statement returns a row from the database, then the password is verified. The password is verified by checking that the inputted password matches the password hash that is stored with the inputted username. If the password is verified, then the *SESSION* variables (*loggedin*, *id*, *username*, *puzzle*, *guessedWordList*) are updated to hold their corresponding fields from the database. When restoring the *guessedWordList SESSION* variable, as the *guessedWordList SESSION* variable is an array, the *guessedAnswers* field from the database is unserialized from its `string` representation back into an `array` (first checking that the field is non-null). If the user has never played the game before (i.e., is a new user and has just come from the sign up page), the *puzzle* and *guessedWordList SESSION* variables are set to null or empty, respectively. However, if the inputted username or password is not verified, the login page outputs an error message to the user. Error handling is achieved in *login.php* via a Bootstrap form.

5.5 Logout

1. *logout.php*: This PHP file connects to the database with the *connectToDB.php*. To save the user's current state in the game, the *puzzle* and *guessedAnswers* columns (see Table 2) for that user in the database are updated. This is done by loading in the *puzzle*, *guessedWordList*, and *id SESSION* variables from the session (see Table 3). Then, an *UPDATE* statement is prepared with the user's unique *id* and their *puzzle*. Once this *UPDATE* statement has successfully executed on the database and the *puzzle* column has been updated to contain the *puzzle SESSION* variable. The same process happens with the *guessedWordList SESSION* variable and the *guessedAnswers* column. Once this second *UPDATE* statement is successfully executed on the database, the state of the game has been saved down to the database. Therefore, the user's session is then destroyed, and the user is redirected to the landing page.